IN3030/IN4330 BUNDLE FOR EXAM 2026

This is a bundle of most – if not all – material that has been presented in the course.

GOOD LUCK ☺

Sincerely Eric & Paulo

Effektiv Parallellprogrammering, kompendium i IN3030/4330

av

ARNE MAUS

PT (Programmeringsteknologi),
Institutt for informatikk,
Universitetet i Oslo

15. mars 2024

(Ris og ros, feil, forslag til forbedringer/utvidelser/strykninger og andre kommentarer sendes: arne@maus.no)

Innledning

Dette er en tredje oppdatering av et kompendium i parallellprogrammering i Java på en vanlig flerkjerne maskin med felles lager. Dette stoffet foreleses på Institutt for informatikk i kursene IN3030/4030 på bachelor- og master-nivå. Det poengteres at dette kompendiet er et tillegg til kurset som kan hjelpe til å forstå hvorfor dagens datamaskiner har stadig flere kjerner, hvilke mekanismer disse inneholder som gjør at programmer kan gå fortere alt etter hvordan vi skriver våre parallelliserte algoritmer. Å parallellisere kjente algoritmer på en datamaskin med flere prosessorkjerner synes de siste 10-20 årene å være den mest generelle løsning på svært mange typer av problemer innen ingeniør-beregninger, administrasjon og KI. Når vi ser på antall ulike typer av datamaskiner solgt og installert i serverparker og i generelt salg som mobiltelefoner/ PC-er og arbeidsstasjoner, finner vi at multikjerne maskiner med felles lager nå er klart dominerende i antall; mens maskiner som skal brukes f.eks. til KI-anvendelser (både opplæring av KI -modellene og senere kjøring av disse KI-modellene) er opp til en faktor 1000 ganger raskere ved å bruke spesiell KI-maskinvare på visse problemer. Det er imidlertid viktig, uansett hvordan markedet for ulike typer av maskiner går, bør man lære grunnleggende parallellisering i dette kurset før man gå løs på mer spesialiserte brikker som KI-brikker.

Studentene har som en forutsetning hatt det første kurset i Algoritmer og Datastrukturer og har hatt to introduksjonskurs i OO (objektorientert) programmering hvor de også har sett noen få eksempler på parallelle programmer i Java, men ingen inngående opplæring i parallellisering. Kurset IN3030 starter derfor på bunnen med hva tråder er, men kommer ganske fort opp på et rimelig avansert nivå i parallellisering av algoritmer. Dette kompendiet er tenkt som støttenotat og utdyping av forelesningene og forelesningsfoilene i kurset.

Vi programmerer i dette kurset typisk for en vanlig PC /mobiltelefon med 4-16 kjerner, men programmene i denne boka vil også virke effektivt på mer uvanlige prosessorer som Intel Xeon Phi med 62 kjerner eller store server-prosessorer med 32-64 kjerner med felles hukommelse (som Graviton3 brikken med 64 kjerner laget med 55 milliarder transistorer på én brikke). Det som her foreleses her er ideen om en datamaskin med felles hukommelse for flere prosessorkjerner, og at det er flere nivåer, typisk 3, med cache-hukommelse på hver kjerne (delt eller ikke delt cache er ikke viktig her). Cache hukommelse forklares senere. Valg av programmeringsspråk, Java, er nok viktig for eksemplene i dette kompendiet, men det er få kjente grunner til at tilsvarende og nær identiske programmer ikke skulle være like effektive i Scala, C++, C#, Object C , Kotlin eller andre objektorienterte språk med tråder og med et felles adresserom (i hovedhukommelsen og cachene).

Det er to krav vi stiller til de parallelle programmene vi skal lage:

- De skal være **riktige** – produsere samme resultater som en riktig og rask sekvensiell løsning
- De skal være **mer effektive, klart raskere** enn den sekvensiell løsning av samme problemet

Viktig i dette kurset er hvordan man parallelliserer et riktig, sekvensielt program + og lage egne parallelle algoritmer. I dette kurset begrenser *vi oss til at* ***våre parallelle programmer skal gå på én PC med ett felles lager.*** Vi vil altså ikke ta opp bruk flere andre aktuelle typer av maskiner, som også nyttes til å løse parallelle problemer:

Grunnen til at vi ikke omhandler bruk av grafikk-kort GPU med mange tusen enkle prosessorer er at programmering av de alle krever en helt spesiell kode og at de løser bare visse typer av problemer svært effektivt (mye data, med samme instruksjon utført på alle disse data i parallell). Andre spesielle brikker er Field programmerbar array, GPU og ASIC -kretser (Se referansen nest sist i dette kompendiet til en lang drøfting av ulike brikker og trender innen elektronikkindustrien med 225 referanser!).

Dagens virkelig store dataanlegg består klynger av flere millioner PCer koblet sammen i et raskt nett. Alle de metodene som læres i IN3030/4030 er aktuelle der, i tillegg er det her mange ekstra problemer man får med en slik komplisert maskin – f.eks. strømforbruket (verdens største slik klynge i 2022 bruker ca. 30 000 KW), med tilsvarende store luft-kjølingssystemer. For slike klynger trenger vi også teknikker og programvare til det å spre ut en beregning ut på så mange enkelt-maskiner over et nettverk med den relativt store forsinkelsen i datanettet mellom maskinene (ca. 1000 ganger så stor forsinkelse sammenlignet med tidene til lesing og skriving i én PCs sin hovedhukommelse), og særlig det til sist å få samlet så mange delberegninger til ett, felles svar krever egne programmerings systemer og filsystemer.

Dere skal altså lære de mange problemene vi støter på når vi bruker på én multikjerne PC og hvordan disse kan relativt enkelt kan løses med parallellisering av et sekvensielt program. Kurset er *empirisk* (med tidsmålinger), og ikke basert på en teoretisk modell av parallelle beregninger. Vi oppfatter programmet som en god nok modell av det problemet vi skal løse. Vi trenger ingen modell av modellen. Siden slike teoretiske modeller, særlig ulike varianter av PRAM-modellen (Parallel Random Access Machine) brukes i mange lærebøker i parallelprogrammering, vil vi i dette kurset flere ganger kommentere hvor feilaktige anslag disse teoretiske modellene er. De kjøretider PRAM anbefaler kan bli svært misvisende. PRAM modellen er mer utfyllende behandlet i kap. 11.

N.B. En liten irritasjon: Java har gått over fra ASCII til UTF-8 koding av Java-programmer og aksepterer i kompilatoren **ikke lenger bokstavene**: *æ, ø , å*. Gir stygge feilmeldinger. Tar du klipp og lim av programmer fra denne teksten må du f.eks. bytte ut: *æ* med ae, *ø* med oe , *å* med aa. Også i kommentarer.

1. OM VESENTLIGE FORHOLD SOM PÅVIRKER EKSEKVERINGSTIDENE

Når vi kjører våre programmer vet vi at vi skriver det i et programmeringsspråk, i vårt tilfelle Java, som så kjører på en CPU-basert maskin med sine maskininstruksjoner og oppbygging med ulike typer elektronikk.

I det følgende vil vi gå gjennom de (forbausende mange) mekanismer, både i elektronikken og i Java som vil redusere eksekveringstidene i meget stor grad, og som vi kan benytte oss av når vi skriver programmer. Det presiseres at det er ikke hele datamaskinen eller hele Java som beskrives – bare de vesentligste mekanismene som vi som programmerere kan bruke til å få raskere programmer - både sekvensielle og særlig for oss – parallelle programmer.

Vi begynner først med datamaskinen før vi tar for oss Java, som nå er under stor utvikling med stadig nye begreper som er innført i språket. I den grad slike nye begreper i også vil kunne endre hastigheten av våre programmer negativt, er et vesentlig eksempel beskrevet i kap. 12.

1.1 DATAMASKINENE

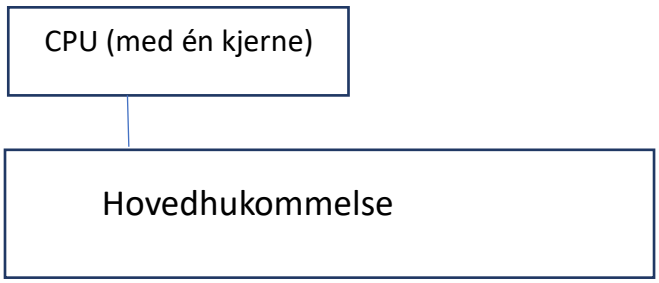
Hukommelses-systemet

a) **Lageret er byte-adressert**
Lageret i alle Intel, AMD og Apple/Arm -maskiner er byte-adressert – med 8 bit i en slik byte. Det betyr at den minste data-enheten vi kan lese og skrive fra CPUen til hukommelsen er én byte lang– f.eks. er en `byte` variabel i Java er 8 bit lang (eks: `byte a;`), mens en vanlig `int` er 4 byter. Dette er kanskje ikke så viktig å vite i sekvensielle programmer, mens i parallelle programmer er det avgjørende at to parallelle tråder ikke samtidig prøver å skrive på samme byte i lageret, Da er det viktig å vite at hvis to tråder samtidig ønsker å skrive eller endre på ett av bit-ene i en slik byte samtidig, går det galt selv om det er ulike bit vi ønsker å endre på i denne byten.

b) **Cache-systemet.**
Når vi regner med data i programmet vårt, f.eks. skal utføre setningen: `a = b + c`, tenker vi at alle data er i en stor hukommelse og at variablene a, b og c der har hver sin plass i den. Vi leser/kopierer først verdiene av b og c fra hukommelsen, legger disse sammen og skriver resultatet ned i a-plassen i hovedhukommelsen. Det som skjer, er imidlertid langt mer komplisert.

1.2 LITT OM MASKINENS KONSTRUKSJON, CACHE OG MULTIKJERNE

Før 1980 var datamaskiner relativt enkle slik det framstilles i fig 1.1. Man hadde en beregningsenhet kalt CPU (Central Prosessing Unit). Den utførte én instruksjon av gangen i den rekkefølge de var spesifisert i programmet og leste, og skrev sine data direkte i hovedhukommelsen.



Figur 1.1 *Skisse av en datamaskin i ca. 1980 hvor det bare var én beregningsenhet, en CPU, som leste sine instruksjoner og både skrev og leste data (variable) direkte i hovedhukommelsen.*

Fra 1980-tallet begynte imidlertid CPUene å gå mye raskere enn hovedhukommelsen. Ordbruken skiftet også, slik at nå snakker vi om en prosessor istedenfor en CPU. Dagens avanserte prosessorer bruker om lag 150-200 ganger så lang tid å skrive til, eller lese fra, hovedhukommelsen som den tid det tar å utføre en enkel instruksjon (som å legge sammen to heltall). Her ville det ha blitt mye dødtid.

Svaret var å legge først én, senere flere mellomhukommelser (cache-hukommelser) mellom prosessoren og hovedlageret., Nå er det vanlig med tre cache-hukommelser som er laget av hurtigere, men dyrere elektronikk enn hovedhukommelsen. Når prosessoren 'tror' at den lagrer verdien av en variabel i hovedhukommelsen, lagres det først bare i nivå1-cachen (kalt L1, se fig 1.2), og prosessoren kan fortsette. Så snart som mulig lagres så disse data deretter i nivå2-cachen (L2), så i nivå3-cachen(L3) – og til sist i hovedhukommelsen omlag 200 klokkeenheter senere (1 klokkeenhet= ca. ½ milliarddels sekund = ½ nano-sekund). Tilsvarende gjelder for lesning. Hvis prosessoren ikke finner de opplysningene den vil ha i noen av cachene, må det leses fra hovedhukommelsen og inn i alle cachene før prosessoren får adgang til data.

Flere kjerner. En annen viktig utvikling av prosessorene var at man etter ca. år 2005 ikke greier å få én prosessor til å gå fortere. Prøver man med en vesentlig raskere klokke, vil prosessoren rett og slett først feile og så evt. smelte. Imidlertid greier man stadig å lage hver prosessor mindre og mindre ved at de transistorene den består av, blir laget mindre. Hva skulle så databrikke-designere som Arm, Intel og AMD gjøre? De la flere prosessorer, heretter kalt prosessorkjerner eller bare kjerner på hver brikke. Vi fikk da maskiner med to prosessor-kjerner (dual-core), så med fire kjerner, osv. Det er uklart hvor dette ender, men det er i alle fall laget forsøksproduksjon av brikker med ca. 100 slike prosessorkjerner, og det er helt sikkert at utviklingen stopper ikke med det. I tillegg finnes maskiner hvor man har satt 2 eller 4 slike multi-kjerne prosessorer på samme hovedhukommelse. Dette er ikke uvanlig eller spesielt dyrt. I tillegg kan noen av disse kjernene kjøre 2 tråder hver i parallell; såkalt hyperthreading, fordi en del av elektronikken er duplisert i hver kjerne. Eksemplene i dette kapittelet er eksemplene fra 2017 testet på to slike maskiner, en med 8 tråder (=1 prosessor med 4 kjerner som hver har hyperthreading) og en med 64 kjerner (=4 prosessorer med 8 kjerner som hver har hyperthreading), mens 2022 resultatene er testet på en nyere maskin også med 8-tråder (4 kjerner).

Det er også vanlig slik at hver kjerne har sin egen L1 og L2 cache, men deler som oftest L3 cachen med alle kjernene på samme brikke, men ikke med de andre prosessorene som evt. er i maskinen. Alle trådene i vårt program deler samme område i hovedhukommelsen. En viktig konklusjon på dette er at selv om en tråd som går på en av kjernene og har skrevet ned verdien av en variabel som alle trådene har utsyn til, så kan det ta lang tid før de andre trådene greier å se denne nye verdien fordi den f.eks. holder på å bli skrevet ned via alle cachene og det kan ta flere hundre klokkesykler før den oppdaterte verdien er i hovedlageret. Det er også ca. 10 til 30 registre per kjerne. Disse kan brukes til å holde de mest nyttede variablene i en beregning, slik som indeksen 'i' i en forløkke eller de mest sentrale variablene i en metode, som heltallet **`s`** i metoden **`sum:`**

```
long sum (int [] arr) {
   long s = 0;
   for (int i = 0; i < arr.length; i++) {
         s = s + arr[i];
    }
    return s;
}// end sum
```

Program 1.1 *Programeksempel, summering av verdiene i en array.*

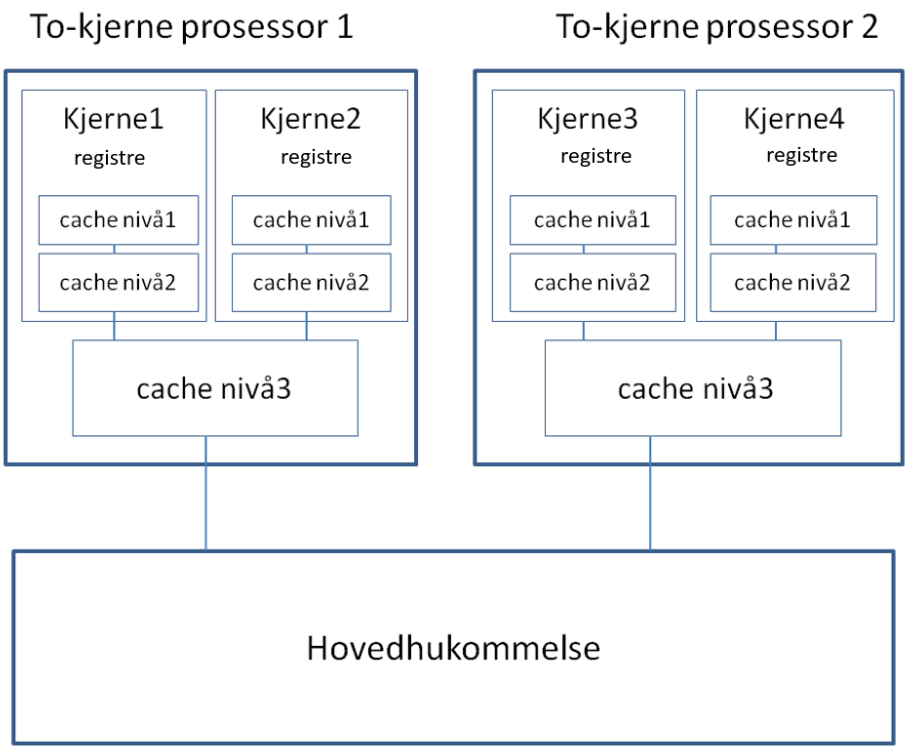
Det er Java-kompilatoren **javac** og senere kjøresystemet **java** som bestemmer hvor de ulike variablene i et program plasseres. Det er slik at alle variablene har sin plass i utgangspunktet i hovedhukommelsen. Men i koden Program 1.1 ser vi at metoden sum inneholder to variable, **s** og **i ,** som ikke er synlige utenfor metoden og som ikke kan leses av resten av programmet etter at metoden er utført. Disse to variablene vil derfor i en optimalisering av koden bli lagt i hvert sitt register og ikke i hovedlageret da heller ikke i cachene.

Parallelle instruksjoner I de siste årene har det kommet en rekke maskin-instruksjoner, som sammen med en rekke nye registre i kjernen, kan de f.eks. både med en og samme instruksjon multiplisere sammen to og to av disse registrene og summere resultatene fra disse 10-20 sett av registre. Dette er også en optimalisering av Java-koden kan nytte seg av.

```
int tall =…
void oekTall(int ant){
      for (int i = 0; i < ant; i++) {
            tall++;
       }
  }// end oektall
```

Program 1.2 *Metode oekTall som øker variabelen tall med 1 ant ganger.*

I det programeksempelet vi har i 1.2, vil vi illustrere flere viktige poeng i parallellprogrammering. Først må vi vite at operasjonen *tall*++ ikke blir utført som én operasjon, men egentlig er tre operasjoner: Først les verdien av tall, så legg **1** til denne verdien, og til sist skrives den nye verdien ned i variabelen *tall*. Som vi vet betyr dette at både gamle og nye verdien går via L1 cachen og veien ned til hovedhukommelsen er lang. Vitsen med disse cachene er at selve beregningsenheten bare forholder seg til sin L1 cache, leser og skriver i den, mens resten av systemet stadig forsøker å holde de andre cachene og hovedhukommelsen oppdatert. Beregningsenheten greier altså å gå så fort som L1 cachen greier å lese og skrive data, nesten hundre ganger raskere enn hovedhukommelsen hvis den ikke må vente på data fra en av de langsommere cachene eller hovedhukommelsen.



Figur 1.2. *To dobbeltkjerne prosessorer i en maskin med ulike hukommelser. Når det går parallelle tråder på hver av disse kjernene, ser vi at de som oftest ser ulik verdi på en felles variabel (eks. int i) i hovedlageret, hvis noen av trådene leser på en slik variabel samtidig som en annen tråd endrer dens verdi ved skriving. Dette fordi de ulike cashene ikke hele tiden er fullt oppdatert på siste verdi som er skrevet. Samtidig lesing og skriving på en variabel av mer enn én tråd må derfor alltid synkroniseres.*

Man kan jo til slutt spørre hvorfor man har alle disse lagene med hukommelse. Siden man greier å lage rask L1 cache, hvorfor kunne ikke all hukommelse vært slik? Poenget er at slike cacher er laget på en mye dyrere måte enn hovedhukommelsen, hver byte i cachene tar langt større plass og krever mer strøm. Det er også slik at måten disse cachene adresseres på en måte som forutsetter at de er relativt 'små'. Det ville kort sagt ikke lønne seg eller mulig å lage all hukommelse slik.

Typiske størrelser og forsinkelser måles i hvor mange tikk (cycle) som klokka i prosessoren må gjøre før en operasjon er ferdig. En prosessor som er på 2GHz, har da en klokke som gjør 2 milliarder slike tikk per sekund. Forsinkelsene i de ulike delene av hukommelses-systemet kan da være:

Registre i kjernen	størrelse = 0,1-1 Kb	tidsforsinkelse = 1 cycle
Cache nivå 1	størrelse = 32Kb	tidsforsinkelse = 2 cycler
Cache nivå 2	størrelse = 512Kb	tidsforsinkelse = 7 cycler
Cache nivå 3	størrelse = 4096Kb	tidsforsinkelse = 16 cycler
Hovedhukommelse størrelse = 8 – 16 GB		tidsforsinkelse = 190 cycler

Ekstra L3 cache. En nyhet i 2022 var at en produsent (AMD) ha laget sin siste prosessor for spill og tekniske beregninger så tynn at det er mulig å legge en ekstra L3 hukommelse rett oppå brikken slik

at den faktisk da får ca. 90 Mbyte totalt med L3 cache. Vitsen med å gjøre den tynn, er at det fortsatt er mulig å kjøle brikken selv med den ekstra L3 hukommelsen lagt oppå den.

Cache-linje størrelsen Når cache-systemet leser data fra hovedhukommelsen opp til nivå 3, så nivå 2 og til sist nivå1 cache, er det ikke én byte av gangen, men 64 byte hver gang (fordi det tar så lang tid, er det fordelaktig å laste opp mer enn det det er bedt om). Hvis programmet før eller siden trenger disse bytene, ser vi at programmet vil gå mye raskere enn om man skulle gå helt ned i hovedlageret hver gang programmet trengte neste heltall, flyttall eller neste byte.

Prefetch mekanismen Hvis programmet leser eller skriver tilfeldige adresser i en stor array med én indeks (større enn L3 cachen): a[i] , a[k], a[j], a[m],. .. så vil dette gå vesentlig langsommere enn sekvensiell aksess: a[i], a[i+1], a[i+2],... Spesielt oppstår denne effekten når vi leser bortover ***radene*** i en i en litt større to-dimensjonal array (så stor at det ikke er plass til hele arrayen i L2 eller L3 cachen), og programmet leser f.eks. a[i,j], a[i,j+1], a[i,j+2] vil en mekanisme i prosessorkjernen starte å lese neste cache-linje på 64 byte ***før*** programmet har bedt om det selv. Dette gjør at programmet går vesentlig raskere – mindre venting på data fra hovedhukommelsen. Denne mekanismen virker like bra hvis vi leser radene i en array baklengs fra slutten mot begynnelsen.
Merk at dette gjelder ***radvis*** lesing av en array. Leses derimot arrayen **kolonne**-vis (a[i, j] a[i+1,j] , a[i+2,j], .. vil elektronikken måtte lese en ny cache-linje for hver aksess, og vi vil få ingen hjelp av at vi allerede har lest 64 byter i cache-linjen som inneholder a[i, j] og prefetch-mekanismen gjenkjenner ikke dette som sekvensiell lesing. Lesing av neste kolonne-element vil derfor medføre at hver gang foretas en ny lesing 'helt' fra hovedlageret med en forsinkelse på ca. 100 nano-sekunder mot lesning av L1 cachen på 2 nanosekunder for radvis lesing av hvert element. En 'bom' på hva vi skal lese neste gang kalles en cache-miss.

Pipeline: Ikke alle maskininstruksjoner tar like lang tid, og de er da delt opp i flere mikroinstruksjoner som samlet sett løser oppgaven som den større instruksjonen skal løse – f.eks. divisjon. En kjerne vil da ved neste klokke-tikk prøve å starte første mikroinstruksjon i neste instruksjon, så igjen ved neste tikk neste instruksjon, ... osv. Kjernen kan holde da på med å beregne typisk inntil 5 maskin-instruksjoner samtidig på forskjellige grader av fullføring. Dette kalles en 'pipeline'.

Trådbytte: er når en tråd som utføres på en av kjernene avbrytes, og en annen tråd overtar kjernen og kjører sitt program på denne kjernen. Den tråden som holdt på med sine beregninger må stoppes og innehold av de registrene som denne avbrutte tråden hadde til sine beregninger må lagres i hukommelsen. Så kan register-innholdet til den nye tråden lastes opp fra der den andre tråden hadde lagt sitt registerinnhold. Først når registerinnholdet fra den nye tråden er i registrene, dvs. hvor den var i programmet og hva den foreløpig har regnet ut, kan denne nye tråden startes. Når vi har langt flere tråder enn vi har kjerner i en prosessor, og det har vi alltid når vi teller med de langt over 1000 tråder vi har i operativsystemet, så må de ulike trådene bytte om på å kjøre på kjernene. Selv om operativsystemets tråder i sum ikke bruker mer enn ca. 10% av prosessoren klokkesykler fordi de i all hovedsak venter på I/O eller andre begivenheter, så tar de noe tid. Også, hvis ingen tråd ønsker å kjøre på en prosessor, er det et enkelt lite program som kjøres: *idle-løkken*- det enkleste program vi kan tenke oss som bare går rundt i en evig løkke, og som med en gang vil foreta trådbytte med en tråd som har noe å gjøre. Når idle-løkken kjøres vil også prosessoren søke å redusere klokkehastigheten slik at den sparer strøm. Den PC-en som noen av eksemplene i dette kompendiet er kjørt på kan variere klokkehastigheten mellom 1.2 GHz og opp til 4.2GHz. *Kort sagt*: prosessoren med sine kjerner stopper aldri, og kjernene bytter ofte om på å kjøre de ulike trådene.

Spekulative beregninger Når koden som utføres inneholder en test (i f.eks. *if- while* eller *for-løkke*) inneholder kjernen ekstra registre og elektronikk slik at den starter med å regne *på begge grenene* av utfallet av testen – både om testen kommer til å gi sann eller falsk. Når testen er ferdig beregnet, vil

kjernen beholde beregningene som fulgte etter den beregningen med det riktige sanne svaret på testen og fortsette der. Beregningene som fulgte etter at testen fikk det gale svaret vil bli strøket.

IKKE så viktig her: Program-cache og **cache for virtuell-tabellene mm.** I tillegg til cache-system for data slikt det er beskrevet ovenfor er det også en tilsvarende rekke av cacher for den optimaliserte koden. Også den virtuelle adresseringsmekanismen har cache-områder for sine tabeller, dvs at alle programmer får byttet ut de øvre delen av adressefeltet i instruksjonene med den «virkelige» adressen til hvor data ligger i hovedhukommelsen. Dette er et system som gjør det lettere å skrive programmer ved at *de alle kan skrives som om at data ligger sammenhengende* i hovedhukommelsen. Tidligere var det slik at når lageret ble fullt, ble større deler av data midlertidig ble skrevet til disk – et roterende platelager (nå til SSA- 'disken'). I våre dager har man funnet ut at visse dataområder helles kan komprimeres med ZIP-algoritmen i selve hurtighukommelsen, slik at mer plass blir gitt programmer med høyere prioritet (disse ZIP-komprimerte områdene kan senere de-komprimeres når det blir plass til dem og hvis et program etterspør disse dataene).
Alle mekanismer i dette avsnittet bør man vite om, men de er klart mest viktige for de som skal skrive operativsystemer eller kompilatorer o.l., Det er mindre viktige hvis man som programmerer av brukerprogrammer for å påvirke kjøre-hastigheten. Hyggelig er det jo at noe av kompleksiteten i elektronikken er allerede tatt hånd av andre når vi skal skrive våre algoritmer og lage større datasystemer.

Hvorfor er hukommelsen så 'langsom'. Når man kjøper en datamaskin, kan man se som om hovedhukommelsen har om lag samme klokkehastighet, f.eks., 2600 MHz som CPU-en, men hvorfor tar det da så lang tid å lese og skrive i den? En lesning av en cache-linje på 64 byte sendes over data-kanalen som en ordre med startadressen til hukommelsen over en linje som enten kan sende 64 bit i parallell, eller mye raskere serielt 1 bit av gangen. Deretter skal data leses i hovedhukommelsen. Den er meget billig, men også meget kompakt laget. Å lese data der tar ca. 15-17 klokkesykler før de er klare for å sendes ut på datakanalen til CPU-en. Siden vi ber om 8 slike forsendelser – en cache-linje, skulle det ta ca. 8*16 cycler – dvs ca. 125 cycler bare for lesingen i tillegg til overføringen. Det korte svaret er da at hovedhukommelsen er meget rimelig og kompakt organisert, men at det går ut over hastigheten.

1.3. HVORFOR LAGE PARALLELLE PROGRAMMER MED TRÅDER?

Det er i hovedsak tre grunner til at vi kan ønske å ha parallellitet i et program:

1. Man skiller ut visse aktiviteter som går *langsommere* i en egen tråd, som tegning av grafikk på skjermen eller søk i en database. Resten av programmet kan da fortsette uten opphold.
2. Logikken i programmet er slik at det naturlig består av en rekke uavhengige aktiviteter som bare sjelden trenger å bruke felles data. Hvis hver slik aktivitet programmeres med hver sin tråd blir programmet faktisk *lettere* å skrive. Et godt eksempel er at du lager et system for direkte salg av flybilletter (eller innlevering av oppgaver i et kurs i programmering). Siden mange samtidig skal kunne gjøre dette, skriver du programmet slik at en tråd snakker bare med én kunde, og betjener bare den. Det er relativt lett. Dersom flere 'kunder' melder seg, starter vi bare en ny slik tråd for hver ny kunde. Vi ser at av og til må disse ha adgang til felles data, som for eksempel de ledige plassene på en bestemt flyavgang, og da må vi synkronisere trådene og sørge for at bare én får tilgang til å endre felles data av gangen. Feil som kan oppstå da må vi håndtere, men jevnt over kan disse trådene operere i full parallell.
3. Vi ønsker i dette kurset å bruke parallelliteten til å få visse beregninger til å gå *raskere*! Eksempler kan være store ingeniørberegninger, generering av bilder i spillgrafikk eller som det eksempelet vi til sist skal se på, sortering av større datamengder.

Vi skal i det etterfølgende basere oss på oppgaver av type 3, at vi ønsker et raskere program, men mesteparten av det vi skriver kan også brukes direkte i de to andre tilfellene.

2. JAVA.

Java er et språk som er under sterk utvikling. Oracle, som nå eier Java, lager hvert år nye versjoner. Det som i hovedsak kommer til er nye begreper og konstruksjoner, men det hender også at 'gamle' (*depricated)* metoder og begreper fjernes. Ett eksempel på et nytt begrep er records, som er objekter bare med data og ikke metoder. Prinsippet er at noen versjoner (hver tredje til femte) som lages er stabile (Java 5, Java 8, Java 11, Java 17 og Java 21, …). Disse versjonen vil bli vedlikeholdt (feilrettet og tilpasset stadig nye versjoner av operativsystemer) i mange år fremover mens versjonene mellom disse er 'forsøksversjoner' hvor nye konstruksjoner kan komme og gå og bli endret eller fjernet til neste versjon. Powerpointfoilene i kurset er hovedsak laget med Java 5 og Java 8, mens noen av hastighetsbetraktningene i dette kompendiet er laget med Java14 og Java 21(komplett liste over java - versjoner på: [redacted] . For å få bedre forklaringer på det nye som kommer i Java anbefales å abonnere på 'Java-magazine' (gratis) : [redacted]

2.1 PARALLELLE PROGRAMMER I JAVA

Vi kan altså få feil i programmene våre hvis mer enn én tråd samtidig skriver på en variabel som er felles. Vi ser at alle andre trådene som ønsker å skrive samme stedet da må stoppes og vente mens den første tråden blir ferdig med å skrive. Dette kaller vi å synkronisere trådene. I tillegg sørger det for at alle tradene ser samme verdier i alle felles datastrukturer.

Parallell programmering kan være vanskelig, og det er derfor utviklet flere synkroniseringsmåter og biblioteker for mer strukturert parallell programmering i Java. Vi skal senere i dette kapittelet gjennomgå bruk av tre nyttige synkroniseringsmetodikker; Bruk av barriere-synkronisering (for synkronisering mellom trådene våre), synchronized methods (synkronisering mellom metodene i *ett* objekt) og ReentrantLock (for synkronisering av trådenes adgang til *én* bestemt metode). Vi starter med en gjennomgang om både hvordan vi kan programmere mer parallelt og særlig hvorfor det så komplekst.. Etter denne gjennomgangen kan vi lett få inntrykk av at det er umulig å få parallelle programmer riktige. Det er ikke tilfellet, men for å få til det må man følge noen klare og enkle regler basert på bruk av de tre nevnte synkroniseringsmetodikkene nevnt ovenfor. Bryter man bare én av disse, vil det kunne gå veldig galt.

2.2 JAVA OPTIMALISERING, DEL 1

I det overstående er viktige mekanismer i elektronikken som kan øke hastigheten på programmet vårt beskrevet, men dette er mekanismer som i hovedsak opererer på maskinkodenivå, og vi skriver jo programmene våre i Java. Det er også mye vi kan oppnå med hvordan vi skriver Java-koden:

1. Data som vi til enhver tid bruker bør være minst mulige slik at de om mulig passer inn i L1 eller L2 cachene.
2. Uansett, prøv å lese og skriv data mest mulig sekvensielt for å utnytte prefetch mekanismen, og spesielt må vi i to-dimensjonale matriser lese/skrive data langs med radene, aldri langs kolonnene (i Java og nesten alle andre programmeringsspråk). Fortran er derimot annerledes, og lagrer data i to-dimensjonale data kolonnevis, og da må vi i Fortran lese/skrive disse matrisene kolonnevis).
3. Lag helst mange små metoder som hver løser ett enkelt problem (som f.eks. summen av elementene i en array, eller som finner neste primtall i en primtalls-array). Den mekanismen vi beskriver nedenfor som optimaliserer koden, kan ikke like lett optimalisere lange metoder med mange løkker enn et program med som mange små metoder som kaller hverandre

4. Vi har i tidligere kurs lært at java-kompilatoren **javac** oversetter vårt Java program til en enkel og kompakt kode, bytekode, som kan sees på som instruksjonene til en byte-maskin. Denne utfører så disse byte-instruksjonene i samme rekkefølge sekvensielt; noe tilsvarende som Python utføres. I den første varianten av Java, Java1 var det det som skjedde – man ga bytekoden til java som så leste de ulike bytekodene og så utførte dem en-etter-en.

   Det som nå, fra og med Java 3 skjer, er følgende:

   Første gang et objekt fra en klasse opprettes og en metode kalles i koden, blir den først oversatt til maskinkode på den maskinen man kjører på (just-in time kompilering). Og det er denne maskinkoden som utfører programmet vårt. Det er klart raskere enn å tolke byte-koden som instruksjoner til en tenkt byte-maskin. Merk at det bare er de delene som utføres som blir oversatt til maskinkode – resten forblir i byte-kode. Utføres en metode flere ganger (si 10-100 ganger), så optimaliseres den oversatte maskinkoden, Instruksjoner kan bli byttet om og forenklet, men den 'optimaliserte' koden gir samme svar som en ikke-optimalisert kode. Denne koden er nå mye raskere enn den ikke-optimaliserte maskinkoden

   Kalles en metode enda flere ganger, f.eks. over 100 000 ganger, blir koden for denne metoden ytterligere optimalisert (nå for 3dje gang) og kan igjen gå enda raskere. Disse optimaliseringene som gjør at noen operasjoner i Java kan gå fra 4 - 100 000 ganger fortere, er beskrevet i tabellen nedenfor hvor ulike java-elementer og to sorteringsalgoritmer testes. Utføres en slik optimalisert metode mange ganger – si mer enn 200 000 ganger som blir koden ytterligere optimalisert, og går da enda mye raskere

Tider i usek per iterasjon som funksjon av n, antall iterasjoner											
n	1	2	3	10	100	1000	10000	100000	1000000	X bedre (from n=1)	X bedre (from n=2)
for-loop, len = 100	0,4	0,2	0,1	0,022	0,020	0,015	0,002	0,002	0,00000	**181**	**90**
metodekall	4,5	0,9	0,3	0,098	0,087	0,063	0,005	0,005	0,00043	**10465**	**180**
int[] new , , len = 100	1,1	0,3	0,2	0,117	0,108	0,275	0,160	0,160	0,09815	**11**	**2**
array kopi for-løkke , len = 100	2,4	1,7	3,2	1,980	1,880	0,889	0,018	0,237	0,01217	**197**	**7**
System.arraycopy , len = 100	4,4	0,6	0,3	0,124	0,120	0,043	0,025	0,022	0,02008	**219**	**27**
new Thread,start&join	1164	362,9	224,0	197,424	174,278	172,948	175,820			**7**	**2**
new Class C m.metodekall	791,4	15,3	1,8	0,268	0,220	0,045	0,002	0,004	0,00274	**288832**	**3438**
int [] a skriv , len = 100	0,8	0,7	0,1	0,036	0,035	0,027	0,023	0,008	0,00659	**121**	**93**
int [] les ,, len = 100	0,3	0,3	0,0	0,028	0,026	0,022	0,015	0,006	0,00638	**47**	**54**
double to long	3,7	1,2	0,3	0,218	0,164	0,066	0,042	0,000	0,00004	**92500**	**60000**
insertSort double[] , len = 100	93,1	164,0	150,1	55,343	7,266	1,754	1,634	1,618	1,61902	**58**	**101**
Arrays.sort double[] , len = 100	404,4	73,1	60,9	56,506	6,028	1,357	1,165	1,116	1,11126	**364**	**65**

Figur 2.1 *Kjøretider for ulike* ***sekvensielle programmer*** *i* ***2022*** *som funksjon av antall ganger eksekvert i µsekunder (million dels sekunder) for ulike Java-elementer og for to sorteringsprogrammer: Ett brukerskrevet med kode i testprogrammet (innstikk-sortering av flyt-tall) og et er fra Java-biblioteket (Kvikk-sortering av flyt-tall). De to siste kolonnene viser speedup= hvor mange ganger fortere en eksekvering er etter 1 million kjøringer, regnet ut fra tidene for første kjøring (n=1) eller andre kjøring (n=2). Testene er kjørt på en AMD Ryzen 5 3500U PC med Java versjon 14.01 i Windows 10.*

Kommentarer til tabellen:

1. Vi ser at optimaliseringen er meget sterk for nesten alle konstruksjoner, men særlig for metode-kall og det å lage et objekt av en klasse (samt konvertering av flyt-tall (double) til heltall). Disse effektiviseres vesentlig. Dette er viktig for vår programmering: å dele opp vårt program i klasser med mange mindre metoder, koster lite eller ingen tid når programmet har kjørt 3-10 ganger eller mer.
2. Vi ser at de to kolonnene til høyre regner ut Speedup (SU) henholdsvis fra tiden det tar å kjøre første gang og andre gang. Grunnen til også å regne ut SU fra andre kjøring er at da er all

tidsbruk som kompilering til maskinkode og henting av klassen evt. fra Java-biblioteket unnagjort. Vi ser dette spesielt i eksempelet Arrays.sort som første gang tar 404 µs mot innstikksort med 93 µs som har koden liggende i testprogrammet. Derimot er optimalisering av maskinkoden der den senere foretas, inkludert i kjøretidene (som f.eks. Inntikksort andre og tredje gang).

3. Det er meget tilfredsstillende at en egen skrevet brukerkode som Innstikksort kan optimaliseres opp med en faktor 60-100.
4. Av det overstående kan det se ut som f.eks. Innstikksort blir effektivisert for all mulig bruk i programmet. ***Det er galt***. Optimalisering av metodene består bl.a. i at hele koden for Innstikksort (og alle andre metoder) blir stappet inn koden der den ***kalles fra*** – og det erstatter kallet med selve koden. Det betyr at hvis vi bruker og kaller Innstikksortering fra et annet sted i programkoden må den gå igjennom samme optimalisering. Dette gjelder også optimalisering av å lage et objekt av en klasse (**new).**

2.3 Prosesser og tråder

Et program (en prosess) i Java består nå av en eller flere tråder. Hver av trådene er ett sekvensielt program som deler prosessens dataområde. I tillegg til at de kan ha hver sin private del av hukommelsen, og koden i trådene utføres ovenfra og nedover slik vi tidligere har lært at programmer oppfører seg. Har vi flere tråder i programmet vårt har vi et parallelt program.

1. Når man starter java-programmet, får vi én tråd: maintråden. Den tråden starter med å utføre metoden main(). Fra main-tråden kan så programmet vårt starte flere andre tråder som vi har skrevet kode for.
2. For å få disse andre trådene må de programmeres som en egen klasse som er en subklasse av klassen Thread (eller de kan være en klasse som implementerer grensesnittet Runable). Alle de objektene vi så lager av denne subklassen vil da inneholde en egen tråd som vil utføres i parallell med de andre trådene i vårt program.
3. Denne tråd-klassen, her kalt Para, legges helst inni den klassen som inneholder main() – metoden, og den skal selv inneholde en metode public void run() { .. } som dere må skrive selv. Denne metoden run() tilsvarer på mange måter main()-metoden i hovedprogrammet, og er den metoden som kalles av systemet når den nye tråden er laget og startet:

```
Thread[] t = new Thread [antTraader ];
for (int i = 0; i< c; i++) {
    t[i] = new Thread(new Para(i));
    t[i].start();
}
```

4. For senere i programmet å vente på alle disse **`antTraader`** stk. trådene blir ferdige, kan man utføre flg. setninger:

```
for (int i = 0; i< antTraader; i++){
    try{t[i].join();} catch (Exception e) {};
}
```

Kurset inneholder flere andre måter, f.eks. vente på en egen CyclicBarrier i maintråden på at alle de trådene man har startet er ferdige.

5. Legg merke til at det nye trådobjektet er en parameter til klassen Thread.

6. Man kaller ikke `run()` direkte, men med å kalle metoden `start()` i dette nye trådobjektet, som gjør mye bak kulissene for å lage denne nye tråden og til sist kalles `run()` i trådobjektet.
7. En tråd avsluttes når den har utført siste setning i sin main() eller run() – metode.
8. Ingen tråder må drepe/avslutte en annen tråd, men ofte vil en tråd legge seg å vente på at en bestemt operasjon (utført av andre tråder) er ferdig.
9. Ikke før alle trådene, også main-tråden i et program er avsluttet, er programmet ferdig.
10. Alle disse trådene deler samme adresserommet i hovedlageret – de ser de samme variablene (data), classer og metoder som er deklarert der ut fra sitt skop (sitt utsyn til deklarasjoner), men kan se litt gamle eller nyere verdier på ulike data de har utsyn til (løsning på dette problemet kommer senere i dette kapittelet).
11. Trådene utføres også samtidig og på hver sin kjerne i CPU-en hvis vi har en kjerne for hver tråd. Er det flere tråder enn kjerner, vil operativsystemet prøve å la trådene dele på bruken av kjernene. Dette ordner operativsystemet (Windows, Linux eller MacOS).
12. Operativsystemet har i tillegg en rekke tråder for å ulike oppgaver (administrasjon, I/O, nettet, vinduer på skjermen,…,osv.). De denne setningen ble skrevet hadde Win10 operativsystemet 3789 tråder som var aktive, men de aller fleste av disse trådene lå og ventet. I sum tok de omlag 10% av maskinens kapasitet. Disse systemtrådene står i motsetning til de trådene vi skal skrive i IN3030 som hver vil forsøke, grovt sett, å bruke *hele kapasiteten* til en kjerne.

Med flere tråder har vi da et parallelt program-system hvor flere sekvensielle programmer kjører samtidig. De fleste problemene med slike parallelle systemer er når to eller flere tråder vil skrive på samme variabel. Vi må da synkronisere trådene (mer om det senere), eller hver tråd må da lokalt ha en kopi av slike data og skrive/lese på disse. Senere samstilles data fra disse lokale kopiene.

2.4 NYE KLASSER, SØPPELTØMMING MM

For å lete kodeskrivingen er det i Java 17 innført begrepet 'record' som gjør det lettere å behandle klasser som bare inneholder data (eks: Punkt med en x- og en y-verdi), Videre er det rent statiske classer som inneholder data som ikke skal endres etter at de er laget og metodenavn som parametere. Viktigst er kanskje at det er definert tekstblokker som er et antall linjer med tekst. Dette er syntaks som gjør det enklere å skrive kode, men siden det implementeres som 'vanlige' klasser er det ikke begreper eller tillegg til Java som gjør at parallelle programmer går fortere .
Det som gjør at Java 17 programmer nok går fortere enn Java 8 programmer er at søppeltømmingsalgoritmer i Java 17(dvs av de objektene som er laget under kjøring og som nå ikke lenger kan brukes fordi ingen av trådene har en peker til dem) nå er byttet helt ut med den 'gamle' algoritmen med en ny som er mer inkrementell, Den tar ikke alt søppel med en gang, men gradvis fjerner alle 'søppel'- objekter.

Et annet prosjekt (Valhalla) prøver å lage en mer felles inn-utpakking av enkle variable , slik en 'int' blir pakket inn som en Integer med et objekt rundt seg, eksempelvis i ArrayList <Integer>, noe som klart tar lenger tid og plass. Dette kompendiet peker på dette problemet i kap. 12, hvor en enkel int[] nå er klart er raskere enn en ArrayList<Integer>. Arbeidet med å få basale typer (som byte, int og double) og den motsvarende (Byte, Integer og Double) inn som felles begreper med felles metoder vil bli først komme i Java 22 i 2023 eller 2024 ([redacted]). Poenget med Valhalla er å endre hvordan generiske typer som Integer er definert i Java slik at man kan blande basale typer og generiske typer og at generiske typer kan både få mer effektivitet og ikke mer plasskrevende enn basale typer som 'int'.

3. OM BRUK AV SYNKRONISERINGSPRIMITIVER OG LÅSER

Vi har sett at det er store problemer når ulike tråder vil skrive nye verdier inn i samme variabel – ulike tråder kan på samme tidspunkt se ulike verdier av samme variabel. Det er faktisk også vanskelig å avgjøre for én tråd når en annen parallell tråd er ferdig.
Til å løse disse problemene har man innført flere typer av låser i Java. En lås er en mekanisme som

kan stoppe og la en eller flere tråd(er) vente. Tråden kaller på låsen og låsen gjør en test, og avgjør om kallende tråd må vente eller ikke. Felles for låsene i Java er følgende gode egenskap:

Når flere tråder gjør et kall på ***samme lås,*** vil alle disse trådene være sikret at all skriving trådene har gjort på felles variable ***før dette kallet*** er synlig for alle de andre etter kallet. De ser da samme verdier på felles variabler**.** Da blir bla. alle felles variabler for disse trådene som er endret skrevet ned i hovedhukommelsen fra cashene før trådene kan fortsette.
I Java er det mange titalls forskjellige låser som kan synkroniserer trådene. Vi skal i hovedsak i våre løsninger bare bruke tre av disse som er kortfattet beskrevet i neste avsnitt. Men for å illustrere visse problemer skal vi også i kurset nytte noen andre slike låser senere. Til alle slike låser/synkroniserings-mekanismer er det en rekke metoder (totalt 19 stk. for den 'enkleste': ReentrantLock) hvor en bruker kan spørre om hvor mange andre tråder som venter på denne låsen, spørsmål om å neste i køen osv.
Vi vil illustrere dette senere med programmer som benytter tre typer av låser: CyclicBarrier som sikrer at vi vet når aller trådene er ferdige og som vil stoppe tråder å la dem vente inntil alle trådene er ferdige med en bestemt del av koden. En nyttig, og den raskeste låsen er ReentrantLock for å beskytte en bestemt metode fra at flere tråder samtidig kan utføre den metoden. Bare en tråd slipper inn ad gangen og andre tråder som litt senere kaller denne metoden må vente til den første tråden er ferdig (en tråd av gangen). Til siste skal vi bruke Javas innebygde synchronized mekanisme som skal sikre at bare én tråd av gangen er inne i en av alle de metoder som er 'beskyttet av' synchronized-ordet i *ett* objekt. Hvis metodene er deklarert som static, vil denne synkroniseringen gjelde i alle objekter av denne klassen hvor denne beskyttelsen er brukt. Hvis metodene ikke er static, vil metode-beskyttelsen bare gjelde metodene i ett objekt av gangen. Synchronized (som er et reservert ord i Java) kan da beskytte en hel metode eller bare en blokk med setninger inne i en metode.

3.1 HVORDAN VIRKER SYNKRONISERING AV TRÅDER

Tråder er altså hver selvstendige sekvensielle programmer som kjører samtidig, vi sier i parallell, på én multikjerne PC. For å få adgang til CyclicBarrier, ReentrantLock som begge er klasser som man finner på Java-biblioteket ved å ha følgende import-setninger i toppen av programkoden :

```
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
```

Vi lager først ett objekt av en slik synkroniserings-klasse, og det er bare de trådene som kaller metodene i dette objektet som synkroniseres med hverandre (ved å kalle f.eks. metoden await()).

I tillegg skal vi bruke synchronized som er bygget inn i programmeringsspråket Java og som kan sørge for at bare én tråd av gangen kan utføre en slik metode av alle de metodene i dette objektet med synchronized ordet foran seg i deklarasjonen. Vi kan bare synkronisere tråder i forhold til hverandre som bruker samme synkroniserings-objekt og alle objekter kan synkroniseres.
Et enkelt eksempel er at vi ønsker å debugge et parallelt program med tråder, og at vi ønsker å skrive ut verdiene av en variabel i en slik tråd.
De er altså alle tre mekanisme som greier å stoppe andre tråder midlertidig når en tråd skal skrive på data, en fil eller skjerm som er felles for alle trådene.

3.2 OM BRUK AV REENTRANTLOCK

Den enkleste og raskeste låsen er ReentrantLock.

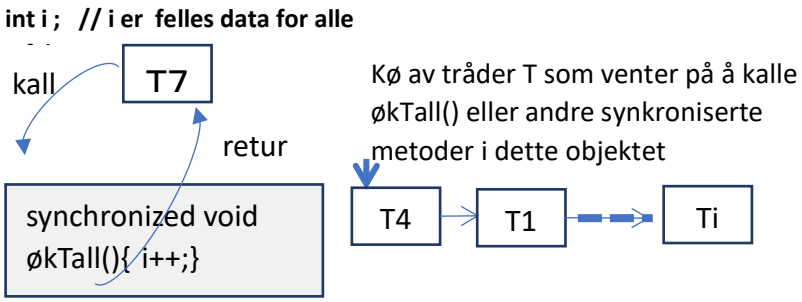
```
class X {
   private final ReentrantLock lock = new ReentrantLock();
   // ...

   public void m() {
     lock.lock();  // block until condition holds
     try {
        // ... method body
     } finally {
        lock.unlock()
     }
   }
 }
```

Program 3.1 *Kode for bruk av ReentrantLock som beskytter en metode m() fra å bli brukt av høyst en tråd.*

3.3 OM SYNKRONISERTE METODER

Synkroniserte metoder bruker låsen til det objektet de tilhører (alle objekter har én slik lås hver) og sørger for at hvis mer enn én tråd kaller den, vil bare én slippe til av gangen og de andre må vente i en kø (fig 3.1). Kallende tråd får gjort seg helt ferdig og resultatet blir skrevet ned i hovedhukommelsen før neste tråd får utføre sitt kall.



Figur 3.1 *Synkroniserte metoder lar én tråd slippe til av gangen og køer opp andre tråder som evt. samtidig ønsker å gjøre et kall på synkroniserte metoder i et objekt av denne klassen. Her holder tråd T7 på med et kall på økTall(), og mange andre tråder som har forsøkt å gjøre kall mens T7 holder på å få utført sitt kall, må da vente i en kø. Når T7 er ferdig, slipper en av de andre løs fra køen og kan gjøre sitt kall, osv.*

Merk at hvis det er flere synkroniserte metoder i samme objektet, vil denne låsen i objektet sperre også for samtidige kall fra andre tråder på disse andre metodene. I ett objekt kan altså høyst én synkroniserte metode bli eksekvert av gangen. Merk at hvis man har flere objekter av den klassen hvor de synkroniserte metodene er, er det mulig å få utført en synkronisert metode samtidig av to eller flere tråder. Det kan være en utilsiktet feil, hvis kallene kommer til ulike objekter av denne klassen.
Vi kan først prøve å kjøre programmet i 3.2. med alle de 4 ulike definisjonene av metoden økTall(). Først spør den systemet om antall kjerner i maskinen og skriver det ut. Deretter lager den et objekt av klassen Parallell og så leser den inn fra kommandolinja hvor mange tråder den skal starte om hvor mange ganger hver av dem skal øke heltallet tall med 1.Vi bruker her en long, 64 bit heltall for variabelen tall fordi summen av antall opptellinger (antGanger*antTråder) kan bli større enn største verdi for et 32 bit heltall. Vi kaller så metoden utfør() i dette objektet p av klassen Parallell.
Merk at: Hvert objekt som skapes med *new* har en lås. Det er den låsen som nyttes ved kall på alle de synkroniserte metoder i det samme objekt som metoden er i Merk at: Hvert objekt som skapes med *new* har en lås. Det er den låsen som nyttes ved kall på alle de synkroniserte metoder i det samme objekt som metoden er i.

```
import java.util.*;
import java.util.concurrent.*;

/** Start >java Parallell <ant tråder> <ant ganger i løkke> */
class Parallell{
      long  tall=0;      // Sum som 'antTråder' tråder teller opp
      CyclicBarrier b ;  //sikrer at alle er ferdige før vi tar tid og sum
      long antTråder, antGanger ; // Etter summering: riktig svar er
                                  // antTråder*antGanger
     void utskrift(double tid) {
        System.out.println(«Tid «+antGanger+» kall * «+
          antTråder+» Traader =»+Format. align(tid,9,6)+ « sek,\n sum:»+
          tall +», tap:»+ (antTraader*antGanger -tall)+» = «+
          Format.align( (antTraader*antGanger - tall)*
          100.0/(antTråder*antGanger),5,1)+»%»);
     } // end utskrift

    synchronized void økTall(){ tall++;}         // 1)
    //  void økTall() { tall++;}                 // 2)

    public static void main (String [] args) {
       int antKjerner = Runtime.getRuntime().availableProcessors();
       System.out.println("Maskinen har "+ antKjerner + " kjerner.");
       Parallell p =  new Parallell();
       p.antTråder = Integer.parseInt(args[0]);
       p.antGanger = Integer.parseInt(args[1]);
       p.utfør();
     } // end main

     void utfør () {
        b = new CyclicBarrier((int)antTråder+1);  //+1, også main venter
         long t = System.nanoTime();               // start klokke
         for (int i = 0; i< antTråder; i++)
               new Thread(new Para()).start();
         try{ // main tråden venter
              b.await();
         } catch (Exception e) {return;}
               double tid = (System.nanoTime()-t)/1000000000.0;
               utskrift(tid);
       } // utfør

     class Para implements Runnable{
              public void run() {
                      for (int i = 0; i< antGanger; i++) {
                           økTall();
                        }
                        try { // wait on all other threads + main
                            b.await();
                         } catch (Exception e) {return;}
              } // end run

             //    void økTall() { tall++;}               // 3)
             //    synchronized void økTall(){ tall++;}  // 4)
      } // end class Para
}// END class Parallell
```

Program 3.2. *Et program som viser både riktig og gal bruk av låser i Java. Vi ser 4 ulike plassering av en metode økTall() – kommentert med 1), 2) 3) og 4). Bare 1 er riktig. Inndata fra kommandolinja er hvor mange tråder vi vil starte og hvor mange ganger hver av disse trådene vil telle opp en felles variabel (long tall) i klassen Parallell. Hvis man fjerner kommentarmarkeringen // for én av kallene på økTall() vil programmet kunne kompileres og kjøre. Bare én av disse plasseringene er riktig. Alle de feilaktige plasseringene av metoden 'økTall()' vil som sluttresultat få alt for liten sum i tall. Kjører vi et feilaktig program flere ganger med samme parametre vil det også nesten alltid gi ulike svar; typisk for synkroniseringsfeil.*

Vi kan lett gjøre den feilen at vi skaper flere objekter, og dermed flere låser. En tråd som kaller en synkronisert metode vil låse med den låsen som er i det objektet som utfører metoden. Program 1.2 viser et eksempel på en slik feil. Hvis vi 'av-kommenterer' plassering 4) og nytter den ser vi at vi får mange feil når vi kjører programmet (med 1000 eller flere oppdateringer). Grunnen til dette er at vi har laget en lås for hvert av tråd-objektene av klassen Para. Nå vil de synkroniserte variablene definert på denne måten, låses bare de kallene som nytter samme lås. Men siden hver tråd har sitt objekt og sin lås, vil ingen av trådene greie å låse ute de andre trådene – fordi de bruker hver sin lås.

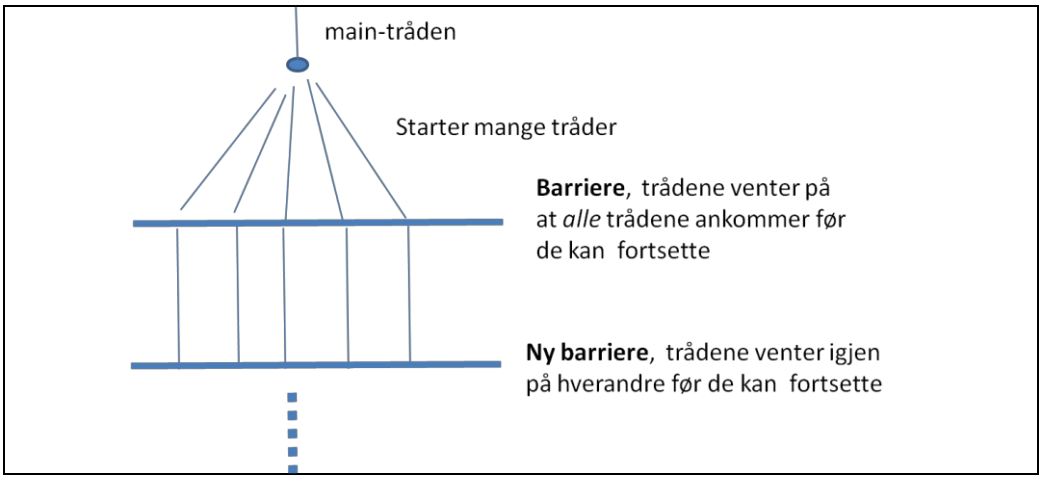
Tre kjøringer	**1**	**2**	**3**
Svar **uten** synchronized **3)**	7 112 531	5 911 630	6 169 492
Svar **med** synchronized **1)**	10 000 000	10 000 000	10 000 000

Tabell 3.*1* *Tre kjøringer hvor vi starter 100 tråder som hver forsøker å øke variablen i 100 000 ganger med 1, dvs. i skulle da bli = 10 mill. Vi ser at uten at økTall() er riktig synkronisert, får vi mange feil og ulikt svar i hver kjøring.*

Hvis vi derimot nytter plassering 1) for den synkroniserte metoden økTall(), ser vi at den er inne i ett objekt av klassen Parallell. Det er bare dannet ett objekt av denne klassen, og alle kall på økTall() nytter da samme lås, og alt går da bra uansett hvor mange tråder og kall på økTall()vi har. Plasseringene 2) og 3) går også galt fordi det ikke brukes noen låser, tilsvarende som plassering 4) med mange opptellinger som går tapt. Alt fra 0,001 % til over 90 % av summen mangler. Resultatene varierer også fra kjøring til kjøring med samme parametere. Grunnen til at vi ikke alltid får tapte oppdateringer hvis vi har få kall på økTall() per tråd, er at hver tråd vi da starter, greier å gjøre seg ferdig før neste tråd starter. Vi får da ikke et skikkelig parallelt program, men at trådene utføres etter hverandre, sekvensielt.

4 BARRIERE SYNKRONISERING

Ikke alle beregninger kan greit eller effektivt løses med synkroniserte metoder. Mange beregninger kan parallelliseres ved at man deler dem opp i flere trinn. Hvert trinn gjøres i parallell av et antall tråder, men *alle trådene må være ferdig* med ett trinn før beregningene i neste trinn kan begynne. Da kan alle trådene fortsette med del to av beregningen, og da vet de at de kan lese hva de andre trådene skrev i forrige trinn av beregningen. Kanskje er det mange slike trinn i beregningene. Et viktig spesialtilfelle er at vi ikke trenger å dele opp selve beregningen i flere trinn, men at vi vil at hovedtråden, dvs. den tråden som programmet starter med i main, skal vente til alle trådene den har startet er ferdige med beregningene. Først da kan hovedtråden presentere resultatet til brukeren.



Figur 4.1 *Programmet starter alltid først bare en tråd – maintråden. Den lager ett objekt b av klassen CyclicBarrier og et større antall tråder. Hvis barrieren er laget for **k** tråder, så vil alle tråder, også evt. maintråden, vente hvis de sier b.await() inntil **k** tråder har sagt b.await(). Da slipper alle de **k** trådene løs og kjører videre.*

For å få til en slik ventemekanisme for k stk. tråder, lager vi et objekt av klassen CyclicBarrier, og parameteren er antallet tråder som den skal køe opp; og når den siste melder seg skal alle trådene igjen slippes løs.

```
import java.util.concurrent.*;

CyclicBarrier b = new CyclicBarrier (antTråder);

< I trådene vil vi finne følgende kode når vi skal vente
i 'b' på at alle de andre trådene også er ferdige med
beregningene sine:>

     try {
            b.await();
     } catch (Exception e) {….}
```

Program 4.1. *Program som skisserer riktig bruk av CyclicBarrier i Java.*

Vi vil i neste programeksempel 4.2 se at vi har en parameter som er 1 større enn antall tråder vi lager, fordi vi bruker den sykliske barrieren *b* til at alle trådene og main-tråden venter på hverandre. Grunnen til at det heter CyclicBarrier er at når så mange tråder som den er spesifisert for har ventet og blitt sluppet fri, kan den uten videre motta det samme antallet tråder til ny runde med venting og frislipping uten ny initialisering (den er straks gjenbrukbar). Hvis trådene har flere trinn hvor de må

vente på hverandre, har vi gjerne to CyclicBarrier – én som settes opp med antTråder og som trådene bruker seg imellom, og én som initieres med antTråder + 1, som trådene venter på når helt ferdige og som main-tråden også har lagt seg til å vente på etter at den startet alle de andre trådene. I main-tråden vet vi da at når den slipper løs, har alle trådene blitt ferdige med koden sin.

Husk at her gjelder også det første punktet om synkronisering. Det å kalle på await() på en barriere sørger ikke bare for at alle venter, men også at alle etterpå kan se alt hva de andre skrev på felles variable før await()-kallet.

5.1 ET PROGRAM SOM BRUKER CYCLICBARRIER OG BEREGNER MAX-VERDIEN I EN ARRAY

```
import java.util.*;
import java.util.concurrent.*;

/** Start >java   FinnMax2 <ant tråder>  */
class FinnMax2{
      int[] a,lokalMax;  //finn max verdi i a[]
      CyclicBarrier b;   // sikrer at alle er ferdige før vi tar tid og
                        //  sum
      static int antTråder, ant;

      public static void main (String [] args) {
               antTråder = Integer.parseInt(args[0]);
               new FinnMax2().utfør();
       } // end main

      void  utfør () {
               a= new int[antTråder*antTråder]; // større problem
               ant= a.length/antTråder;   // antall elementer per tråd
               lokalMax = new int [antTråder];
               Random r = new Random(1337);
               for (int i =0; i< a.length;i++) {
                   a[i] = Math.max(r.nextInt(a.length)-i,0);
               }
               b = new CyclicBarrier((int)antTråder+1); //+1, også main
               int totalMax = -1;
               long t = System.nanoTime();       // start klokke
               for (int i = 0; i< antTråder; i++) {
                         new Thread(new Para(i)).start();
               }

               try{ // main venter på Barrieren b
                        b.await();
               } catch (Exception e) {return;}

              // finn den største max fra alle trådene
               for (int i=0;i < antTråder;i++)
                    if(lokalMax[i] > totalMax) totalMax = lokalMax[i];

               System.out.println("Max verdi parallell i a:"+totalMax +
                 ", paa: "+((double)(System.nanoTime()-t)/1000000.0)+
                  " millisek.");

             // sammenlign med sekvensiell utføring av finnMax
             t = System.nanoTime();
             totalMax = 0;
             for (int i=0;i < a.length;i++)
               if(a[i] > totalMax) totalMax = a[i];
```

```
            System.out.println("Max sekvensiel:"+totalMax +", paa: "+
               ((double)(System.nanoTime()-t)/1000000.0)+ " millisek.");
       } // utfør

      class Para implements Runnable{
            int ind, minMax = -1;
            Para(int i) { ind =i;} // konstruktor

            public void run() { // Det som kjøres i parallell:
                for (int i = 0; i< ant; i++) {
                      if (a[ant*ind+i] > minMax) minMax = a[ant*ind+i];
                }
                lokalMax[ind] =minMax;  // levér svar

                try {  // wait on all other threads + main
                     b.await();
                } catch (Exception e) {return;}
              } // end run
        } // end class Para
}// END class Parallell
```

Program 5.2. *Et program som viser riktig bruk av CyclicBarrier i Java. Parameter til programmet er antall tråder, og det lages et array a[] som er antTråder*antTråder lang med tilfeldig positivt innhold. Vi starter så antTråder i parallell som finner maksimalverdien i hver sin del av a[], og tråd nr i legger sitt svar inn i lokalMax[i]. Hovedprogrammet som venter på den sykliske barrieren kan, når alle trådene er ferdige, selv gå gjennom lokalMax[] og finne totalMax-verdien. Vi skriver ut denne og tidsforbruket. Som sjekk går vi så sekvensielt gjennom a[] og skriver ut den max-verdien vi da finner og tidsforbruket for sekvensiell gjennomgang til sammenligning.*

Vi ser at dette er et program som greit parallelliserer beregningen av max-verdien i en array, men vi ser også av tidene som programmet skriver ut, at den parallelle beregningen tar ca. 50-100 ganger så lang tid som bare å lese gjennom arrayen sekvensielt fra start til slutt for beregningen av maksverdien. Dette eksemplet lærer oss forhåpentligvis bruk av en syklisk barriere, men også at parallellisering av svært enkle oppgaver hvor vi bare ser på hvert dataelement én eneste gang i beregningene, er det ingen vits i å parallellisere. Den ekstra tiden det tar å starte og stoppe tråder tar da langt lenger tid enn selve beregningene. Vi skal nå se på et problem, sortering, hvor parallellisering vil lønner se; i alle fall hvis vi skal sortere mer enn 100 000 tall.

6 GENERELT OM PARALLELLISERING AV ALGORITMER, DEL 2

Her er en skisse av de stegene vi vanligvis foretar når vi lager en parallell algoritme for å løse ett problem.

1. Start med et vel testet sekvensielt program som løser problemet.
2. Del opp problemet i flere mindre deler som kan løses hver for seg.
   Vanligvis vil man søke å dele dataene i like store deler, ofte langt flere enn du har kjerner – f.eks. 16* antKjerner. Grunnen til dette er at en tråd som løser et slikt delproblem kan få ventesituasjoner, og da er det greit at en annen tråd er i stand til å eksekvere. Imidlertid må det advares mot alt for mange tråder. Det tar tross alt noen få millisekunder å skape og starte den første tråden; raskere for de neste trådene. Vi kan godt bruke 10-500 tråder for å løse et problem, men ikke mange 10-tusner.
   Husk også at hvis lengden av problemet ikke er delbart med tallet antKjerner , så får vi en liten rest som den siste tråden, den med høyest indeks, også må ta med i sitt område.
3. Start en tråd for hver av disse delene av problemet.
   Dette gjør vi hvis ikke oppdelingen og start av tråder er avhengig av hvilke, og hvor mye data vi har. Da skjer en kombinasjon av oppsplitting av problemet og start av tråder samtidig under eksekvering – pkt. 2 og 3. kombinert.
4. La hver tråd løse en slik del.
   Vanligvis vil enten den samme, eller en lett modifisert versjon den sekvensielle algoritmen nyttes for hver slik del. Ofte erstattes da rekursjon med tråder. Felles data som flere tråder skriver på samtidig, beskyttes med synkroniserte metoder.
5. Vent til alle trådene er ferdige – f.eks. med en syklisk barriere.
6. Kombiner til sist del-svarene til en løsning på hele problemet. Av og til er det ikke nødvendig, men ofte er det slik at det er noen avsluttende beregninger.
7. Når du skal debugge et slikt parallelt program og f.eks. ønsker å skrive ut verdier av en eller flere variable i de ulike trådene, virker det dårlig å nytte System.out.println(String s) fordi den ikke er synkronisert og du vil oppdage at utskriften fra de ulike trådene blander seg på skjermen i en ikke lesbar mølje. Dette løser du enkelt ved å lage flg. variant av println:

   synchronized void println (String s){System.out.println(s);}

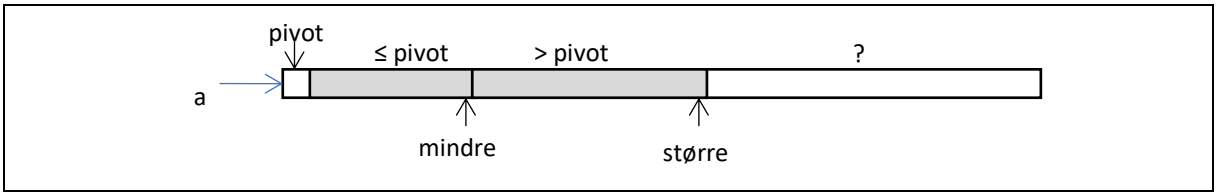
   Da vil Javas synkronisering sørge for at utskriften fra de ulike trådene *ikke* blander seg.

Punkt 2, 4 og delvis 6 er de vanskeligste og vil kunne variere fra problem til problem, pkt.3 og 5 går greit.

7 KVIKKSORT, EKSEMPEL PÅ PARALLELLISERING AV EN REKURSIV ALGORITME

Som nevnt ovenfor, er en av de gylne reglene i parallellprogrammering at før man skriver et parallelt program, lager man et godt testet sekvensielt program som løser problemet. Parallelliseringen er endringer til det sekvensielle programmet.

Vi skal da presentere først en sekvensiell sorteringsmetode KvikkSort (eng. QuickSort), laget av Tony Hoare i 1962. Den fikk en enkel utforming av Nico Lamuto som vi nytter her. Senere skal vi parallellisere denne. Idéen er enkel: Velg ut et element i arrayen, kalt pivot-elementet. Bytt om elementene i arrayen slik at alle elementer som er ≤ pivot kommer til venstre for alle de andre elementene som er > pivot. For å forenkle koden plasserer vi selve pivot elementet helt til venstre i den delen vi nå sorterer. Når vi er helt ferdige med sorteringen, plasseres 'pivot' elementet mellom de to delene, Arrayen a er da ikke ferdig sortert, men hvis vi nå for hver av de to delene gjør samme type oppsplitting *med nye valg av pivoter*, så kan de igjen oppsplittes gjentatte ganger til alle de mange delene til slutt har en lengde på 1 eller 0. Da er a[] sortert fordi ethvert element da står til høyre for et annet element som er mindre eller lik dette.



Figur 7.1 *Idéen bak Kvikksort. Vi velger først et vilkårlig element pivot, så bytter vi om på elementene i a[] slik at vi får de som er mindre eller lik pivot til venstre, så alle de som er større enn pivot til høyre og pivot midlertidig helt til venstre. Til slutt plasseres pivot mellom de to delene Dette gjentas på hver av de to delene, på hver av disses to deler igjen osv. til a[] er sortert.*

Her er skjelettet til programmet som gjør denne sorteringen med oppsplitting gjentatte ganger, og som måler tiden og skriver ut denne:

```
import java.util.*;

/** Sekvensiell implementasjon av Kvikksort */
class SeqKvikk {
      int [] a;

      /** bytter om a[i] og a[j] */
      void bytt(int [] a, int i, int j) {..}

      /** Innpakning for for sKvikk  - enklere kall  */
      void sKvikk(int [] a) { sKvikk(a, 0,a.length-1);}

      /** Rekursiv, sekvensiell KvikkSort av a[lav..høy] */
      void sKvikk (int [] a, int lav, int høy) {..    }

      /** Konstruktor, fyller a[0..n-1] med tilfeldige tall */
      SeqKvikk(int n) {.. }

       /** Tar tider, kaller sKvikk og gjør en enkel test */
       void utførOgTest() {.. }

       public static void main (String [] args) {
             new SeqKvikk(Integer.parseInt(args[0])).utførOgTest();
       }
} //end SeqKvikk
```

Program 7.1 *Skjelettet for den sekvensielle programmet for Kvikksort som starter i main, som først lager et objekt av klassen SeqKvikk med kall på konstruktøren og deretter kaller utførOgTest.*

Koden for konstruktøren og utførOgTest:

```
     SeqKvikk(int n) {
           a = new int [n];
           Random r = new Random(1337157);
           for (int i =0; i< a.length;i++)
                 a[i] = r.nextInt(a.length); // random fill >=0
      } // end konstruktør

      void utførOgTest( ) {
           long  t = System.nanoTime();       // start klokke
           sKvikk(a);
           t = System.nanoTime()-t;
           System.out.println("Sekvensiell Sort av "
                               +a.length+" tall paa:"+
           ((double)(t)/1000000.0) + " millisek.");

            // test
            for (int i = 1; i< a.length; i++) {
                  if (a[i-1] > a[i] ) {
                  System.out.println("FEIL  a["+(i-1)+"]:"
                                    +a[i-1]+"a["+i+"]:"+a[i]);
                  return;
            } }
      } // end utførOgTest
```

Program 7.2 *Koden for konstruktøren, som oppretter a[] og fyller den med tilfeldige tall <n; og utførOgTest(), som tar tida med System.nanoTime() som gir tida i nanosekunder (milliardedels sekunder). Den kaller så sKvikk(), skriver ut tida i millisekunder og gjør en enkel test på om sorteringa gikk bra.*

Koden er forklart under kodelistingen. Testen som utføres er strengt tatt ikke god nok. En enkel og komplett test ville være å sortere de samme tallene med Javas innbygde sorteringsalgoritme: java.util. Arrays.sort, og så sammenligne de to sorteringene element for element. Man kunne også ta tida på Arrays.sort og sammenligne tidene (se oppgave 1).

```
void bytt(int [] a, int i, int j) {
      int t = a[i];
      a[i] = a[j];
      a[j]=t;
  }// end bytt

  /** Rekursiv, sekvensiell KvikkSort av a[lav..høy] */
   void sKvikk (int [] a, int lav, int høy) {
     int ind =(lav+høy)/2,
         pivot = a[ind];
     int større = lav+1,  // hvor lagre neste '> piv'
         mindre = lav+1;  // hvor lagre neste '<=  piv'

     bytt (a,ind,lav);     // flytt 'piv' til a[lav] , sortér resten

     while (større <= høy) {
           if (a[større] < pivot) {
                 // a[større] er 'mindre' - bytt
                 bytt(a,større,mindre);
                 ++mindre;
           }
           ++større;
```

```
      } // end gå gjennom a[lav+1..høy]

      bytt(a,lav,mindre-1);     // Plassert 'piv'- mellom store og små

      if ( mindre-lav > 2)  sKvikk (a, lav,mindre-2); // sorter <= pivot
      if ( høy-mindre > 0)  sKvikk (a , mindre, høy); // sorter > pivot
  }// end sKvikk
```

PROGRAM 7.3 *Koden for selve sorteringen: metodene sKvikk og bytt.*

Bytt er rett fram kode. Koden til sKvikk går i to faser. Først er det et valg av **pivot**-element og oppdeling av den delen av arrayen som pekes ut i kallet med lav og høy. Deretter kommer to rekursive kall – ett på de som er ≤ **pivot** og ett på den delen hvor de som er > **pivot** ligger. Selve logikken til oppdelingen illustreres av fig 1.6. Vi har to pekere i en løkke: **større**, som peker på den plassen vi vil ha *neste* element som er > **pivot**; og mindre som peker på den plassen hvor vi vil ha *neste* element som er **≤ pivot.** Variabelen **større** økes alltid med 1 i hver løkkegjennomgang hvor vi ser på elementet **a[større].** Er **a[større] ≤ pivot**, så bytter vi det med **a[mindre]** som jo peker på det elementet som er 'lengst-til-venstre' av de som er > **pivot.** Så øker vi **mindre** med 1.

Merk at vi plasserer **pivot** mellom de to delene når vi er ferdige. **Pivot** står da på sin endelige plass i sorteringen. Den skal aldri mer flyttes og er ikke med på den videre todeling (som egentlig da er en tredeling) av hver del som vi skal dele videre opp. Behandlingen av **pivot** er litt spesiell – først tar vi og setter den helt til venstre i **a[lav].** Så deler vi resten av arrayen i to deler, og så bytter vi **pivot**, som står på i **a[lav]** med det elementet som står lengst til høyre av de som er ≤ **pivot**: **a[mindre-1].** Det er to grunner til dette. Hvis **pivot** viste seg å være det største av alle elementene vi nå skal sortere, ville vi få en uendelig rekursjon hvis vi ikke gjør dette fordi vi ikke fikk delt opp i to deler, men i en. Den andre grunnen er at vi da plasserer **pivot** på sin endelige plass, og at summen av de delen vi sorterer videre er ett element mindre. Dette gjør at programmet vil terminere uansett hvor uheldige vi er i valg av **pivot**.

Etter at vi har byttet inn **pivot** mellom de to delene, gjør metoden kall på seg selv for de to delene til venstre og høyre for **pivot** hvis disse har større lengde enn ett element. Det er særlig denne kode-delen som blir annerledes i en parallell versjon av kvikksort vi skal se på i neste avsnitt.

Denne koden for Kvikksort er spesielt rask for alle små verdier av n, og like rask for større verdier av n som den innebygde sortingsmetoden **Arrays.sort(..)** i biblioteket **java.util**, som er en annen og mer komplisert koding av Kvikksort.

7. EN BLANDET PARALLELL OG REKURSIV KVIKKSORT

Grunnidéen i en parallell versjon av kvikksort er at vi bytter ut de rekursive kallene med at vi istedenfor starter en ny tråd for hvert av de to kallene. Men som vi så av de to foregående eksemplene tar det en viss tid å starte og stoppe tråder, og sortering går meget fort (fig. 6.1). Vi må derfor lage en blandet algoritme som nytter tråder når vi f. eks deler opp en del av arrayen som er lenger enn 50 000 elementer, men som nytter rekursjon for kortere deler.
Skjelett-koden til programmet for parallell sortering en ganske likt det for sekvensiell kvikksortering:

```
import java.util.*;
import java.util.concurrent.*;

class ParaKvikk {
      int [] a;
      int antTråder = Runtime.getRuntime().availableProcessors();
      final static int PARA_LIMIT = 50000;

      synchronized void  tellOppAntTråder () { antTråder ++;}

      void bytt(int [] a, int i, int j) {…}

      void pKvikk(int [] a) {  pKvikk(null,a, 0,a.length-1); }

      void pKvikk (CyclicBarrier b,int [] a, int lav, int høy) {…}

      ParaKvikk(int n) { … } // konstruktor

      void utførOgTest() {… }

      public static void main (String [] args) {
             new ParaKvikk(Integer.parseInt(args[0])).utførOgTest();
      }

      class Para implements Runnable{
            int [] a; int lav,høy;CyclicBarrier b;
            Para(CyclicBarrier b, int []a,int lav,int høy) {
                  this.a=a;this.b=b;this.lav=lav;this.høy=høy;
                  tellOppNumThr();
               }
               public void run() {
                     pKvikk(b,a,lav,høy);
               } // end run    }
} //end ParaKvikk
```

Program 7.1 *Skjelettkoden for parallell kvikksortering. Konstruktøren paraKvikk har samme kode som konstruktøren SeqKvikk i prog. 6.3. Også metoden bytt er identisk med den sekvensielle bytt.*

Vi ser at internt i klassen **ParaKvikk** har vi i tillegg til arrayen **a**, en variabel **antTråder** som spør operativsystemet hvor mange kjerner vi har og da hvor mange tråder ***k*** vi kan starte i parallell på denne maskinen. Den vesentligste endringen i forhold til den sekvensielle versjonen, er at vi har innført en indre klasse **Para** som inneholder **run()** - metoden som er den koden vi skal utføre i denne tråden parallelt med de andre trådene. Den parallelle koden er et kall på **pKvikk** for å få sortert den delen av **a[]** som denne tråden skal sortere.

```
void pKvikk (CyclicBarrier b,int [] a, int lav, int høy) {
      int ind =(lav+høy)/2,
      piv = a[ind];
      int    større=lav+1,  // hvor lagre neste 'større enn piv'
      mindre= lav+1;        // hvor lagre neste 'mindre enn piv'
      bytt (a,ind,lav);     // flytt 'piv' til a[lav] , sortér resten

      while (større <= høy) {
         if (a[større] < piv) {
            bytt(a,større,mindre);
            ++mindre;
         }
         ++større;
      }

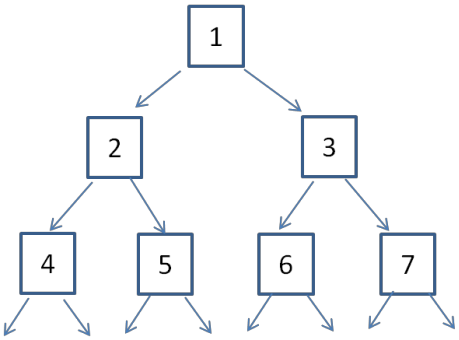
      bytt(a,lav,mindre-1);        // Plassert 'piv' mellom store og små

      if ( høy-lav > PARA_LIMIT){
         CyclicBarrier b2 = new CyclicBarrier(3);
         new Thread(new Para(b2,a,lav,mindre-2)).start(); //  <= piv
         new Thread(new Para(b2,a,mindre,høy  )).start(); //   > piv
         try {  // wait on own two calls to complete
                  b2.await();
          } catch (Exception e) {return;}
      } else {
          // korte arraysegmenter raskere med rekursjon
          if ( mindre-lav > 2) pKvikk (null,a, lav,mindre-2);//  <= piv
          if ( høy-mindre > 0) pKvikk (null,a, mindre, høy); //  > piv
      }

      if (b!= null) {
           try {  // signaliser til kallende tråd at denne er ferdig
              b.await();
           } catch (Exception e) {return;}
      }
  }// end pKvikk
```

Program 7.2 *Sorteringsalgoritmen pKvikk. Delingen av a[] er akkurat den samme som ved sekvensiell sortering. Det interessante er hvordan vi parallelliserer de to delene etter oppsplittingen. Hvis hele den delen vi skal sortere er større enn PARA_LIMIT, oppretter vi en syklisk barriere som skal vente på 3 tråder – de to nye trådene som startes og den tråden som skaper disse. Ventesetningen like etter at de to trådene som skapes er at den tråden som skapte disse to trådene, venter på at de begge er ferdige. Helt på bunnen av metoden ser vi et nytt await() kall. Det er signalet til den tråden som kalte denne metoden, at denne tråden er ferdig – altså et signal oppover.*

Koden forklares i teksten under program 7.2. I oppgave 2 skal dere prøve ut og forklare tidsforbruket uten å kode dette med rekursiv løsning for kortere deler, men bare med tråder.



***Figur 7.2** Kall-treet. Parallell Kvikksort bruker enten tråder eller rekursjon for å løse problemet. Dette kan illustreres som et tre (med roten i toppen) av de ulike instansene av metoden pKvikk hvor ett høyere nivå gjør to kall på neste nivå. Merk forskjellen på rekursjon og tråder. Starter vi tråder, vil de (grovt sett) starte i den rekkefølgen som de her er nummerert: først 1 som starter 2 og 3, så vil 2 starte 4 og 5, mens 3 vil starte 6 og 7, osv. Dette kalles bredde-først. Hvis det imidlertid dreier seg om rekursive kall, da vil 1 starte 2 som vil starte 4,…og først etter at alle kall som genereres av 4 og dets 'barn' har returnert til 2, vil 5 bli kalt, så 3,6 og 7. Dette kalles dybde-først traversering av treet.*

7.1 EFFEKTIVITETEN PÅ SEKVENSIELL OG PARALLELL KVIKKSORT

Vi testet de to versjonene av Kvikksort på to ulike maskiner, én med 8 kjerner og én med 64, og resultatene er referert i tabell 1.2 og 1.3.

	10	100	1 000	10 000	100 000	1 mill	10 mill	100 mill
Sekvensiell	0,01	0,05	0,70	13	18	89	963	11196
Parallell	0,01	0,06	0,84	16	24	67	356	3579
SU (Speedup)	1,00	0,83	0,84	0,81	0,75	1,32	2,71	3,12

***Tabell 7.2** Antall millisekunder det tar å sortere en array med tilfeldig innhold med den sekvensielle og parallelle versjonen av Kvikksort (parallelt for deler som er lengre enn 50 000) og Speedup (sekvensiell/parallell). Kjørt på en Intel i7 870, 3Ghz klokke med 8 kjerner.*

	10	100	1 000	10 000	100 000	1 mill	10 mill	100 mill
Sekvensiell	0,01	0,05	0,81	18	21	197	1413	16497
Parallell	0,01	0,06	0,99	21	56	106	1332	5025
SU (Speedup)	1,00	0,83	0,82	0,85	0,38	1,85	1,06	3,28

***Tabell 7.3** Antall millisekunder det tar å sortere en array med tilfeldig innhold med den sekvensielle og parallelle versjonen av Kvikksort(parallelt for deler som er lengre enn 50 000) og Speedup (sekvensiell/parallell). Kjørt på en maskin med 4 Intel Xeon prosessor L7555, 1.87Ghz klokke, totalt med 64 kjerner.*

Kommentarer til tabellene. Først tidene for sekvensiell utførelse. Vi ser at tidene er langt mer enn 20 ganger så lange når vi går fra å sortere 1000 til 10 000 tall (de burde bare vært litt mer enn 10-ganger) og tilsvarende videre til 100 000. Det kan forklares med at mesteparten av arrayen vi sorterer først ikke får plass i nivå 1 cachen, men i nivå 2, og for 100 000 tall i nivå 3 cachen og hovedhukommelsen som jo er mye langsommere.

Det beste vi synes å oppnå er at den parallelle versjonen går om lag 3 ganger så raskt som det sekvensielle programmet og ikke så mange ganger fortere som vi har kjerner. De parallelle

resultatene kan forklares ved flere faktorer, men det interessante spørsmålet er: Hvis vi har k kjerner, hvorfor går det ikke k ganger fortere?

Det er i hovedsak to grunner. Først er ikke parallelliseringen optimal. Den første oppdelingen av **a[]** i to deler skjer sekvensielt og like langsomt som den sekvensielle algoritmen. Først da starter vi 'bare' to tråder. Når hver av disse igjen deler opp hver sine deler i to, får vi fire tråder som er aktive (og ikke venter), osv. Vi får altså ikke brukt alle kjernene helt fra starten av programmet.

Den manglende parallelliseringen er imidlertid ikke alltid hovedforklaringen. I figur 1.2 ser vi en strek som forbinder hver prosessor med hovedhukommelsen. Det kalles en hukommelses- eller data-kanal, og kan ikke gå fortere enn hukommelsen kan operere. Vi husker også at hovedhukommelsen var mye langsommere enn hver kjerne. Det betyr kort sagt at det er en kø av kjerner på hukommelseskanalen for å lese og skrive i hovedhukommelsen, og det blir mye venting. Vi greier ikke å få trådene til å jobbe med full hastighet.

Lite køing på hukommelseskanalen har bare problemer som har lite data og som gjør mer arbeid med hvert dataelement enn de problemene vi har sett på her. Et slikt problem ("Selgerens rundreise") ser vi på i oppgave 3. Da vil data som det jobbes med i kjernen i stor grad være in cache-hukommelsene og antall lesinger og skriveoperasjoner i hovedhukommelsen vil ikke skje så ofte at det skaper kø på hukommelseskanalen. Parallelliseringen av kvikksorteringen kan imidlertid ikke sees på som mislykket – at store sorteringer går tre ganger så fort er klart noe man vil ønske i praksis.

8 AMDAHLS LOV

Vi ser av Kvikksort-eksempelet først hadde en sekvensiell del (dele arrayen i to) , og så kunne parallelliseringen begynne. Dette er generelt for mange algoritmer, at de har ofte først har en sekvensiell del. Et eksempel kan være at den sekvensielle delen tar 10% av tiden og de delene som kan parallelliseres tar da 90% av tiden når vi kjører alt sekvensielt. Vi ser da at det beste vi da kan oppnå av en parallell algoritme, er at den maksimalt vil gå 10 ganger så fort som den sekvensielle. Uansett hvor mange hundre- eller tusenvis av kjerner vi bruker på den parallelle delen, kan vi ikke få den til å gå raskere enn på 0,0 sekunder. Vi står da igjen med 10% av beregningene i den sekvensielle delen – altså maksimalt 10 ganger raskere enn en sekvensiell beregning..
Generelt, anta at vi har en algoritme som må utføre p % av algoritmen sekvensielt og at vi greier å få den delen av koden som kan parallelliseres til å gå k ganger raskere. Da er den maksimale hastighetsforbedringen S vi kan få:

$$S = 100/(P + (100-P)/K)$$

Setter vi inn at den sekvensielle delen p = 10% og antall kjerner k = 1000, ser vi at S blir 9,9 – vi greier altså å gå 9,9 ganger fortere. Greier vi å få p ned i 1% blir S=91 med de samme forutsetningene. Lærdommen fra Amdahls lov er at før eller siden er det den delen som ikke kan parallellisere som vil dominere kjøretiden. Det er nesten alltid sånn at noe må gå sekvensielt – f.eks. skal data leses inn, svar skal skrives ut og selve programmet må leses inn i hukommelsen og settes i gang. Vi husker også at det kan ta noen få millisekunder å starte én tråd, så det tar alltid noe tid før vi kan få startet parallell kjøring. Selv om vi kanskje håpet at med 1000 kjerner ville problemet vårt gå 1000 ganger så fort, så trenger vi ikke fortvile. At beregninger går fra 9 til ca. 90 ganger fortere er klart nyttige resultater.

9. OM PROSESSOREN, JAVA-KOMPILATOREN OG HUKOMMELSES-MODELLEN

Det er skrevet et større notat kalt «Java memory model» som beskriver hvordan tråder skal kunne bruke hukommelsen under eksekvering. Dette er et meget komplekst dokument og ikke nødvendig å lære seg når man skriver parallelle programmer beskyttet av låser som de som beskrives i dette kompendium og i kurset. I hovedsak er dette et notat for de som skriver kompilatorer med optimaliseringer for Java, men det kan likevel være greit å se på ett tilfelle hvor en vanlig programmerer lett kan få seg en overraskelse.

I begynnelsen av kom ga vi inntrykk av at instruksjonene blir utført i den rekkefølgen de er i programteksten. Det er ikke nødvendigvis riktig av to grunner. Først vil selve prosessoren gjerne bytte om på instruksjoner den holder på å utføre hvis det kan gå fortere, og *dersom det ikke har noen* innvirkning på sluttresultatet. Det samme prøver også java-kompilatoren. Det er mye tid å spare på å flytte rundt på instruksjoner når de utføres, f.eks. at flere andre operasjoner utføres samtidig som vi holder på med en flyttall (double)-operasjon, som tar lang tid. Slik ombytting kan bare foretas hvis *det ikke får innvirkning på sluttresultatet.* Prosessorkjernene har også nå fått egne instruksjoner som kan kjøre flere instruksjoner av type flyttall i parallell, f.eks. multiplikasjon dersom variablene til disse multiplikasjonene, som skal utføres, ligger etter hverandre i lageret. Slik omorganisering av rekkefølgen på instruksjoner må imidlertid ikke gå ut over slik vi har tenkt når vi laget programmet - programlogikken. Trenger vi resultatene av én beregning i høyresiden i en annen beregning, eller i for eksempel i en utskrift til skjerm eller fil eller i en test, blir *ikke* slik ombytting eller parallellkjøringer av instruksjoner foretatt.
Det Java og prosessoren garanterer deg er at du får utført et program som gir eksakt samme resultat som om programmet ble utført instruksjon etter instruksjon, ovenfra og nedover, en etter en, og hver gang du bruker verdien på en variabel, er den der som om programmet ditt ble utført enkelt og greit ovenfra og nedover.
Siden en programmerer nesten aldri merker effekten av slik ombytting av instruksjoner av prosessoren og kompilatoren, behøver vi ikke dvele mer med det unntatt å advare mot en feil en programmerer kan gjøre. Se på flg. to linjer i et program:

```
x = 12;
y = 19;
```

***Program 9.1**. Tilordning av verdier til to variabler. Hvis vi senere i programmet tester og finner at y er lik 19, kan vi **ikke** dermed slutte at x er lik 12 (selv om det ser ut som om x=12 ble utført 'før' y=19). Siden både prosessoren og java-kompilatoren kan bytte om på tilordningen av verdier til de to variablene, og utsette 'x=12' til verdien av x enten brukes i en annen beregning, i en test eller i en utskrift, kan det godt hende at y får sin verdi lenge før x får sin verdi.*

Vi kan konkludere at den gamle enkle modellen om at vi har en prosessor og en kjerne og at den leser og utfører instruksjonene en etter en, ovenfra og nedover, ikke er helt riktig, men at det ikke gjør noe i de aller fleste tilfeller i et program med flere tråder når vi synkroniserer med låser. Dette er viktig for oss som programmerere. Uten denne enkle modellen vil det nesten ikke være mulig å skrive riktige programmer.

9.1 HVA SKJER NÅR VI SYNKRONISERER FLERE TRÅDER PÅ SAMME OBJEKT

Når to eller flere tråder synkroniserer på **samme** objekt (sier f.eks await på samme CyclicBarrier , eller kaller en metode som er beskyttet ReentrantLock) vil alle disse oppleve følgende:

- Alle kode som er ovenfor synkroniserings-setningen i trådene og som hittil ikke er utført (f.eks, er utsatt av optimaliseringsgrunner), blir utført.
- Alle data som er skrevet på av de synkroniserende trådene blir skrevet ned i hovedhukommelsen (eller i alle fall en *felles* nivå3 cache), noe som er vanlig på en CPU med mange kjerner).

Dette betyr at alle tråder som har synkronisert på felles variabel *ser samme verdier* på felles variable slik de er hittil skrevet på av de deltagende trådene. Tråder som derimot ikke har synkronisert på dette felles synkroniseringsobjektet, er ikke garantert å se de siste verdiene på slike felles data.

Det er også slik at alle felles data (som har blitt endret av en av trådene) blir skrevet ned i hovedhukommelsen når trådene er ferdige, dvs. er ferdige med siste setning i run-metoden.

10. OPPSUMMERENDE KOMMENTARER OM OPTIMALISERING

Noen programmer er man avhengig av at går raskt, og da oppdager man en egenskap med Java, at første gang utfører en viss type kode kan koden ta en viss tid, mens neste gang kan samme koden gå mye raskere. Kjøre man samme koden veldig mange ganger har vi sett at den i Java kan den typisk gå fra 20 til flere 1000 ganger raskere enn første gang.

Det er riktig å påpeke at de fleste programmeringsspråk kan optimaliseres på denne måten slik som C, Fortan og C#. To ting bør man merke seg:

- Denne optimaliseringen kan gjøres totalt av kompilatoren når vi kompilerer vårt program og da kan man f.eks i C spesifisere om man vil ha o1, o2 eller o3 optimalisering, og da blir hele programmet optimalisert og blir da en del større enn den holdningen Java har, at bare den koden som virkelig blir brukt kompileres til maskinkode, og det er antall ganger den koden-biten blir eksekvert som avgjør hvor sterkt den etter hvert optimaliseres. I Java er det også slik at hvis det brukes en klasse eller metode fra en biblioteks-klasse (som Arrays.sort i Fig 17.3 nedenfor) må bytekoden fra biblioteket først lastes ned før den kan kompileres og så utføres, og blir da langsommere første gang den utføre enn kode som ligger i selve programmet.
- I hvor stor grad det er mulig å optimalisere kode avgjøres i stor grad om språket som skal optimaliseres er statisk typet eller ikke (dvs. at hver variabel får fastlagt sin type som int eller String, som deklarasjoner i selve koden eller ikke). De fleste vanlige programmeringsspråk er statisk typet. Men i Pyton, som ikke er statisk typet, kan en variabel 'x' for eksempel av og til inneholde et flyttall og av og til en tekst i samme programutførelse. Det er først når programmet kjører og f.eks. setningen: y = x+ 1 skal utføres er at typen til x må bestemmes, og det skjer da under selve kjøringen og forsinker utførelsen og begrenser selvsagt da også hvor mye koden kan optimaliseres.
- Uansett hvordan og hvor mye et program optimaliseres, så vil det utad, dvs. det som skrives ut til fil, på skjerm e.l. være det samme resultat som det et uoptimalisert program vil gi. Dvs. at vi kan alltid tenke oss at programmet blir utført slik vi har lært i begynnerundervisningen, ovenfra og nedover, linje for linje til vi er ferdige.

Vi vet fra innføringskapitlene at først oversetter *javac* (java-kompilatoren) koden – f.eks klassen *MittProgram* din til byte-kode (det ligger på filen *MittProgram.class).* Dette er kode for en tenkt maskin med enkle byteinstruksjoner. Så kaller du opp kjøresystemet *java* (som også kalles JVM – Java Virtual Machine) for å kjøre programmet. Det første den gjør er å oversette all byte- kode den utfører til maskin-instruksjoner på den maskinen du bruker. Java oversetter *ikke* hele programmet ditt, bare de delene (metodene og klassene) du virkelig utfører. Første gang du kjører får du altså tider som både er oversetting til maskinkode + selve kjøretida for din kode. Hvis den så kjøre en viss del av koden din flere ganger), vil den som vi tidligere har sett begynne å optimalisere på den maskinkoden den har laget. Enda mer kjøring kan gi ytterligere optimalisering. Optimalisering kan grovt sett beskrives som at den lager enda lurere og raskere maskin-kode av de delene av programmet du kjører ofte. Slik optimalisering kan gå i 2-3 omganger og bli stadig raskere. Det du derimot skal være sikret, uansett hvor mye maskinkoden forbedres, er at vi har sekvensiell semantikk. Dvs *du kan store på at du kan tenke og feilsøke programmet ditt ut fra selve Java-koden og du kan tenke på den som om koden blir utført ovenfra og nedover, linje for linje, løkke for løkke som om det aldri ble utført noen oversettelse til maskinkode eller senere optimalisering av denne.*

La oss se på ett eksempel fra beregning av en av radene i tabellene 10.1 og 10.2 nedenfor:

```
class C { int i;
   C(int i){this.i =i;}
   int les () { return i+10;}
 } // end C

   String s = "new Class C + metodekall";

   long t = System.nanoTime();
      for (int i = 0; i< n; i++) {
           k= new C(k).les();
      }
   double d = (double) ((System.nanoTime() -t)/(n*1000.0));
```

Vi ser at dette er en kode som først gjør en *new C(k)* og så kaller les-metoden i objektet vi har laget av klassen C og skriver ut tiden for det. Dette gjør vi ant ganger , ant =1,2,3,100,10000, 100000 ganger. Første gang tok det 2697 µs (milliondels sekund), mens det med ant=2 tok i snitt 0.45µs. En forbedring på ca. 5000 ganger raskere! Selvsagt kan vi gjøre samme for ulike Java-konstruksjoner (se tabellen nedenfor). Grunnen til at det går mye raskere er ikke bare at vi oversetter til maskinkode første gang, men også at det i JVM bygges opp datastrukturer for dine klasser og metoder slik at neste gang har JVM en mye lettere og raskere jobb med å kjøre ditt program.

n= ant. ganger	1	2	3	100	10000	100000	X bedre, (speedup)
for-løkke	0,3	0,15	0,03	0,018	0,009	0,007	42
metode-kall	2697	0,45	0,06	0,054	0,026	0,026	103730
new int[100]	1,2	0,6	0,24	0,195	0,151	0,136	33
array copy med for-loop, n=100	1,8	1,5	2,64	2,500	1,177	0,188	9
System.arraycopy, n=100	5,7	0,3	0,15	0,126	0,072	0,064	89
new Thread med start & join()	3015	336	66,6	61,68	61,87	61,86	48
new C(int) og metodekall	2697	0,45	0,15	0,21	0,035	0,035	77 057
Int [] array read	0,3	0,3	0,06	0,036	0,012	0,012	25
Innstikk-sortering (n=100)	46,6	42,8	42,42	21,27	19,60	1,45	32

Tabell 10.1 Kjøretider fra sekvensielt testprogram i **2017** i µs med Java 8.0 for ulike Java-konstruksjoner

og et enkelt sorteringsprogram som funksjon av antall utførte ganger + speedup for n = 100 000, Intel i7-7600 @3,4 Ghz.

n	1	2	3	100	10000	1000000	x bedre (SU)
for-loop	0,7	0,2	0,023	0,023	0,002	0	350,0
metodekall	9,7	0,9	0,1	0,102	0,005	0	1940,0
int[] new	0,6	0,3	0,152	0,24	0,137	0,087	6,9
copy array for-loop	2	2,3	2,793	1,336	0,635	0,011	181,8
arraycopy	4	0,7	0,208	0,036	0,033	0,021	190,5
new Thread,start&join	2576,9	301,8	210,659	175,962	0		14,6
new Class C m.metodekall	2301,6	8,4	0,272	0,011	0	0	209236,4
int [] a skriv	0,6	0,5	0,028	0,006	0,01	0,007	60,0
int [] les	0,3	0,2	0,023	0,005	0,013	0,007	23,1
double to long	4,2	1,2	0,164	0,007	0,044	0	95,5
insertSort int[]	118,3	154	47,332	3,201	1,65	1,771	71,7
Arrays.sort	471	70,9	45,023	4,68	1,221	1,187	385,7

Tabell 10.2. En Litt forenklet versjon av fig. 2.1. Kjøretider fra **2022** i µs + speedup fra n=2 til n= 1000 000, Java 14.0 for ulike Java-konstruksjoner og to sorteringsprogram som funksjon av antall utførte ganger av et sekvensielt testprogram på en AMD Ryzen 5 3500U, 2.0 3.4GHz. Kommentar: New Thread eksemplet er bare kjørt for n= 1,2,3 og 100, pga. tidsforbruket.

Tallene ovenfor varierer noe for hver gang de kjøres, men viser klare trekk som at særlig metodekall og det å lage objekter effektiviseres ekstremt, men at arrayer ikke effektiviseres i samme grad. Oppmuntrende er det at brukerkode som en innstikks-sorterings algoritme som vi har skrevet (og som hver gang sorterer en ny usortert double array av lengde 100) kan effektiviseres med en faktor ca. 70 .

Man kan også lure på hvilke resultater som kommer fra en noe raskere maskin og hva som kommer fra en bedre optimalisering i Java i disse to tabellene. Uten å ha kjørt store tester på dette vil jeg antyde at en bedre speedup kommer fra Java, men lavere tider for samme 'n' vel kommer fra en raskere maskin.

Delvis kommer det at din kode blir forbedret, men også at de data og variable du har i ditt program blir kraftig forbedret. Uansett behøver en Java-programmerer bry seg om hvordan det skjer, men bare vite at ethvert Java-program med tiden vil gå stadig fortere, men alltid produsere samme svaret.

Dette at optimaliseringen gir oss samme svaret som om vi ikke hadde optimalisert koden, kalles ***sekveniell konsistens*** og er meget viktig for oss når vi skal feilrette programmet. Vi kan tenke på vårt program som om det blir utført linje for linje, ovenfra og nedover. Selve optimalisingen kan man **slå av** med å starte programmet vårt slik etter kompileringen til bytekode:

```
>java -Xint MittProgram  … de vanlige parametrene …
```

(brukes bare ved debugging hvis man tror at optimaliseringen har 'ødelagt' programmet, noe den nesten aldri har gjort.)

Det er viktig å vite at kallet på en metode eller det å lage et objekt av en klasse er knyttet til ***kallstedet***. Det betyr at hvis du f.eks kaller samme metode fra et annet sted i din kode, vil den også bli optimalisert der på ny. Det som grovt sett skjer er at det ikke blir foretatt et hopp til en optimalisert metode, men at denne optimaliserte koden blir lagt inn direkte der kallet på metoden er. Vi kan si at kallet som blir optimaliser ved å bli fjernet på kallstedet og erstattet med metodens kode.

Likevel kan dette lære oss en del om hvordan vi bør skrive våre programmer:

1. Siden metodekall og det å bruke klasser og lage objekter av disse effektiviseres sterkt, bør vi ikke være redde for å bruke disse i rikt monn i våre programmer. Mange algoritmer kan være ganske kompliserte med mange steg og løkker. Hvis hvert slik steg kan utformes som en egen, mindre metode, og at det hele så bygges sammen med en overordnet metode som i rekkefølge kaller disse mindre metodene, får vi et program som både er lettere å forstå og debugge, og mulig også raskere (optimalisatoren liker godt små metoder).
2. Når vi skal lage parallelle programmer med tråder, ser vi at det bare er det første trådobjektet som tar lang tid å lage og starte/avslutte (koster ca. 2,5 millisek). De neste, nr 2,3,.. tar hver bare ca 1/10-del av denne tida. Antall tråder vi deklarerer i et parallelt program behøver derfor ikke ha mye å si for kjøretiden. Ofte vil f.eks. å bruke flere tråder som vi har kjerner være et vellykket valg for løsning av et parallelt problem.

3. Ofte er det også slik at noen problemer, som sortering av tall, går så raskt at det ca. 4 millisek. å sortere si 25 000 tall. Det gjør at en sekvensiell sorteringsalgoritme vi være ferdig før den parallelle versjonen av algoritmen har greid å ha fått starte sine tråder og begynt å løse problemet, De fleste parallelle programmer vil derfor inneholde en første metode av typen:

```
void løsProblemet ( ..param..) {
    if(størrelsen_av_problemet   <   minimum_størrlse)
        løsProblemetSekvensielt(..param..);
    else løsProblemetParallelt)..param…);
}// end løsProblement
```

4. Med få unntak, ikke prøv å optimaliser koden selv, optimalisatoren er langt flinkere enn deg.
5. Unntakene er at arrayer med to dimensjoner alltid bør leses, beregnes og skrives ***radvis***, og at de data vi leser/skriver i en løkke helst bør passe inn i størrelsen på cache, nivå1 eller cache-nivå 2.
6. Råd nr. 5 synes umulig å følge i f.eks matrisemultiplikasjon av to n*n matriser B og C for å lage en ny matrise : A= BxC, hvor den matematiske definisjonen krever at element A[i,j] er lik summen av hvert element i raden B[i] ganger hvert element i *kolonnen* C[j]. Det synes som om vi *må* lese matrisen C kolonnevis. Her kan vi gjøre det trikset har sett på forelesningene, hvor vi først, før multiplikasjonen bytter om elementene C [i,j] med C[j,i] – og så multipliserer hver *rad* i den ombyttede matrisen C med hver *rad* i B. Da blir A slik det er matematisk definert. Det kalles å transponere matrisen C, og det er en meget rask algoritme – hvert element blir bare lest og skrevet én gang, mens i selve multiplikasjonen blir hvert element i B og C lest og summert n ganger. Den sekvensielle multiplikasjonen vi da gå om lag 40 ganger fortere fordi vi ikke får cache-miss og får brukt prefetch-mekanismen. Når man er ferdig med matrise-multiplikasjonen A = BxC, bør man transponere tilbake C slik at vi får den samme C som vi begynte med - fortsatt en operasjon som er meget rask og som igjen omtrent ikke berører kjøretiden. Merk at transponering av C er et råd som ikke er vesentlig hvis matrisene er meget små (f.eks 4x4 matriser) hvor hele matrisen får plass i level 1 eller level 2 cache.
7. Du skal selvsagt, når du lager en parallellisert løsning på et problem, bruke den raskeste, sekvensielle metoden som et utgangspunkt for den parallelle løsningen.
8. Husk at når vi lager en parallell løsning med *k* tråder har vi laget *k* sekvensielle programmer som deler felles kode, men som har ulike deler av problemets data. Hvis de data som deles har blitt delt opp i *k* deler, er det å forvente at hver del passer langt bedre inn i cache nivå 1 og 2. Hvis det er tilfellet, vil et slikt parallelt program kunne gå raskere enn *k* ganger fortere enn det opprinnelige sekvensielle løsningen fordi mer av data ligger høyere opp i cache-hierarkiet.
9. Derimot er det et annet forhold som peker mot at programmet ikke går så fort, og det er forbindelsen mellom kjernene og hovedhukommelsen, kalt datakanalene. Når *k* kjerner samtidig vil skrive og lese via datakanalene i hovedhukommelsen, blir det ofte kø og ventetider med langsommere eksekvering som resultat.
10. For å oppsummere, det er et rimelig håp at et parallellisert program på en multikjerne PC med *k* kjerner, vil ha en speedup på ca. *k*.

11. Feilaktige antagelser i PRAM-modellen

Som tidligere nevnt har det blitt laget en teori for å analysere og lage parallelle programmer som kalles PRAM (Parallell Random Access Machine). Den gjør følgende forenklinger for lettere å analysere parallelle maskiner og programmer:

1. Tiden det tar å lese eller skrive i hukommelsen er konstant og settes =1.
2. Vi kan lage programmer som kan ha så mange kjerner vi vil - f.eks. *n* kjerner hvis vi skal sortere *n* tall.
3. Alle parallelle programmer kan startes samtidig på samme klokke-sykel
4. De parallelle kjernene går synkront, dvs. hvis de utfører samme program, vil de i én klokke-sykel utføre eksakt samme instruksjon på samme tidspunkt.

Alle disse påstandene er ganske langt fra virkeligheten og gale, og kan gi feilaktige analyser og dysfunksjonelle programmer fordi:

1. Påstanden om konstant tid for lesing og skriving ignorerer prefetch og cache-systemet. Leser/skriver vi data som befinner seg i et register eller i cache nivå1 vet vi at det går 100-200 ganger fortere enn om vi får en cache-feil og vi må da gå helt ned i hovedhukommelsen for data til hver ny instruksjon. I praksis opplever vi tidsforskjeller på minst faktor 20-40 hvis en litt stor (si minst 100x100) todimensjonal array aksesseres kolonnevis istedenfor radvis.

2. Feilen med at vi kan ikke fritt skrive programmer med veldig mange tråder (si mer enn ca. 1000-100 000) og i tillegg tro at dette går greit fordi man i parallell har like mange kjerner. Dette vil det raskt fylle hele data-maskinen, da hvert objekt av klassen Thread tar en viss plass. Og siden ingen multikjerne maskin i praksis har mer enn 32-64 kjerner, vil det bli en evig køing av tråder på å få eksekvere på en virkelig kjerne med mye utskifting av systemdata i hovedhukommelsen og kø på data-kanalene. Antagelsen av vi har valgfritt antall kjerner tilgjengelig vil lett få oss til å skrive dysfunksjonelle programmer.

Riktignok kan vi med å programmere GPU-en (grafikkprosessoren) kunne få flere tusen prosessorer i parallell som utfører samme instruksjon på ulike data. Men her er vi begrenset til noen tusen kjerner, og det er bare noen typer problemer som løses effektivt med grafikkprosessoren. Dette kurset et kurs i å programmere parallelle algoritmer på en multikjerne CPU. Tilleggskurs i GPU- og programmering av beregningsklynger samt på KI-maskiner finnes på Ifi. Generelt kan det sies at ikke alle typer av problemer lar seg løse mer effektivt på slike spesielle maskiner enn på en multikjerne PC.

1 og 4 : Antagelse om full synkronisering er gal. I praksis startes trådene så raskt man greier sekvensielt av et sekvensielt program. Hvis trådene f.eks. skal gjøre veldig lite (si øke en felles variabel 100 ganger), så kan vi lett oppleve at vi ikke får noen feil fordi den første tråden som starter er ferdig før neste tråd får begynt på å utføre sin kode. Vi kan da lett få det inntrykket at vi ikke trenger å synkronisere adgangen til denne felles variabelen med f.eks en *ReentrantLock* eller som *synchronised.* Elektronikken i kjernene er heller ikke alltid synkronisert slik at vi fysisk sett kan få litt ulik hastighet på de ulike kjernene.

Riktignok har PRAM-modellen blitt modifisert i ulike retninger. og vel særlig antagensen om all aksess til data tar samme tid, men kurset IN3030 finner det uproduktivt og galt å basere sin programmering på en sterkt forenklet teori og modell for parallellprogrammering. Vi trenger ingen annen modell annet enn programmet for det problemet vi skal løse. Å se bort fra en rekke mekanismer i elektronikken og i optimaliseringen i kjøresystemet java (JVM Java Virtuell Machine), finner vi

uproduktivt og tidvis villedende for å finne den beste løsningen og algoritmen for et problem. Kurset IN3030/4040 baserer seg på tidtaking som målestokk på en effektiv algoritme i tillegg til at vi nytter O(n) - analyse av både den sekvensielle og parallelle algoritmen for å fine forventet kjøretid som funksjon av antall dataelementer *n*.

12 MENGDENE I JAVA KAN VÆRE LANGSOMME

Når man ser på hastighet i Java-programmer, så bør man også se på at i alle mengdene i Java-biblioteket som var mengder av basale typer som int, double, long ... nå er byttet ut med tilsvarende klasse- representasjonen av typer, som Integer for int, Double for double, …osv. Vi deklarerer en ArrayList for heltall slik, og her legger vi samtidig inn tallene 1 til n i lista:

```
ArrayList<Integer> liste1 = new ArrayList <Integer> ();
for (int i = 1; i <= n; i++) liste1.add(i);
```

Det som skjer bak kulissene, er at hver int-verdi blir pakket inn i et Integer objekt før det blir lagret med en peker fra lista til dette objektet. (på engelsk: boxing) . Skal vi så lese verdien i Integer-objektet , må heltallsverdiene pakkes ut byte for byte (på engelsk: unboxing) fra et Integer-objekt på 16 byte + 8 byte til en peker til objektet i motsetning til en 'vanlig' int i en array på 4 byte . Det vil si at en Integer i en ArrayList tar omlag 6x så stor plass som en int-variabel, noe som gjør at Integer-objektene raskere fyller opp de ulike cachene, og det tar lenger tid for å gjøre les og skriv på heltallselementene man tross alt behandler.

For å vurdere hvor rask ArrayList er, kan man lage en alternativ implementasjon *IntList* med en heltallsarray med den funksjonalitet som ArrayList har:

```
class IntList{
// representerer k heltall, adressert: 0..k-1
int [] data;
    IntList(int len){data = new int [Math.max(1,len)];}
    IntList() {data = new int [16];}

    void add(int elem) {
        if (len == data.length) {
          int [] b = new int [data.length*2];
          System.arraycopy(data,0, b,0,data.length);
          data =b;
        }
        data[len++] = elem;
    }  // end add

    void clear(){
       len =0;
    } // end clear

    int get (int pos){
        // antar at pos er en array-indeks
         if (pos > len-1 ) return -1;  // error
         else return data [pos];
      } // end get

    int size() {
        return len;
    }
} // end class IntList
```

Program. 12.1 *Klassen IntList, bruk av en array for oppbevaring av heltallsverdier som alternativ til ArrayList*

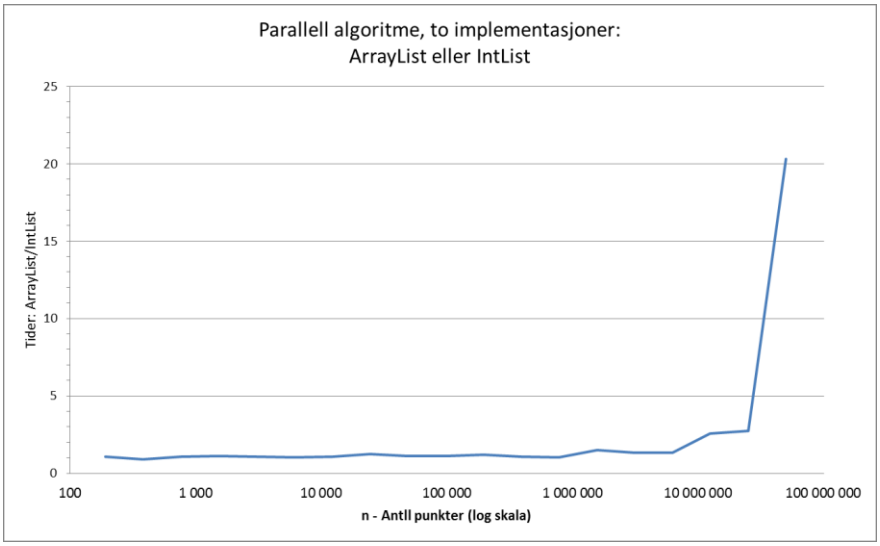
Kjører vi en enkel test av innlegging (add) og les(get) n ganger, ser vi at ArrayList er klart langsommere enn IntList .

Tabell 12.2 *Forholdet i kjøretider for ArrayList/IntList for ulike lengder av listene, og viser hvor mange ganger langsommere ArrayList er enn IntList i et sekvensielt program.*

n	IntList(ms)	ArrayList(ms)	forhold
500 000 000	6 194.84	211 277.47	34.11
50 000 000	146.46	2 054.59	14.03
5 000 000	15.06	41.22	3.05
500 000	1.13	3.44	3.43
50 000	0.10	0.36	1.57
5 000	0.02	0.04	2.50
500	0.00	0.00	2.00

ArrayList er per definisjon sekvensiell. Hvis du imidlertid bruker ArrayList inne i en parallell strøm i et parallelt program, kan en slik ny parallellisering være en dårlig idé. Fordi istedenfor *k* parallelle tråder har du plutselig nå *k*k* parallelle tråder som slåss om de *k* fysiske kjernene. Dette gir vanligvis en hastighetsreduksjon. Parallelle tråder bør ikke selv starte sine *k* tråder osv.

Kjører vi et parallelt program som bruker IntList eller ArrayList mye (testet på et som løste den konvekse innhyllinga til n punkter), får om lag like store forsinkelser ved å bruke ArrayList. Derfor bruker vi IntList.



Figur 12.*1 Vi ser at jevnt over er den parallelle algoritmen som bruker IntList ca 10-15% raskere når n er liten, men blir når n > 50 mill blir IntList mye raskere enn ArrayList.*

13 OM PARALLELLISERING MED STREAMS

I Java 8 og i senere versjoner, er det innført begreper som stammer fra funksjonell programmering – med lambda-uttrykk og strømmer (streams). Grunnen til å nevne det her er at her synes det å være en programmerings-vennlig måte å parallellisere et program. Først en kort innføring i selve begrepene: lambda-utrykk og sekvensielle strømmer og deretter et eksempel på en vellykket parallellisering med parallelle strømmer, og til sist kommentarer til når denne mekanismen er velegnet til generell parallelisering.
(Kan du vanlige sekvensielle strømmer og lambda-uttrykk, kan du hoppe over avsnittene 13.1-13.5.)

13.1 Lambda-uttrykk og strømmer (streams)

Enkelt sagt er *lambda uttrykk* en kompakt måte å skrive (enkle) metoder som vi skal utføre. Kompilatoren *javac* finner da ut hvilke typer parametre vi har i *lambda-metoden* slik at vi kan slippe å skrive det, og mye annet vi kanskje ellers må skrive for å lage en metode. Koden vår blir kortere og mer lettlest. Andre variable må vi deklareres som vanlig (trenger vi et heltall i, deklarerer vi det som: int i;

Vi skal her gå gjennom hvordan *lambda-uttrykk* skrives, deklarasjoner og bruk. Slike uttrykke egner seg vel mest for små korte metoder som kan skrives på noen få linjer. Alt vi kan gjøre med *lambda-uttrykk* kan vi også få til med vanlige metode-deklarasjoner og bruk, men riktig brukt er *lambda-uttrykk* er mer elegant, lettere forståelig og gir brukere langt mindre å skrive særlig når *lambda-uttrykk* kombineres med *streams*.

13.2 Hvordan skrive lambda-uttrykk

Det er to måter å skrive lambda-uttrykk:

(parametre) -> uttrykk **eller** (parametre) -> { setninger;}

Begge tar parametre spesifisert på venstre side og bruker den på høyre side til å evaluere et uttrykk eller utføre setningene i klammeparentesen. Et lambda-uttrykk kan returnere en verdi eller bare gjøre noe.

Her er eksempler på lambda-uttrykk:

```
1. () -> 7              // har ingen parameter og returer 7
2. x -> 2 * x           // en parameter (x) og returnerer den
                        // doble verdien
3. (x, y) -> x – y      //tar to tall inn, returnerer
                        // differansen
4. (int x, int y) -> x + y // tar to int inn, returnerer
                           //  summen
5. (String s) -> System.out.print(s) // Tar en String s og
                                     //  printer den
```

Følgende er valgfritt når vi skriver *lambda-uttryk* :

- Type-deklarasjoner på parametre (kompilatoren finner det ut)
- Parentes rundt parametre hvis vi har bare én parameter.
- Krøll-parenteser {} rundt høyresida hvis det bare har en setning.
- Bruke av *return*-ordet i høyresida. Kompilatoren returnerer bare siste verdi beregnet, men utelater vi *return* må vi ha krøll-parenteser.

13.3 Hvordan lages lambda-uttrykk

Når vi lager et *lambda-uttrykk,* må vi ha et grensesnitt med en metode som har samme antall og typer på parameterne som det vi trenger i vårt *lambda-uttrykk.* Hvis du enda ikke har lest kapittelet om grensesnitt, så er *grensesnitt* enkelt forklart en java – konstruksjon som ser ut som en meget forenklet klasse-deklarasjon. Et grensnitt begynner med ordet *interface* (i steden for class) og inneholder bare

spesifikasjon av metoder (deres navn, parametre med typer og metodene returverdi, men *ikke* kode inne i metodene). Her er tre eksempler du selv kan skrive i ditt program:

```
interface  Calc{
      double beregn (int x);
}
interface  Hei{
      void si (String s);
}

interface  Matte{
      int oper (int x, int y);
}
```

Merk at grensesnitt for å lage *lambda-uttrykk,* kan bare har én metode. (Java-bibliotekene inneholder også mange slike grensesnitt som kan brukes til å lage *lambda-uttrykk,* men du kan like godt deklarere selv de grensesnitt du trenger):

I programmet ditt kan du så lage flere ulike *lambda-uttrykk* fra samme metode i samme grensesnitt når de bare har samme antall og type av parametre og samme returverdi.

13.4 Hvordan bruke Lambda

Før vi kan bruke et *lambda-uttrykk* må vi altså koble det uttrykket vi vil bruke med et grensesnitt med samme antall og type parametre. Vi navngir da et *lambda-uttrykk* (egentlig navngir vi et objekt av en klasse som lager (implementerer) som den metoden som er i grensesnittet) slik :

```
Hei joa  = (s) -> System.out.println("Jeg sier:"+ s);
```

Og vi kan kjøre denne metoden slik (i en av våre vanlige metoder som main):

```
joa.si(" Lambda er bra");  // ut: Jeg sier: Lambda er bra
```

Her er er flere eksempler:

```
Calc areal = x -> x*x*3.14159/4;
Calc radius = x -> x*1.0/2;
Matte mult = (x,y) -> { return x*y;};
```

Og det kan brukes slik:

```
Scanner keyboard = new Scanner (System.in);
System.out.print ( "Gi et heltall: ");
num = keyboard.nextInt();

System.out.println(" Sirkel med diam:"+num+" m har
areal:"
 + areal.beregn(num)+" m2, og radius:"
 + radius.beregn(num)+"m");
System.out.println("3*7=" + mult.oper(3,7) );
joa.si(" - dette er bra");
```

Vi ser at grensenittet Calc har to ulike implementasjoner (areal og radius). Kjører vi programmet over får vi:

```
Gi et heltall: 137
Sirkel med diam:137 m har areal: 14741.1256775 m2, radius:68.5m
3*7=21
Jeg sier: - dette er bra
```

For å oppsummere: *lambda* gir ikke noe nytt, men du skriver mindre kode for å få det utført. I neste delkapittel ser vi hvordan *lambda-uttrykk* med brukes sammen med strømmer.

13.5 Strømmer (streams) fra mengder

Strømmer i Java er kort fortalt å kunne stille spørsmål om mengder av objekter som minner om SQL spørsmål fra database-verden. Et noe alternativt syn på strømmer fra Eyvind W. Axelsen. Den ligger sist i kap. 13.7.

Vi har en startmengde i Java (List, ArrayList, array, HashMap,..) som vi først omgjør til en strøm av enkeltobjekter og fra denne lager vi en ny mengde (svar-mengden), som er de fra startmengden som tilfredsstiller de kravene vi kommer med i den strømmen vi lager. Ta et eksempel fra programmet nedenfor om personer, deres navn, kjønn, inntekt osv. Vi kunne være interessert i å vite hvilke personer som tjener over 1 mill. kr. Da søker vi ut en ny mengde. Vil vi bare ha navnet til én person med så stor inntekt, søker vi etter en string. Vi kan også være interessert i bare å få vite hva gjennomsnittet er til de som tjener over 1 mill. kr. Vi søker da et tall. Vi kan også være interessert i finne kvinners inntekt.

Vi kan altså både søke ut en ny mengde, men også tall, Stringer ol.

```
import java.util.*;

class Person{
    String navn; String kjonn; int alder; int lonn;
    Person (String na, String kj, int ald, int ln) {
           navn = na; kjonn = kj; alder =ald; lonn = ln;
    }
 }

 public class StreamEks{
     ArrayList <Person> alle = new ArrayList <Person> ();

     public static void main (String [] args) {
           new StreamEks().doIt();
    }
    void doIt() {
       alle.add(new Person("Ola",  "M", 55,1200000));
       alle.add(new Person("Kari" ,"F", 44, 600000));
       alle.add(new Person("Jonas","M", 65, 110000));
       alle.add(new Person("Tora", "F", 12,  10000));
       alle.add(new Person("Arne", "M", 69, 2000000));

      // 1) skriv alle som tjener mer enn 1. mill
        alle.stream()
        .filter(s-> s.lonn > 1000000)
        .forEach(p->System.out.println(p.navn+
                      " tjener kr. "+ p.lonn+" per år"));

      // 2) Finn inntekt til person med størst inntekt
        int ml =
          alle.stream()
           .mapToInt(s -> s.lonn)
           .max()
           .getAsInt();

     // 3) finn gjennomsnittsinntekt for alle
          alle.stream()
          .mapToInt(p -> p.lonn)
          .average()
          .ifPresent(t ->
                System.out.println("snitt lønn= "+ t));
```

```
    // 4) Beregn snitt inntekt for de med lønn > 1.mill
          alle.stream()
            .mapToInt(p -> p.lonn)
            .filter (r -> r > 1000000)
            .average()
            .ifPresent(t -> System.out.println(
                "Snitt de over 1.mill = " +t+ " kr."));


    // 5) Beregn snitt kvinners lønn
           double kvinneLønn =
              alle.stream()
              .filter(p -> p.kjonn == "F")
              .mapToInt(p -> p.lonn)
              .average()
              .getAsDouble();

             System.out.println("Gjsnitt kvinnelønn er kr."+
                kvinneLønn+", max lønn er:"+ ml);

        // 6) Finn første kvinne med lønn < kr. 100 000
              alle.stream()
                 .filter(p -> p.kjonn == "F")
                .filter (r -> r.lonn < 100000)
                .findFirst()
                .ifPresent(t -> System.out.println(
                           "Lavlønnet kvinne er: "+t.navn));
     } // end doIt
  } // end StreamEks
```

Resultatet fra kjøring er:

```
Ola tjener kr. 1200000 per år
Arne tjener kr. 2000000 per år
snitt lønn= 784000.0
Snitt de over 1.mill = 1600000.0 kr.
Gjsnitt kvinnelønn er kr.305000.0, max lønn er:2000000
Lavlønnet kvinne er: Tora
```

Forklaring til alle eksemplene er at vi omgjør mengden vår først til en strøm (stream()) av enkelt-elementer, her Person-objekter. Videre i strømmen er det enten funksjoner som begrenser hvilke objekter som kommer videre (eks. filter()) , funksjoner som omgjør et objekt til en annen bestemt type (eks MapToInt()) og funksjoner som ber om alle (som max(), average(), sum(), forEach()) eller som bare ber om ett eksemplar (findFirst()). En slik strøm drives fremover ved det stadig bes om nye elementer, og hvis den funksjonen som er sist eller nest sist i har blitt tilfredstillet, som i eks 6 at vi finner første kvinne med lav lønn (under 100 000) stopper strømmen. Se dokumentasjonen til grensesnittene java.util.stream og java.util.collection i Java-dokumentasjonen.

Grunnen til at vi ikke behøver å skrive egne grensesnitt her er at de bibliotekene vi her bruker i java.util har deklarert det vi trenger med riktige typer på parametere og returverdier. Vi ser også at vi må prøve oss frem når vi tak et gjennomsnitt og må i Eks.3 konvertere det som 'flyter nedover strømmen' før vi kan ta average(), eller i Eks. 5 må vi også konvertere resultatet til en ekte double-verdi før vi kan gjøre tilordning til en enkel double-variabel.

Husk at vanlige arrayer ikke kan brukes direkte i en strøm , den må først konverteres til en strøm som her:

```
class GFG {
   public static void main(String[] args)
    {  // Converting int array to stream
        int arr[] = { 2, 1, 3, 4, 5 };
         IntStream stm = Arrays.stream(arr);
         stm.forEach(a -> System.out.print(a + " "));
   }
}
```

Og utskriften blir: **2 1 3 4 5.** Vi skal senere se hvordan vi kan sortere en slik strøm.

13.8 Parallelle strømmer

Her er en meget enkel metode som avgjør om parameteren er primtall eller ikke. Dette er ikke en effektiv metode (Eratosthenes sil er langt raskere) og er bare tatt med her som eksempel på en metode som bruker en del tid – den dividerer parameteren med 2 pluss alle oddetall < enn kvadratroten av parameterverdien. (Dette eksempelet har jeg fått av prof. Peter Sestoft, Teknisk, Universitet i København og omarbeidet litt):

```
private static boolean isPrime(int i) {
   if (i %2 == 0 ) return i == 2;
   // vi starter med å teste '3'
    int k=3;  long k2=9;
    while (k2 <= i && (i % k) != 0){
            k+=2;
            k2=k*k;
     }
     return  k2 > i;
} // end isPrime
```

Denne metoden kan vi bruke til å først lage en sekvensiell strøm som teller hvor mange primtall vi finner under en bestemt grense 'n':

```
int  antall=
        IntStream.range(2,n)
        .filter (i-> isPrime(i))
        .count();
```

så innfører vi bare en parallelliserings-kommando i strømmen:

```
int antall =
        IntStream.range(2, range)
        .parallel()
        .filter(i -> isPrime(i))
        .count();
```

Som Peter Sestoft sier i sin forelesning:" Pure functional programming ... ... and thus parallelizable and thread-safe".

13.9 Hvor rask er strømmer sammenlignet med 'vanlig' trådprogrammering

For enkle problemer hvor vi skal enten telle eller summere verdier i en mengde ferdige objekter som tilfredsstiller visse kriterier (som en strømbasert løsning på primtallene ovenfor), er strømmer en god og effektiv måte å løse ett problem. Også parallelt. Det må imidlertid presiseres at ikke alle problemer egner seg for en slik løsning.

Det ble derfor laget et program som sammenlignet fire ulike måter (sekvensiell strøm, parallell strøm, sekvensielt program og trådbasert parallelt program) for å regne ut antall primtall < n med isPrime(i). Vi kaller da isPrime(i) *for alle heltall* : 10<= i <=n. Øvre grense n var: 10, 100, 1000,…,10 mill.

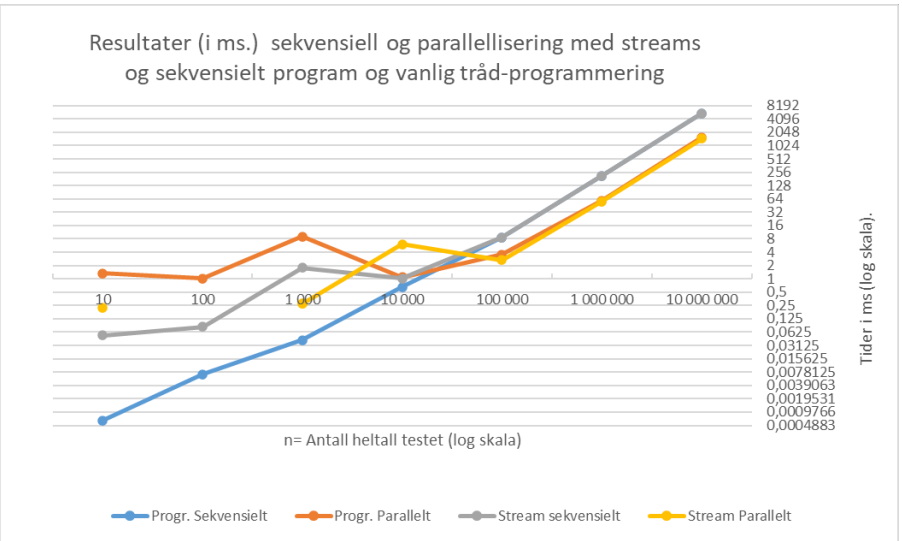


Fig 13.1 *Eksekveringstider for fire ulike måter å finn antall primtall < n (10 <= n <= 20 mill) med isPrime().Begge aksene er logaritmiske, dvs at hvert tall på x-aksen er n 10 ganger så stort som det forrige, mens verdiene på y-aksen er hver linje 2 ganger så stor som linjen under. Vi ser at sekvensielt program (blå)er raskest for alle n < 10 000, mens den trådbaserte parallelle programløsningen (rød) er langsomst i det samme området. I området for n > 10 000 blir kjøretiden for de to sekvensielle (program og strøm) like, og tilsvarende for de to parallelle løsningene. Merk at de to parallelle er begge der nesten 4 ganger så raske som de sekvensielle.*

Vi kan fra dette denne testing av *isPrime(),* at for korte kjøretider, dvs. mindre enn ca. 1 sekund er en vanlig sekvensiell løsning raskest, men for tider større enn ca. 4 sekunder er begge de to parallelle løsningene raskest. Det må også bemerkes art den parallelle strømmen var langt kortere og enklere å programmere enn den trådbaserte parallelle løsningen.

Det er imidlertid slik at strømmer ikke er like velegnet for å løse alle problemer – ikke alle algoritmer kan enkelt formuleres som filtrering av en strøm av objekter. Det er f.eks. vanskelig å tenke seg at en rekursiv algoritme som Kvikksort kan løses effektivt med strømmer. Man måtte vel da først gjøre alle tallene man skulle sortere til en liste av objekter med ett tall i hvert objekt, og så lese denne listen først én gang for å finne pivot-elementet; så lese den originale listen to ganger til, andre gang for å lage en liste av alle elementene med verdi <= pivot-objektet, så lese original-listen for tredje gang for å skjøte på den nye lista med de elementer som er større. Dette er tre ganger lesning av original-listen. I Kap 12 har vi sett hvor langsomt det kan være å operere på lister av slike Integer-objekter. For å fortsette rekursivt bør vi nok da oppbevare dette som to lister, den med elementer mindre enn 'pivot' og den med større enn pivot. Vi kan så fortsette med den 'lille listen på samme måten helt til den stadig mindre lille, venstre listen til sist har lengde =1. Dette høres å være vanskelig og lite effektivt. Viktigere er at primtalls-problemet løses med bare *ett* lite, vanlig sekvensielt program (isPrime(int p)). Det er ikke greit å tenke seg at hvordan man skulle skrive den rekursive logikken som

sekvensielle småprogrammer til de strømmene vi lager. Mulig er det kanskje, men det synes klart at dette vil bli langt mer komplisert og svært mye langsommere enn vanlig (sekvensiell eller parallell) Kvikksort. I java-biblioteket Arrays.sort er det egne sorteringsmetoder (for de ulike typene av basaltyper som byte, int, double,… osv) som også er tilpasset til også å kunne nyttes i en strøm.

```
List<Type> result = list
                    .stream()
                    .sorted(Comparator.comparing(Type::getValue))
                    .toList();
```

Program 13.7.1 *Enkleste måte å bruke en av de to innebygde sorteringsmetodene i Arrays.sort, for arrayer. Man lager en strøm av elementene i en array 'list', som så sorteres med Flette-sortering for lange arrayer og innstikk-sortering for kortere arrayer (Tim-sort), og som til sist omgjøres til en liste 'result' av de nå sorterte elementene.*

For å konkludere – strømmer gir oss en effektiv løsning på visse sekvensielle problemer. Men mer tyngre algoritmer hvor vi f.eks. rekursivt må finne en løsning som ved sortering eller 'den konvekse innhyllingen av en punktmengde', som er en obligatorisk oppgave i kurset, synes strømmer lite effektivt og bare mer komplisert.

13.8 En noe alternativ start på et kapittel om Strømmer i Java (fra Eyvind W.Axelsen)

Strømmer er rett og slett en «strøm» av objekter. En strøm har den viktige egenskapen at den er såkalt «lazy», noe som gjør at hvert objekt «evalueres» når det er bruk for det. Det vil si at en strøm godt kan være uendelig (eller uten kjent lengde). Det er nyttig når man f.eks. strømmer ting over nettet.

Et enkelt eksempel:

```
IntStream is = IntStream.iterate(0, i -> i + 1).filter(i -> i %2 == 0);
```

Her har vi en «uendelig» strøm (selv om vi i praksis er begrenset av størrelsen på en int), som så filtreres på alle som er delelige med 2. Om du legger inn dette i programmet ditt, så stopper det likevel ikke opp, fordi evalueringen er «lazy». Det skjer ingen ting før vi faktisk spør om verdier. Og når vi spør om verdier, så kan vi be om en og en, og ikke havne i en uendelig løkke selv om strømmen i seg selv er uendelig.

Dette poenget er viktig å få med i et kapittel som omhandler strømmer bør vel egentlig starte med et litt mer overordnet blikk på hva en strøm er, og hvorfor det er nyttig.

14 OPPSUMMERING – OM PARALLELLISERING AV ET PROBLEM

Det er mange måter å parallellisere et problem i en multikjerne-maskin, men de tre som er forklart her, bruk av synkroniserte metoder, barriere synkronisering samt ReentrantLock, skulle være tilstrekkelig for de aller fleste problemer. Særlig barrieresynkronisering må også med for større problemer.

1. For at det skal være vits å parallellisere et problem, må det utføre *mer* enn noen få operasjoner for hvert dataelement. Som en huskeregel kan vi si at hvis den største versjonen av problemet vi greier å kjøre sekvensielt tar mindre tid enn 1 sekund, er det ingen vits i å parallellisere det.
2. Start med en godt testet og effektiv **sekvensiell** løsning av problemet.
3. Se etter om man kan dele ***data*** opp i et antall om lag like store deler, hvor helst hver del kan løses etter den sekvensielle metoden i hver sin tråd – altså i parallell.
4. Hvis/når vi under beregningene må ha felles data, må de alltid beskyttes. All skriving og lesing på disse felles data skjer med synkroniserte metoder.
5. Hvis ***alle data alltid*** er felles, er det ingen vits i å parallellisere problemet hvis ikke trådene hver kan ha kopier av relevante felles data og at disse til sist greit kan samstilles etter beregningene.
6. Vi kan godt uten beskyttelse la ulike tråder i parallell skrive på ***ulike elementer*** i en array (men ikke i ulike bit i samme byte)
7. Når hver tråd har løst sin del, og etter at alle har ventet på ***en felles barriere*** når de er ferdige, så kan alle trådene etterpå lese resultatene av hverandres beregninger.
8. Kanskje er vi nå enten ferdig, eller resultatene fra første beregninger kan igjen deles opp i flere tråder med en ny barriere., osv.
9. Tenk spesielt på hvordan main-tråden skal vente og slippe løs når alle trådene er ferdige og har løst problemet. Det kan godt være en barriere som venter på antall tråder +1, som er main-tråden, som main legger seg og venter på når alle trådene er startet.
10. For noen klasser av problemer lønner det seg å sette i gang flere tråder enn man har ***k*** kjerner (inkludert hyperthreaded kjerner) i maskinen – f.eks 2k,3k eller 4k tråder. Bare test om dette evt. går fortere. Grunnen til en hastighetsøkning her er at de deler av data vi behandler i en tråd passer bedre i cache-systemet. Ulempene ved å få oftere bytte av kjerner og mer synkronisering oppveies da av hastigheten av cache-systemet.

To typiske parallelliseringer med *k* kjerner og *k* tråder:

A. De sentrale data i problemet lar seg enkelt dele i k like deler hvor vi kan la den sekvensielle algoritmen løse hver sin 1/k del av problemet med hver sin tråd i parallell Hvis problemet er så enkelt at man skal finne det størst elementet i en array, vil vi til sist etter at hver tråd har køet på en CyclicBarrier, så kan en av trådene (f.eks. tråd nr. 0) sammenligne de k svarene og rapportere den største av disse på skjermen eller fil.

B. Hvis den sekvensielle løsningen er rekursiv, er det en enkel men ikke helt optimal løsning å erstatte de få øverste rekursive kallene med at vi lager en tråd og for hver av disse. Det er viktig at der er toppen av rekursjonstreet (de øverste 2 til 4 lagene hvor de rekursive kallene er erstattet av en parallell tråd). Problemet med denne løsningen er at den første oppdelingen i toppen med to rekursive kall blir sekvensiell. Det er først på neste lag at vi kan få til 2-parallell, så 4-parallell løsning på neste nivå osv. Det er for noen rekursive algoritmer som Kvikksort mulig å endre disse slik at vi kan starte med full parallellitet med alle trådene [1] [A full parallel Quicksort algorithm for multicore processors] .
For andre rekursive algoritmer som Flettesortering er det også mulig å heve parallelliteten til 2-parallell , så 4-parallell,…, på det øverste laget i rekursiviteten ved at de kortere sorterte delene flettes både fra starten for å finne de minste elementene, og samtidig fra den andre enden endene flettes for å finne de største elementene i parallell
[2] [A faster, all parallel Merge sort algorithm for multicore processors].

OPPGAVER

1. Lag den sekvensielle versjonen av Kvikksort, modifiser utførOgTest() for sorteringseksempelet slik at du lager en array med samme innhold som har blitt sortert av sKvikk og sorter den så med Arrays.sort() fra Java-biblioteketS Skriv ut tiden for denne og sammenlign med tiden for kvikksorttiden.
2. I den parallelle versjonen av Kvikksort, pKvikk, fjern de rekursive kallene og løs problemet bare med tråder. Kommenter kjøretidene og antall tråder du får. Er dette lurt?
3. Til tross for advarselen i kap. 13.7, prøv å lage en strømbasert løsning på Kvikksort og sammenlign kjøretiden til denne med Arrays.sort() fra Java-biblioteket.
4. Skriv et program for 'En Selgers rundtur'; først sekvensielt (rekursiv) og så en blanding av parallellt og rekursivt. Dette er en oppgave med svært mange beregninger og svært lite data, og egner seg svært godt for parallellisering.
   Problemet er slik: En selger skal reise og besøke n byer – hver by skal besøkes bare *én* gang. Hun vet avstandene fra enhver by til alle de andre byene. Disse dataene har hun i en todimensjonal array: **`avstand[][]`**, slik at avstand [i][j] er avstanden fra by i til by j. Om avstandene vet du også at **`avstand[i][j]= avstand[j][i]`**, at **`avstand[i][i]`** = **0**, og at det alltid er raskest å reise direkte til en by – det går aldri noen snarvei ved å reise via en annen by.
   Skriv ut den korteste reiseplan av alle mulige hvor selgeren besøker hver by bare en gang og som kommer til slutt tilbake til utgangsbyen.

```
class SeqRSelger {
  int [][] avstand ;
  int [] x,y;
  int bestHittil = Integer.MAX_VALUE;
  int [] besteRute, reiseRute;
  boolean [] besøkt;
  int n;
  void  RSR (int nivå, int by, int lengde) {
      if( nivå == n) { // gjenstår bare reisen tilbake til by:0
            if (lengde+avstand[by][0] < bestHittil){
                  <notér ny beste reisevei og lengde>
            }
      } else {
          for (int nesteBy = 1; nesteBy < n ; nesteBy++) {
             // prøv å besøke alle ikke-besøkte byer unntatt 0
             if (! besøkt [nesteBy]) {
                besøkt[nesteBy] = true;
                reiseRute[nivå] = nesteBy;
                RSR (nivå+1,nesteBy,lengde +avstand[by][nesteBy] );
                besøkt[nesteBy] = false;
    }}}} // end PSR
    SeqRSelger(int n) {
          <opprett og initier arrrayer med tildeldige tall>
          <beregn avstandsmatrisen med Pytagoras>
     } // end konstruktor
     void utfør() {
          < ta starttidspunkt>
           RSR(1,0,0);
           <skriv ut data, tid brukt på PSR() og beste reisevei>;
     }
      public static void main (String [] args) {
           new SeqRSelger(Integer.parseInt(args[0])).utfør();
      }
} //end SeqRSelger
```

Programskisse til oppgave 4. *Selgerens rundreise. Dette er den sekvensielle varianten av problemet. Få denne til å virke og lag så en parallell versjon. Dette er på ingen måte den*

*beste algoritmen som løser dette problemet, men nok den korteste. Merk hvor raskt den løser et 10-bys problem, men at det tar alt for lang tid å løse et 15-byes problem. Kjøretiden går som n! =1*2*..*(n-1)*n. Vi ser at kjøretiden da øker ekstremt raskt med n.*

5. Lag en versjon av parallell Kvikksort hvor du bare bruker én sykliske barriere. Hint: Ser du på kall-treet ser du at det er ett kall på **pKvikk** på nivå 1 (toppen) i treet, samlet 3 kall på nivå 2 og nivå1 i treet, samlet 7 kall til og med nivå 3 osv. (formelen for antall kall med k nivåer er: 2^k-1). Du må da før du starter kallene regne ut hvor mange nivåer du vil ha i kall-treet for å starte nye tråder, og sette opp den sykliske barrieren til å vente på så mange. Resten av problemet løser du som før rekursivt. Er denne raskere enn den som står i avsnitt 7.1?

REFERANSER:

Siden UiO nå har fjernet alle *direkte* referanser til de ansattes arbeider i et åpent journalsystem vi nyttet før 24. februar 2024, er det enklest å finne disse to artiklene (den første på NIK2015, den andre på NIK2018) via https-adressen:

[redacted]

Trykk først på artikkel-navnet, og så på [full-text available], så får du artiklene nedlastet i pdf.

[1] [redacted]

[2]

htt

htt

NYERE PROSESSORER:

1 -Amazone Graviton : [redacted]

Sketch about synchronization

Eric Jul and Michael Kirkedal Thomsen

IN3030/IN4330, Spring 2026

Manuscript synchronization sketch IN3030 lecture 11/3-2026 02:15pm at University of Oslo, Small Auditorium, Kristen Nygaards Hus.

1. Michael enters the room several minutes before the start of the lecture – Michael should stand near the blackboard that is closest to the windows. Eric enters hurriedly at 14:15:05 and heads for the blackboard to the left as seen from the students.

2. As soon as Eric arrives at the blackboard, both starts writing on each their blackboards “IN3030 Today: Synchronization” and bids welcome to the lecture.

3. Discover each other:

> Eric: "Hi, what are YOU doing here?"
> Michael: "I am doing today’s lecture!”
> Eric: "No, I am doing today’s lecture!”

4. Both pull out a version of the lecture plan:

> Michael (pulls out a printed schedule): ”No, I am doing today’s lecture, see this schedule!”
> Eric (pulls out a printed schedule): ”No, I am doing today’s lecture, see THIS schedule!”
> Michael: ”Mine is the REVISED schedule!” (points to the word REVISED)
> Eric: ”Mine also says REVISED!” (points to the word REVISED)

5. EJ: "What do we do about this?”

6. Short pause, then: both *in unison* (triggered by Eric sweeping his hand across toward Michael): "After you!!!" (repeat 3 times, *i.e.*, a total of 4 times).

7. Eric: "This is not working out!! How are we to decide who should do the lecture?? I must be able to devise a method for us to decide who should lecture – after all, I AM a PROFESSOR!” (Eric scratches his head…) – ”… hey, I have a great idea! A FLAWLESS system that will ensure that there is only ONE lecturer!”

8. Eric: Sets up an opaque bowl and explains: “Here is how we ensure that there is no more than one lecturer: When entering, a lecturer goes over to the bowl and check whether or not the bowl is empty. If empty, the lecturer then finds a piece of chalk and puts it into the bowl, indicating that there now *is* a lecturer. If NOT empty then the lecturer knows that there ALREADY is another lecturer, and thus can leave. This FLAWLESS system thus *ensures* that there is ONLY one lecturer!”

9. Eric then walks over to the bowl, checks that it is empty then moves away gets a piece of chalk. He then starts bragging about his FLAWLESS system to ensure that only one will be lecturing while waving the piece of chalk in his hand. As Eric is walking away from the bowl, Michael resolutely and quickly moves over to the bowl, checks it (flip it upside-down), then swiftly finds a piece of chalk, dumps it into the bowl and then moves to the blackboard away from the door and starts writing the agenda on the board.

10. When Eric is done bragging about his FLAWLESS system, he goes to the bowl and drops the piece of chalk into the bowl and then goes back to the board. BOTH start lecturing – saying welcome and that today, we will talk about *synchronization* – BOTH stop when discovering that there are **TWO** lecturers.

11. Small discussion: Michael (sarcastic, shaking head): “FLAWLESS system indeed!”.

12. Eric (again scratching his head): “Let me think … Ah! Now I know: I need TWO bowls. And now we start by putting chalk into one’s OWN bowl and then checking the OTHER bowl. Eric: "This method ensures that there is AT MOST ONE LECTURER!”

13. Both now go to the blackboard, find a piece of chalk, put it into one’s own bowl, then goes over to the other bowl to find chalk in it. Then both points to the other and says: “Ah, it is YOU that will lecture today!” and then both pack up and leave.

14. Eric returns and starts the actual lecturing.

Assignment: (Think carefully about your answer – and explain, *e.g.*, with examples):

- **What went wrong in point 4, 6, 9-10, and 12? Suggest a fix for the problem in 6.**
- **Is the statement at the end of point 8 correct?**
- **Is the statement at the end of point 12 correct? Why is it not sufficient?**



IN3030 L09, spring 2026
Synchronization
2026-03-18

Eric Jul and Paulo Ferreira
PT
Department of Computer Science
University of Oslo

1

1

Review L08v26

- Convex hull

2

2

Plan for L09

1. Chefs and waiters
2. 3 solutions
3. Hoare Monitors
4. Andes trains.

3

3

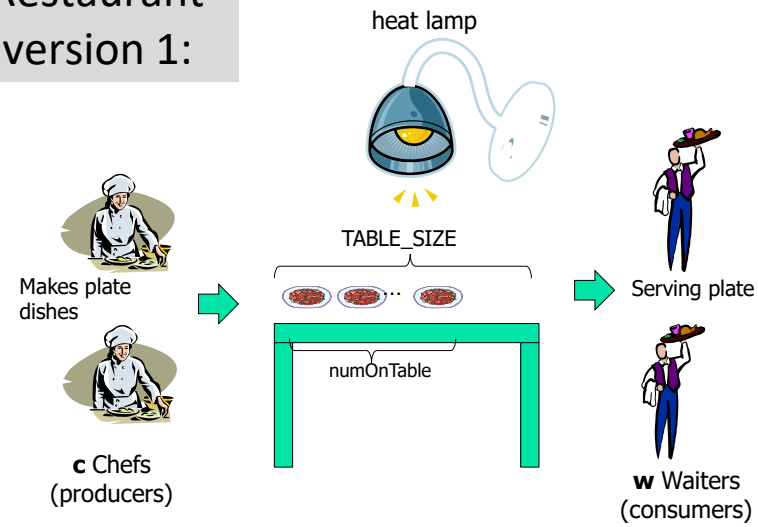
Chefs and Waiters

- The problem we are now going to solve.
- A restaurant with chefs and waiters and a warming table where the food is placed.
- We have **c** Chefs who cook and **w** Waiters serving the food (plate dishes).
- Food that the chefs prepare is placed on a **warming table** (with TABLE_SIZE number of places for plates).
- The chefs cannot make more plates if the hot table is full.
- The waiters cannot serve more plates if the hot plate is empty.
- NUM_TO_BE_MADE number of plates.

4

4

Restaurant version 1:



5

5

Chefs and Waiters

- About monitors and queues (three examples of concurrent programming.)
- We solve synchronization between two different classes.
- First, an actively polling (busy) solution with synchronized methods (Restaurant1.java).
  - Actively waiting in a loop in each Chef and Waiter
    + that they are in the queue to be allowed into a synchronized method
- So a solution using a monitor as it was intended from the start in Java (Restaurant2.java).
  - Chefs and Waiters wait in the same wait () queue
    + in the queue to be allowed into a synchronized method.
- Finally, a solution with a monitor with Lock and Condition queues (several queues – one per waiting condition (Restaurant9.java)
  - Waiters and Chefs wait in their own queue
    + in a queue to be let into the two methods protected by a Lock

6

6

Common to the three solutions

```
import java.util.concurrent.locks .*;
class Restaurant {

Restaurant(String[] args ) {
<Entering the number of Chefs, Waiters and number of
dishes>
<Creates the Chefs and Waiters and starts them>
}

    public static void main(String[] args ) {
          new Restaurant( args );
    }
  } // end main
} // end class Restaurant

class HeatingTable { // MONITOR
     int numOnTable = 0,
         numProduced = 0,
         numServed =0;
     final int MAX_ON_TABLE = 3 ;
     final int NUM_TO_BE_MADE;
     // Invariants:
// 0 <= numOnTable <= MAX_ON_TABLE
// numProduced <= NUM_TO_BE_MADE
// numServed <= NUM_TO_BE_MADE

< + different data in the three examples>
```

```
public xxx boolean putPlate (Chef c)
   <Put another plate on the table
   (true) otherwise (false) Chef must wait)
} // end put

public xxx boolean getPlate (Waiter w) {
<If the table is empty (false) Waiter waits
otherwise (true) - Waiter then takes a
plate and serves it>
   } // end get
} // end class HeatingTable

class Chef extends Thread {
    HeatingTable tab;
    int ind ;
    public void run() {
         while / if ( tab .putPlate (..))
         <Different logic in the examples>
    }
} // end class Chef

class Waiter extends Thread {
    HeatingTable tab;
    int ind ;
    public void run() {
          while / if ( tab. getPlate ()){
         <different logic in the examples>
    }
} // end class Waiter
```

7

Invariants on common variables

- The invariants must **always hold** (be true) outside of the monitor methods.
- What are the common variables here:
  - MAX_ON_THE_TABLE
  - NUM_TO_BE_MADE
  - numOnTable
  - numProduced
  - numServed = numProduced – numOnTable
- Invariants:
  1. **0 ≤ numOnTable ≤ TABLE_SIZE**
  2. **numProduced ≤ NUM_TO_BE_MADE**
  3. **numServed ≤ numProduced**

8

8

The invariants shows 4 states we must take write code for

Invariants :

0 <= numOnTable <= MAX_ON_TABLE

numServed <= numProduced <= NUM_TO_BE_MADE

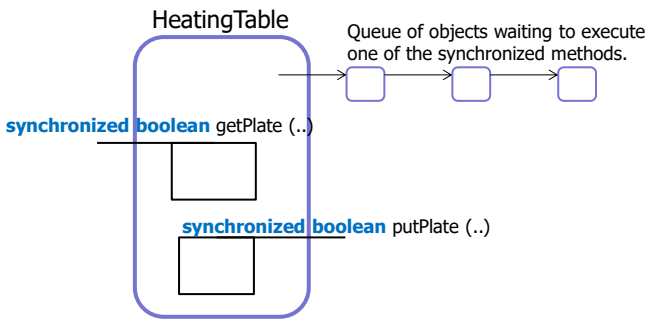
1. **numOnTable == MAX_ON_TABLE**
   → **Chefs are waiting**
2. **0 == numOnTable**
   → **Waiters are waiting**
3. **numProduced == NUM_TO_BE_MADE**
   → **The chefs finished**
4. **numServed == NUM_TO_BE_MADE**
   → **The waiters finished**

9

9

First, an actively polling (busy) solution with synchronized methods (Restaurant1.java).

- This is a solution with **a queue** , the one that all threads enter if another thread is inside a synchronized method in the same object.
- Termination is arranged in each chef and waiter (in their run method)
- The queue used is a common queue of all active objects that may simultaneously try to call one of the two synchronized methods. **goat** and **put .** All objects have such a queue.



10

10

Restaurant solution 1

```
class Chef extends Thread {
....
    public void run() {
       try {
          while ( tab.numProduced < tab.NUM_TO_BE_MADE ) {
            if ( tab.putPlate ( this ) ) {
              // create the next plate
            }
            sleep ((long) (1000 * Math.random ()));
          }
       } catch ( InterruptedException e ) {}
      System.out.println("Cook "+ ind +" done: " );
    }
} // end Chef

class Waiter extends Thread {
.............
    public void run() {
          try {
            while ( tab.numServed < tab.NUM_TO_BE_MADE ) {
            if ( tab.getPlate ( this )) {// serve plate}
            sleep ((long) (1000 * Math.random ()));
            }
          } catch ( InterruptedException e ) {}
          System.out.println("Waiter " + ind +" finished");
    }
} // end Waiter
```

```
synchronized boolean  putPlate (Chef c) {
     if (numOnTable == TABLE_SIZE ) {
        return false;
     }
     numProduced ++;
     // 0 <= numOnTable < TABLE_SIZE
     numOnTable ++;
     // 0 < numOnTable <= TABLE_SIZE
     System.out.println("Cook no :"+ c.ind +",
     made plate no :"+ numProduced );
     return true ;
  } // end putPlate

synchronized boolean  getPlate (Waiter w) {
     if (numOnTable == 0) return false ;
     // 0 < numOnTable <= TABLE_SIZE
     numServed ++;
     numOnTable --;
     // 0 <= numOnTable < TABLE_SIZE
     System.out.println("Waiter no :"+ w.ind +
     ", served plate no :"+ numServed );
     return true ;
}
```

11

11

```
M:Restaurant1>java Restaurant1 11 8 8
Chef no:8, made dish no:1
Chef no:4, made dish no:2
Chef no:6, made dish no:3
Waiter no:5, served plate no:1
Chef no:3, made dish no:4
Waiter no:2, served plate no:2
Chef no:1, made dish no:5
Waiter no:5, served plate no:3
Waiter no:4, served plate no:4
Chef no:7, made dish no:6
Chef no:2, made dish no:7
Waiter no:7, served plate no:5
Chef no:4, made dish no:8
Waiter no:3, served plate no:6
Waiter no:3, served plate no:7
Chef no:1, made dish no:9
Waiter no:2, served plate no:8
Chef no:6, made dish no:10
Chef no:3, made dish no:11
Cook 8 done:
```

```
Waiter no:8, served plate no:9
Waiter no:7, served plate no:10
Waiter no:6, served plate no:11
Cook 3 done:
Cook 5 done:
Waiter 1 finished
Cook 1 done:
Cook 4 done:
Waiter 5 finished
Cook 7 finished:
Waiter 2 finished
Cook 2 finished:
Waiter 4 finished
Waiter 3 finished
Waiter 7 finished
Waiter 6 finished
Cook 6 done:
Waiter 8 finished
```

12

12

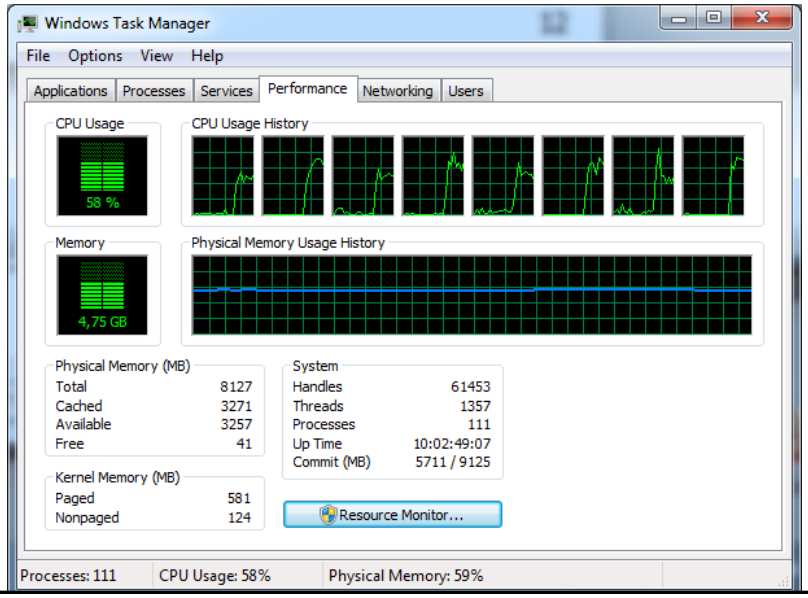
Problems with this solution are active polling

- All the Chef and Waiter threads are actively going around asking:
  - Is there more work for me? Rest a little, about 1 second, and ask again.
  - Wastes a lot of time/machine instructions.
- Especially stressful if one of the thread types (Producer or Consumer) is clearly faster than the other,
  - E.g. set up 18 fast Chefs who sleep only 1 millisecond against 2 slow Waiters who sleep 1000 ms.
  - In that case, this active waiting/masing took 58% of the CPU capacity of 8 cores
- The print must be in the get and put methods. Why?

13

13

Solution 1 with 18 fast Chefs (waiting 1 ms) and 2 slow Waiters (waiting 1000 ms). The Chefs stress the machine with constant failed queries every ms. whether there is room for a plate on the warming table. CPU usage = 58%.



14

14

Solution 2: Java's original scheme with monitors and **two queues** .

- The original Java solution with synchronized methods and a number of other methods and the following built-in methods:
  - **sleep (t):**
    - The currently running thread sleeps for 't' milliseconds.
  - **notify ():**
    - Inherited from the class Object of which all are subclasses.
    - It wakes up one of the threads waiting for the lock on the current object. It then tries again to get what it was waiting for.
  - **notifyAll ():**
    - Inherited from class Object.
    - It wakes up all threads waiting for the lock on the current object. They then all try once to get what they were waiting for.
  - **wait ():**
    - Inherited from class Object.
    - Causes the current thread to wait until it is either woken up by notify () or notifyAll () for this object.

15

15

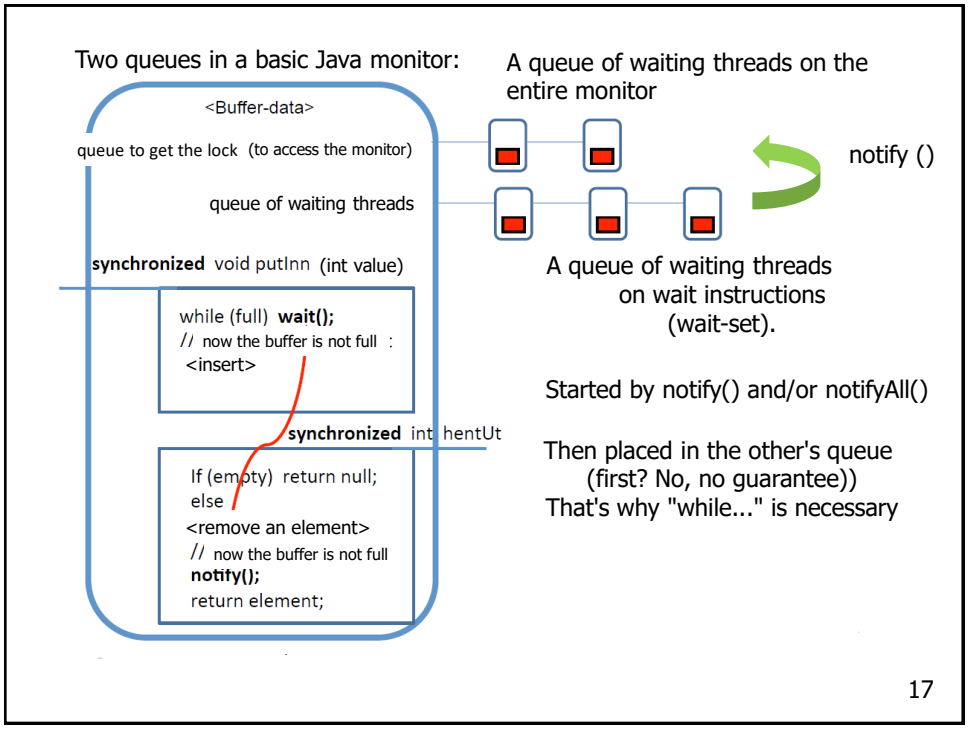
Creating parallel solutions with a Java 'monitor'

- A Java monitor is an object of any class with synchronized methods.
- There are two queues on such an object:
  - A queue for those waiting to enter/continue in a synchronized method
  - A queue for those who said **wait ()** here (and who are waiting for some other thread to wake them up by saying notify () or notifyAll () on them)
    - wait () calls a thread inside a synchronized method.
    - notify () or notifyAll () is also called inside a synchronized method.

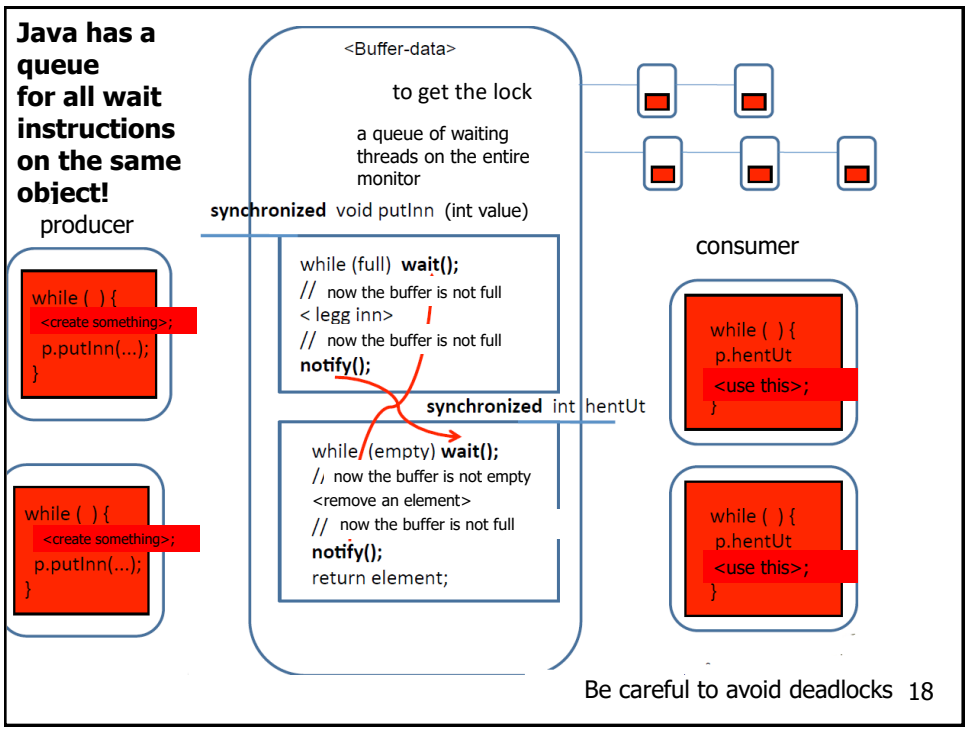
The Monitor idea is heavily inspired by Tony Hoare (the man behind Quicksort)

16

16

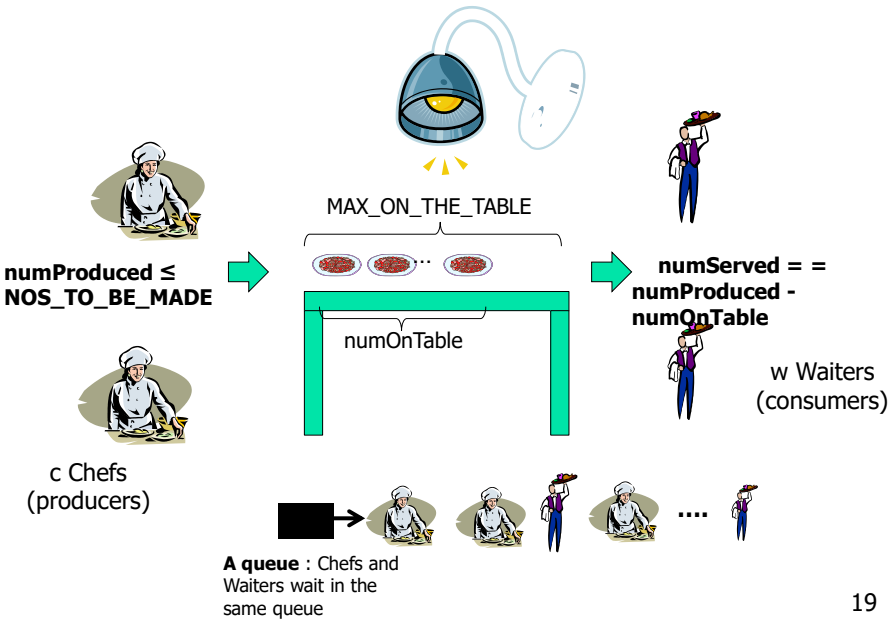

Two queues in a basic Java monitor:
<Buffer-data>
queue to get the lock (to access the monitor)
queue of waiting threads
synchronized void putInn (int value)
while (full) wait();
// now the buffer is not full :
<insert>
synchronized int hentUt
If (empty) return null;
else
<remove an element>
// now the buffer is not full
notify();
return element;
A queue of waiting threads on the entire monitor
notify ()
A queue of waiting threads on wait instructions (wait-set).
Started by notify() and/or notifyAll()
Then placed in the other's queue (first? No, no guarantee))
That's why "while..." is necessary
17

17


Java has a queue for all wait instructions on the same object!
producer
while ( ) {
<create something>;
p.putInn(...);
}
while ( ) {
<create something>;
p.putInn(...);
}
<Buffer-data>
to get the lock
a queue of waiting threads on the entire monitor
synchronized void putInn (int value)
while (full) wait();
// now the buffer is not full
< legg inn>
// now the buffer is not full
notify();
synchronized int hentUt
while (empty) wait();
// now the buffer is not empty
<remove an element>
// now the buffer is not full
notify();
return element;
consumer
while ( ) {
p.hentUt
<use this>;
}
while ( ) {
p.hentUt
<use this>;
}
Be careful to avoid deadlocks
18

18

The restaurant (2):



19

19

Second Solution

- All waiting is inside synchronized methods in one of two queues.
- All waiting inside two synchronized methods
- Chefs and Waiters waiting for the next plate in the wait queue
- We have to wake up everyone in the wait queue to make sure we find someone of the type we need (Chef or Waiter) who can run the program.
- No testing on the invariants in the run methods

20

20

Both solutions 2) and 3):
The run methods try again if the last operation succeeds:

Chefs :

```
public void run() {
      try {
         while ( tab.putPlate ( this )) {
         sleep((long) (1000 * Math.random()));
         }
      } catch (InterruptedException e) {}
      // This Chef is finished
}
```

Waiters:

```
public void run() {
      try {
         while ( tab.getPlate ( this )){
         sleep((long) (1000 * Math.random()));
         }
      } catch (InterruptedException e) {}
      // This Waiter is finished
}
```

21

Solution2

```
public synchronized boolean putPlate ( Chef c) {
    while ( numOnTable == TABLE_SIZE &&
            numProduced < NUM_TO_BE_MADE) {
        try { // The while test holds here meaning that a Chef should
            // but can't make a dish, because the table is full
        wait ();
        } catch (InterruptedException e) { // Insert code to handle interrupt }
    }
    // one or both of the loop conditions are now false
    if ( numProduced < NUM_TO_BE_MADE) {
        // numOnTable < TABLE_SIZE
        // Hence OK to increase numOnTable
        numOnTable ++;
        // numProduced < NUM_TO_BE MADE
        // Hence OK to increase numProduced :
        numProduced ++;
        // numOnTable > 0 , Wake up a waiting
        // waiter, or all if
        // numProduced ==NUM_TO_BE_MADE
        notifyAll () ; // Wake up all waiting
    if ( numProduced ==
        NUM_TO_BE_MADE) {
     return faLSE;
    } else { return true ; }
} else {
    // numProduced ==
    // NUM_TO_BE_MADE
    return false ;}
} // end putPlate
```

22

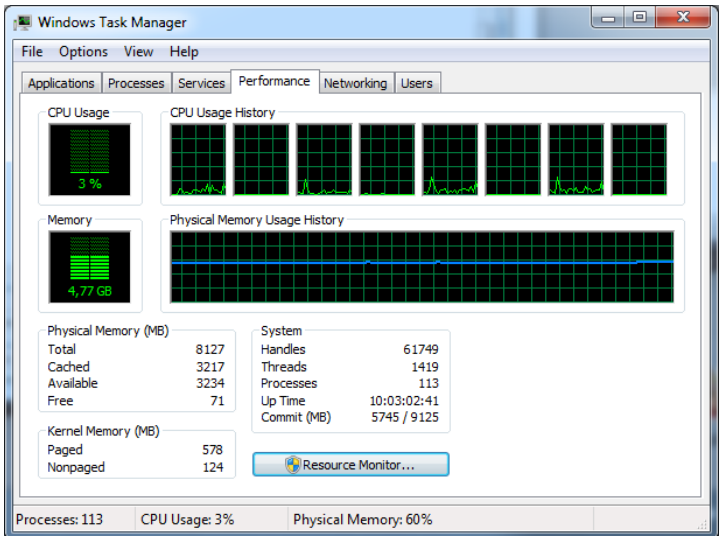
Solution2

```
public synchronized boolean getPlate ( Waiter w) {
    while (numOnTable == 0 && numProduced < NUM_TO_BE_MADE ) {
        try { // The while test holds here the meaning that the table
              // is empty and there is no more to serve
            wait ();
        } catch (InterruptedException e) { // Insert code to handle interrupt }
    }
    //one or both of the loop conditions are now false
    if ( numOnTable > 0) {
        // 0 < numOnTable <= TABLE_SIZE
        // Hence OK to decrease numOnTable :
        numOnTable --;
        // numOnTable < TABLE_SIZE
        // Must wake up a sleeping Cooks :
        notifyAll (); // wake up all queued Waiters and Chefs
        if ( numProduced == NUM_TO_BE_MADE && numOnTable == 0 ) {
            return false;
        } else { return true;}
    } else { // numOnTable == 0 && numProduced == NUM_TO_BE_MADE
        return false ;}
} // end getPlate
```

23

23

Solution 2 with 18 fast Chefs (waiting 1 ms) and 2 slow Waiters (waiting 1000 ms). The Chefs do not stress the machine with constant failed queries, but wait in line until there is room for another plate on the warming table. CPU usage = 3%.



24

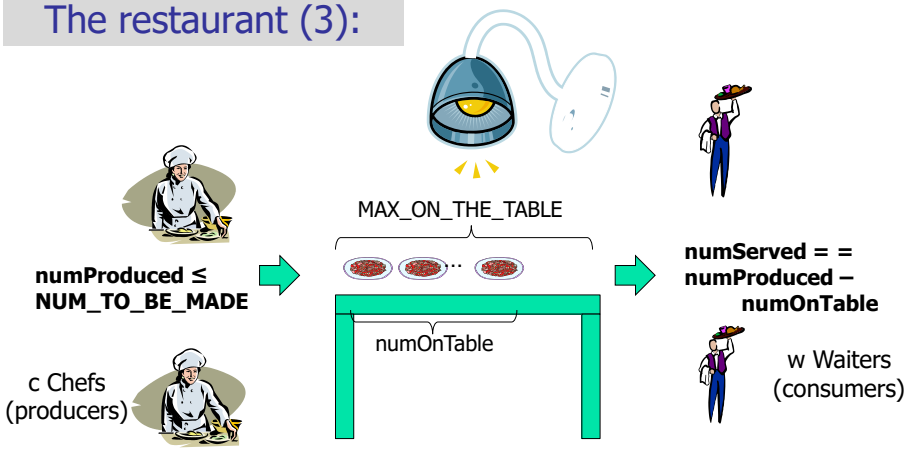
24

3) A parallel solution with Conditions

- Using two queues:
  - One for Chefs waiting for a plate space at the table
  - One for Waiters waiting for a plate
- Then we don't need to wake up all the threads, just one in the right queue.
  - Maybe more efficient
  - Clearly easier to program

25

The restaurant (3):



Two queues:
a) Chefs waiting for a place at the table

b) Waiters are waiting for more plates



26

```
final Lock lock = new ReentrantLock();
final Condition notFull = lock.newCondition(); // queue for Chefs
final Condition notEmpty = lock.newCondition(); // queue for Waiters

public boolean putPlate (Cooks c ) throws InterruptedException {
    lock.lock ();
    try {while(numOnTable==MAX_ON_TABLE && numProduced<NUM_TO_BE_MADE ){
        notFull.await (); // waiting for a place on the table
    }
        if (numProduced < NUM_TO_BE_MADE) {
            numProduced ++;
            numOnTable ++;;
            notEmpty.signal (); // Wake up a waiting waiter to serve
            if (numProduced == NUM_TO_BE_MADE ) {
                // I have produced the last plate,
                notEmpty.signalAll (); // tell Waiters to stop waiting, terminate
                notFull.signalAll (); // tell Chefs to stop waiting and terminate
                return false ;
            }
            return true ;
        } else { return false ;}
    } finally {
        lock.unlock();
} } // end putPlate
```

27

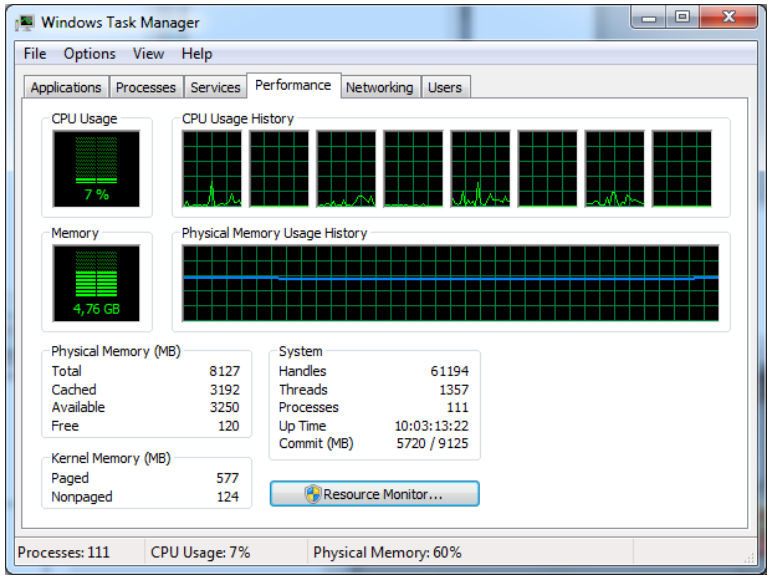
```
public boolean getPlate ( Waiter ) throws InterruptedException {
  lock.lock();
  try {
    while (numOnTable == 0 && numProduced < NUM_TO_BE_MADE ) {
        notEmpty.await (); // This Waiter is waiting for a plate
    }
    if (numOnTable > 0) {
        numOnTable-- ;
        notFull.signal (); // Signal to one Chef in the Chef's waiting queue
        return true;
    } else {
        return false;}
   } finally {
      lock.unlock ();
  }
 } // end getPlate
```

A Waiter or a Chef is signaled for two reasons:
- to treat (make or serve) another dish
- nothing more to do, go home (clear both queues)

28

28

Solution 3 with 18 fast Chefs (waiting 1 ms) and 2 slow Waiters (waiting 1000 ms). The Chefs do not stress the machine with constant failed queries, but wait in line until there is room for another plate on the warming table. CPU usage = 7%.



29

29

Assessment of the three solutions

- **Solution 1** : Simple, but can take up too much CPU time. Especially when the system is getting saturated for other reasons, typically one of the threads will become very slow, and then this solution suddenly takes up half of the CPU time.
- **Solution 2:** Good, but difficult to write
- **Solution 3:** Good, almost as effective as solution 2 and easier to write.

30

30

Concluding remarks on the Producer-Consumer problem

- Invariants are used by all programmers (often unconsciously)
  - program, loop or method (sequential or parallel)
  - saying them explicitly helps with programming
- REMEMBER: synchronized/lock only works when all threads synchronize on the same object.
  - when that happens, there is **sequential thinking** between wait /signal
- Debugging by tracing execution (trace) – System.out.println("..")
  - writing outside a Lock/synchronized method/part of method, then create one:
    - synchronized void println (String s) {System.out.println(s);}
  - otherwise, printing may be mixed up or out of order.

31

31

Problem

- When we say notify () or wait () on a queue, we don't know:
  - which thread starts
  - does the thread that is signaled get on the kernel directly after the one that said notify (), or not??
  - Not defined
- Even after we have tested in the run method that we can handle another plate, we still need to test if it works OK:
  - Another thread may have been inside and changed the variable

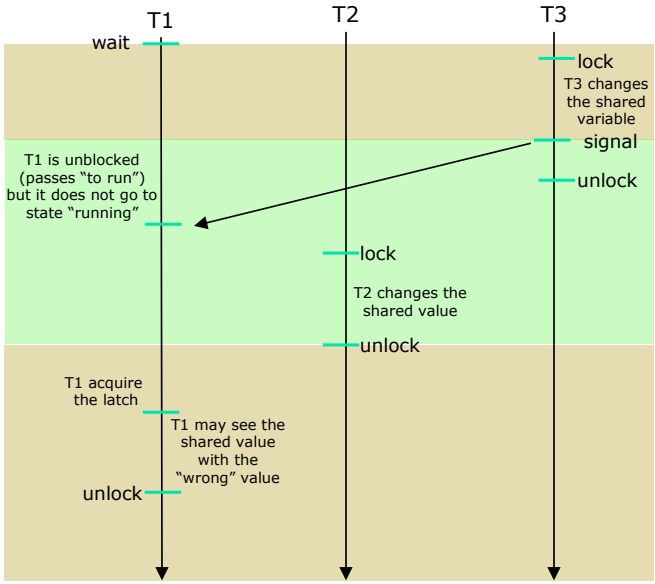
32

32

Problem

- Attention:
  - During the time that passes between the signal (T3) and a thread be unblocked (T1) acquiring the latch
  - The condition variable can be changed by another thread (T2)

Semantics Mesa (here is assumed) vs Semantics Hoare (see monitors)



33

Mesa and Hoare Semantics

- Most modern languages use **Mesa semantics**
- **Hoare monitor:** signal immediately transfers control to the waiting thread.
- **Mesa monitor:** signal only wakes the waiting thread, but the notifying/signalling thread keeps running.
- In **Mesa monitors**, when a thread calls notify()/signal():
  - The waiting thread is only moved to the ready queue.
  - The signalling thread continues executing.
  - Other threads may enter the monitor and modify the shared state before the waiting thread resumes.
  - So, when the waiting thread finally runs again, the condition might no longer be true.

34

34

Hoare Monitors

- The concept of Monitors was originally presented by Per Brinch Hansen who put the concept into his language Concurrent Pascal, in 1970`s
- Brinch Hansen's Monitors were refined and presented in a nice , clean version in his seminal paper : *Monitors: An Operating System Structuring Concept* published in 1974 .

- Monitors paper
- Andes train

35

35

End of lecture L09v25

36

36

IN3030/IN4330, L17, 2026
Yet More Synchronization

Eric Jul and Paulo Ferreira
Programming Technology Group
Programming Section
Department of Informatics
University of Oslo

1

Summary of Previous Class and Plan of this Lecture

- Synchronization
- Yet More Synchronization:
  - cooperation vs. competition
  - typical problems of synchronization (producer consumer, philosophers´ dinner, and barber shop)
  - condition variables
  - monitors
  - oblig5

2

Threads/procs Synchronization

- Several procs run in conjuction one or more threads, in which they indicate several events:
  - e.g., existence/not-existence of resources
- So, we have:
  - cooperation
  - competion

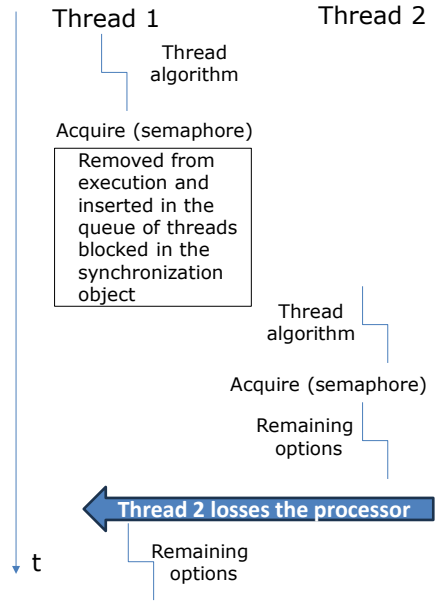
3

Cooperation

- When procs/threads are cooperating we want a mechanism that:
  - Independently of the state of the proc/thread and its own speed of execution, allows to signal that a certain event has already ocorred
  - E.g., thread/proc 2 wants to be informed when thread/proc 1 finished his work unit so that thread/proc 2 can proceed

4

Most Simple Situation of Cooperation



5

Competition

- A proc/thread demands a resource:
  - runs acquire (SemResource)
- A proc/thread frees a resource:
  - runs release (SemResource)
- The semaphore that controls the algorithm is initialized with the number of availabe resources:
  - SemResource = new Semaphore(NUM_RESOURCES)

6

Semaphores (example)

```
#define MAX_STACK 100
char*    stack[MAX_STACK];
int      top = MAX_STACK-1;
Semaphore SemMutEx;
Semaphore SemMem;

char *ask_for_mem() {
   acquire (SemMem);
   acquire (SemMutEx);
   ptr = stack[top];
   top--;
   release (SemMutEx);
   return ptr;
}

void get_back_mem() {
   acquire(SemMutEx);
   top++;
   stack[top] = ptr;
   release (SemMutEx);
   release (SemMem);
}
```

Memory Allocator with semaphore SemMem that controls the free memory !

```
main(){
/*...*/
  semExMut = new Semaphore(1);
  semMem = new Semaphore(MAX_STACK);
}
```

The semaphore is initialized with the number of available resources for the client

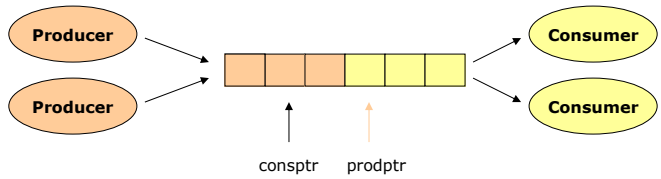
7

Typical Examples of Synchronization

- Producers/Consumers
  - Threads/procs that produce information to a buffer and threads/procs that read the information from that buffer (semaphores)
- Dining-Philosophers
  - Threads/procs that eat spaghetti and has less forks that eaters (semaphores)
- Barber
  - One or more threads/procs that serve the client thread (semaphores and conditions)

8

Example: Producers - Consumers



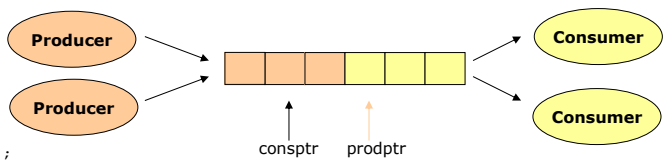

```
/* ProducerConsumer */
int buf[N];
int prodptr=0, consptr=0;

producer()
  while(TRUE) {
    int item = prod();
    buf[prodptr] = item;
    prodptr = (prodptr+1) % N;
  }
}
```

```
consumer(){
  while(TRUE) {
    int item;
    item = buf[consptr];
    consptr = (consptr+1) % N;
    cons(item);
  }
}
```

9

Example: Producers - Consumers




```
int buf[N];
int prodptr=0, consptr=0;
latch_t latch;
```

```
producer() {
  while(TRUE) {
    int item = prod();
    close(latch);
    buf[prodptr] = item;
    prodptr = (prodptr+1) % N;
    open(latch);
  }
}
```

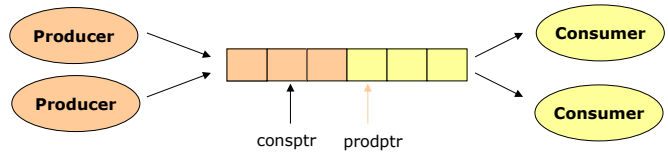
What happens if the buffer is full?

```
consumer(){
  while(TRUE) {
    int item;
    close(latch);
    item = buf[consptr];
    consptr = (consptr+1) % N;
    open(latch);
    cons(item);
  }
}
```

What happens if the buffer is empty?

10

Example: Producers - Consumers




```
int buf[N];
int prodptr=0, consptr=0;
latch_t latch;
semaphore_t can_prod = create_semaphore(N);
           can_cons = create_semaphore(0);
```

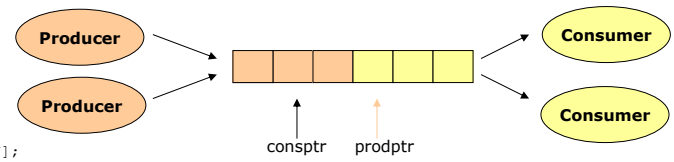
Each semaphore represents a resource:
- can_prod: free space; iniciatialized to N
- can_cons: items in the buffer; iniciatialized to 0

```
prod() {
  while(TRUE) {
    int item = produce();
    acquire(can_prod);
    close(latch);
    buf[prodptr] = item;
    prodptr = (prodptr+1) % N;
    open(latch);
    release(can_cons);
  }
}
```

```
cons(){
  while(TRUE) {
    int item;
    acquire(can_cons);
    close(latch);
    item = buf[consptr];
    consptr = (consptr+1) % N;
    open(latch);
    release(can_prod);
    consome(item);
  }
}
```

11

Example: Producers - Consumers




```
int buf[N];
int prodptr=0, consptr=0;
latch_t latch;
semaphore_t can_prod = create_semaphore(N),
           can_cons = create_semaphore(0);
```

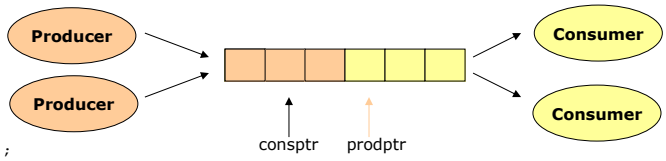
```
prod() {
  while(TRUE) {
    int item = produce();
    acquire (can_prod);
    close(latch);
    buf[prodptr] = item;
    prodptr = (prodptr+1) % N;
    open(latch);
    release(can_cons);
  }
}
```

```
cons(){
  while(TRUE) {
    int item;
    acquire(can_cons);
    close(latch);
    item = buf[consptr];
    consptr = (consptr+1) % N;
    open(latch);
    release(can_prod);
    consome(item);
  }
}
```

Can we have a solution that allows to produce and consume at the same time in diferent parts of the buffer?

14

Example: Producers - Consumers




```
int buf[N];
int prodptr=0, consptr=0;
latch_t latch_p, latch_c;
semaphore_t can_prod = create_semaphore(N),
            can_cons = create_semaphore(0);
```

Optmization: allows to produce and consume at the same time in different parts of the buffer

```
prod() {
  while(TRUE) {
    int item = produce();
    acquire(can_prod);
    close(latch_p);
    buf[prodptr] = item;
    prodptr = (prodptr+1) % N;
    open(latch_p);
    release(can_cons);
  }
}
```

```
cons(){
  while(TRUE) {
    int item;
    acquire(can_cons);
    close(latch_c);
    item = buf[consptr];
    consptr = (consptr+1) % N;
    open(latch_c);
    release(can_prod);
    consome(item);
  }
}
```

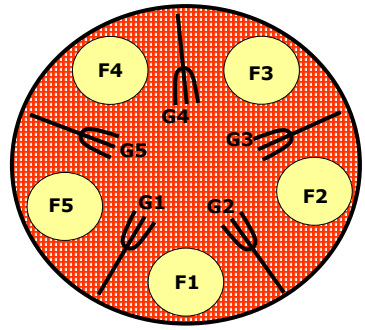
15

Philosophers Dinner

- Five philosophers are re-united to philosophate and to have dinner eating spaghetti:
  - To eat spaghetti each one needs two forks, but the table provides one fork per each
- Conditions:
  - The philosophers can be in one of the following three states: Think, Hungry, Eat.
  - Each place that a pilosopher occupies is fixed
  - Each pilosopher can only use the forks immediately on his left and right

16

Philosophers Dinner



17

Philosophers Dinner

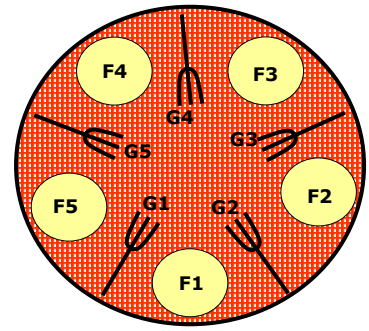
```
philosopher(int id){
   while (TRUE) {
      think();
      <acquire the forks>
      eat();
      <release the forks>
    }
}
```

18

Philosophers Dinner with Semaphores, version #1

```
semaphore_t new Semaphore fork[5] = {1, 1, 1, 1, 1};

philosopher(int id)
{
    while (TRUE) {
       think();
       acquire(fork[id]);
       acquire(fork[(id+1)%5]);
       eat();
       release(fork[id]);
       release(fork[(id+1)%5]);
    }
}
```



Problem?

19

Interblocking

- A situation of interblocking can occur if the four following conditions are simultaneously true:
  - At least on resource is used in a non-sharable way
  - There is at least one process that holds a resource and that is waiting to acquire mode resources
  - The resources are not “preemptible”, i.e., the resources are not voluntarily released by the process that hold them
  - There is a pattern of synchronization in which proc/thread T1 waits for a resource from T2 and circularly Tn+1 waits for a resource from T1

20

Interblocking– preventive solutions

- Making sure that all the resources are acquired in the same order
- Acquiring all the resources when the thread starts
- When the acquisition of resource is not possible for the progression of the algorithm, releasing all the resources a thread holds

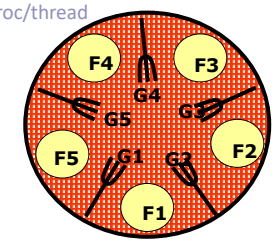
21

Philosophers Dinner with Semaphores, version #2

```
semaphore_t new Semaphore fork[5] =
   {1, 1, 1, 1, 1};

filosopher(int id)
{
    while (TRUE) {
        think();
        if (id%2 == 1)
          acquire(fork[(id+1)%5]);
          acquire(fork[id]);
        } else {
          acquire( fork[id]);
          acquire(fork[(id+1)%5]);
        }
        eat();
        release(fork[id]);
        release(fork[(id+1)%5]);
    }
}
```

- Always acquire the semaphores in the same order - *odd philosophers start left-hand first, and even philosophers start right-hand first*
- Other solutions:
  - Acquire resources at start of the execution with non-blocking call "trylock", release if it fails
  - Detection of interblocking: forced release by elimination of the proc/thread



22

Philosophers Dinner with Semaphores, version #3



23

Philosophers Dinner with Semaphores, version #3

```
#define THINK 0
#define HUNGRY 1
#define EAT 2
#define N 5
int state[N] = {0, 0, 0, 0, 0};
semaphore_t new Semaphore semphilo[N] =
      {0, 0, 0, 0, 0};
latch_t mutex;

Test(int k){
  if (state[k] == HUNGRY &&
      state[(k+1)%N] != EAT &&
      state[(k-1)%N] != EAT){
         state[k] = EAT;
         release(semphilo[K]);
  }
}
```

```
filosopher(int id)
{
    while (TRUE) {
        think();
        acquire(mutex);
        state[id] = HUNGRY;
        Test(id);
        release(mutex);
        acquire(semphilo[id]);
        eat();
        acquire(mutex);
        state[id] = THINK;
        Test((id-1+N)%N);
        Test((id+1)%N);
        release(mutex);
    }
}
```

- Use a semaphore to represent the conditions of blocking (and not the status of the forks/philosophers)
- Represent the status of the forks/philosophers with a set of variables that are accessed in a critical section

24

Philosophers Dinner with Semaphores, version #4

```
semaphore_t new Semaphore fork[5] =
    {1, 1, 1, 1, 1};
semaphore_t new Semaphore room = 4;

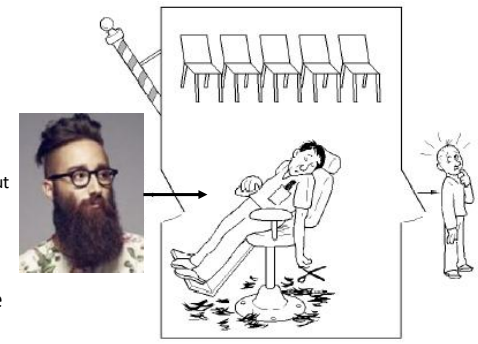
filosopher(int id)
{
    while (TRUE) {
        think();
        acquire(room);
        acquire(fork[id]);
        acquire(fork[(id+1)%5]);
        eat();
        release(fork[id]);
        release(fork[(id+1)%5]);
        release(room);
    }
}
```

- Limit the access to the room to N-1 filosophers (always be free at least a fork)

25

Barber

- In a barber shop there is a chair in which:
  - The barber cuts the hair
  - N (for clients) waiting chairs
- If there are no clients:
  - The barber sits on the chair and sleeps
- When a client arrives:
  - He must wake the sleeping barber for him to cut the client`s hair
- If, in the meantime, more client arrives, while the barber is cutting the hair to the first client:
  - They wait on a waiting chair, or
  - They leave, assuming there no more waiting chairs



26

Barber

```
void barber() {
   while(TRUE) {
      Acquire(clients);
      SeeClient();
      Acquire(mutex);
      clients_wating--;   /* remove a client */
      Release(barber); /* unblock a client */
      Release(mutex);
   }
}
```

```
#define MAX_CHAIRS N
latch_t mutex;
int clients_waiting= 0;
semaphore_t clients, barber;

main{ /*...*/
   mutex = new Semaphore(1);
   clients = new Semaphore(0);
   barber = new Semaphore(0);
/*...*/}
```

```
int customer() {
   Acquire(mutex);
   if (clients_waiting < MAX_CHAIRS) {
      client_waiting++;       /* add a client */
      Release(clients);       /* tell barber that there are clients */
      Release(mutex);
      Acquire(barber);        /* wait for the barber to be ready */
      HairCut();
   }
   else {
      Release(mutex);
      return 0;               /* give up if there are no chairs */
   }
```

27

Barber in Java

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.Random;

public class BarberShop {
    private final Lock lock = new ReentrantLock();
    private final Condition condition = lock.newCondition();
    private final int[] waiting = new int[3];
    private int waitingCount = 0;
    private static final int NUM_CHAIRS = 3;
    private Random random = new Random();

    public void barber() {
        while (true) {
            lock.lock();
            try {
                while (waitingCount == 0) {
                    try {
                        condition.await();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                int cust = waiting[0];
                for (int i = 0; i < waitingCount - 1; ++i) {
                    waiting[i] = waiting[i + 1];
                }
                waitingCount--;
                System.out.println("Barber cutting hair of customer " + cust);
            } finally {
                lock.unlock();
            }
            try {
                Thread.sleep(random.nextInt(3) + 1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("Barber finished with customer " + cust);
            lock.lock();
            try {
                condition.signal();
            } finally {
                lock.unlock();
            }
        }
    }
```

```
public void customer(int i) {
        try {
            Thread.sleep(random.nextInt(4) + 1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        lock.lock();
        try {
            if (waitingCount < NUM_CHAIRS) {
                waiting[waitingCount++] = i;
                System.out.println("Customer " + i + " waiting");
                condition.signal();
                while (waiting[0] != i) {
                    try {
                        condition.await();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                System.out.println("Customer " + i + " got haircut");
            } else {
                System.out.println("Customer " + i + " left (no seat)");
            }
        } finally {
            lock.unlock();
        }
    }
```

28

Barber in Java

```
public static void main(String[] args) {
        BarberShop barberShop = new BarberShop();
        Thread barberThread = new Thread(() -> barberShop.barber());
        barberThread.start();
        for (int i = 0; i < 5; ++i) {
            new Thread(() -> barberShop.customer(i)).start();
        }
        try {
            barberThread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

29

Condition Variables

30

Condition Variables

- A Condition Variable allow the synchronization between threads that depend on the Condition Variable boolean value
- It allows a thread to wait for a condition that depends on the actions of another thread
  - The variable condition is a boolean value that depends on shared variables
- The Condition Variables is always associated to a latch:
  - Latch that protects the critical sections that control the access to the on shared variables on which the Condition Variable depends on
  - There can be more than one single Condition Variable associated to the same latch

31

Condition Variables - operations

- **await() (or wait()):**
  - A thread calls this method to suspend its execution and release the associated lock atomically, placing itself in a wait queue until it is signalled by another thread.
  - When the thread wakes up, it re-acquires/tries to lock before continuing.
- **signal() (or notify()):**
  - A thread calls this method after changing the shared state to wake up a single thread that is waiting on the condition variable.
  - If there are threads waiting (who have invoked wait), unblocks one thread (that goes to the state "to run")
  - The thread that was unblocked can not progress if it does not have the latch
  - Signal is not memorized; so, if there are no threads waiting in the condition variable, the signal has no effect
- **signalAll() (or notifyAll()):**
  - This method wakes up all threads waiting on the condition variable
  - This is generally safer and more foolproof than signal() in complex scenarios.
- **Multiple Conditions Per Lock**:
  - A single Lock instance can have multiple Condition objects associated with it, allowing threads to wait for different, specific conditions (e.g., notFull and notEmpty in a bounded buffer).
  - The Object.wait() mechanism only has one implicit wait set per object.
- **Flexibility and Readability**:
  - The explicit Lock and Condition objects make the synchronization logic clearer and more structured, reducing the chance of errors.
- **Timed Waits**:
  - The Condition interface provides methods like await(long time, TimeUnit unit) for timed waits, which can help prevent deadlocks or handle cases where a condition never occurs.
- **wait:**
  - Wait should be invoked while the latch is closed
- **signal/broadcast:**
  - After a signal has been received by the threads (that were blocked) the latch is not automatically acquired by the same thread (Mesa)
  - It is up to the programmer to posteriorly release the latch (Mesa)

32

Typical Examples

Code that waits for a Condition

```
lock(latch);
/* ..acess to shared variables.. */
while (! conditionOverTheSharedState()) wait(ConditionVar, latch);
/* ..access to shared variables.. */
unlock(latch);
```

Code that activates the Condition

```
lock(latch);
/* ..acess to shared variables.. */

/* if the state has been modified in a way that allows other
threads to progress, it calls either signal (or broadcast)*/
signal/broadcast (ConditionVar, latch);
unlock(latch);
```

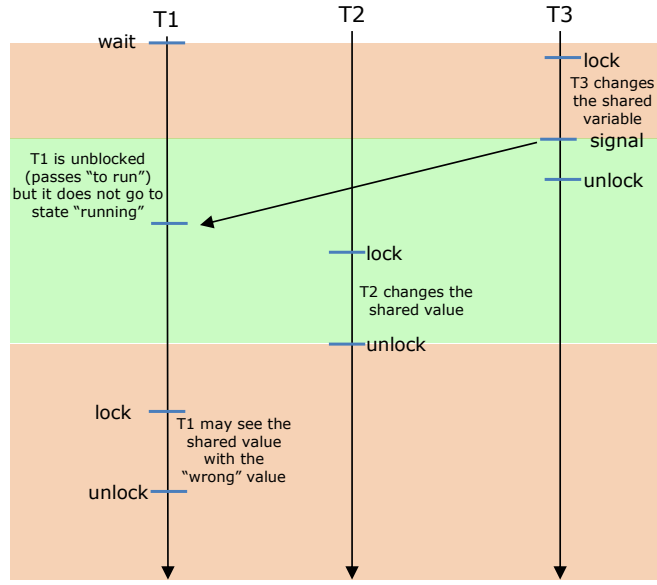
It is crucial to release the lock in a `finally` block to prevent the thread from holding the lock indefinitely in case of an exception. Also, notice the use of a `while` loop when calling `await()` (or `wait()`) to handle spurious wakeups, which can occur even if no signal was sent.

33

Condition Variables - problem

- Attention:
  - During the time that passes between the signal (T3) and a thread be unblocked (T1) acquiring the latch
  - The condition variable can be changed by another thread (T2)

Semantics Mesa (here is assumed) vs Semantics Hoare (see monitors)



34

Condition Variables - solution

- When a thread is unblocked (returns from wait):
  - It is not guaranteed that the value of the shared variables allow the thread to continue:
    - in the previous slide: the thread that is unblocked (T1) can not be the first to enter into the critical section after the thread that has performed the signal (T3)
  - **It should not be done:** if (test shared variable) wait
  - **It should be done**: while (test shared variable) wait

- Errors to avoid:
  - Do not close the latch before invoking wait: it can make the thread to not wait
  - Do not open the latch after doing signal: it can not unblock the thread waiting

35

"If vs While" in Condition Variables

- Always use `WHILE` instead of `IF`
- When you check a Condition Variable, using `while` instead of `if` is always correct
- The correction of using `if` depends on the semantics of signal
- The use of `while` also contemplates the situations in which occur spurious wakeups:
  - In some implementations, it is possible that two threads be unblocked in spite of being done just one signal

36

Condition Variables - POSIX

- Creation/destruction of Condition Variables:
  - pthread_cond_init (condition,attr)
  - pthread_cond_destroy (condition)
  - pthread_condattr_init (attr)
  - pthread_condattr_destroy (attr)
- Acquire and release in Condition Variables:
  - pthread_cond_wait (condition,mutex)
  - pthread_cond_signal (condition)
  - pthread_cond_broadcast (condition)

37

Producers – Consumers with Condition Variables

```
int buf[N], prodptr=0, consptr=0, count=0;

mutex_t mutex;
cond_t empty, full;
```

```
producer() {
  while(TRUE) {
    int item = produce();
    pthread_mutex_lock(&mutex);
    while (count == MAX) pthread_cond_wait(&empty, &mutex);
    buf[prodptr] = item;
    prodptr = (prodptr+1) % N;
    count++;
    pthread_cond_signal(&full);
    pthread_mutex_unlock(&mutex);
  }
}

consumer(){
   while(TRUE) {
      int item;
      pthread_mutex_lock(&mutex);
      while (count == 0) pthread_cond_wait(&full, &mutex);
      item = buf[consptr];
      consptr = (consptr+1) % N;
      count--;
      pthread_cond_signal(&empty);
      pthread_mutex_unlock(&mutex);
   }
}
```

38

Monitors

39

Monitors

- Goal
  - Mechanism of synchronization to programming languages that solves most of the problems of shared data:
    - Guarantee implicitly the mutual exclusion
    - Mechanisms to do the explicit synchronization of threads in algorithms of cooperation or competion

40

Synchronization: Implicit and Explicit

- Implicit:
  - Execution of the procedures is done in mutual exclusion, guaranteeing that the data of the monitor is always updated inside a critical section
- Explicit:
  - Conditions that control explicitly the synchronization of the threads that use the monitor
  - Thus, the primitives of synchronization, in the initial proposition, were called as *condition variables* or *conditions*
  - Conditions are explicitly programmed in the methods of the monitor
  - If a condition is false (the invariant is not garanteed):
    - a thread is blocked
  - If the condition becomes true:
    - A thread wich is blocked should be unblocked

42

Monitors




```
synchronized void myMethod () {
    //Here, I am in the mutual exclusion
    …
    while (condition == false) wait();
    …
    //Here, I am in in the mutual exclusion and the condition verifies
    …
   // When I return, I release the mutual exclusion
}
```

43

Monitors

- Declared as an abstract type, class or module:
  - Internal data structure
  - Functional interface:
    - Methods accessed in mutual exclusion (keyword synchronized)
    - Methods that do not modify the state and that can be invoked without being in mutual exclusion

44

Monitors - Sinchronization

- Mutual exclusion:
  - Implicit in the monitor entry
- Thread that goes into the monitor has access to the critical section
- Thread that leaves the monitor releases the critical section

45

Monitors - Sinchronization

- Condition variables:
  - Declared in the internal data structure
- Wait
  - Releases the critical section
  - Thread is located in a queue dedicated to the wait condition
  - Thread always block
- Signal
  - Signals the condition
  - If there are threads in the condition queue, unblock the first one
  - If there are no threads in the condition queue, signal has no effect

46

Signal Semantics

- Usual semantics (Mesa):
  - Signal unblocks a thread of the queue associated with the condition variable.
  - But it does not release the critical exclusion area.
    - Wait only returns when the thread that was unblocked enter into the critical section
    - When does it happen?
  - If there are no threads the effect of a signal is lost (contrary to semaphores, conditions variables do not memorize the events).
- There are others semantics.

47

Monitors Semantics

- Mesa (the most usual):
  - Thread T1 that invokes the signal keeps the latch and the CPU
  - Thread T2 blocked on wait is unblocked going into the state “to run” (but not yet acquiring the latch)
- Hoare:
  - Thread T1 that invokes the signal releases the latch, passes the CPU to thread T2 that executes immediately (acquiring the latch atomically)
  - Thread T2 that was unblocked, when it releases the latch (i.e., it leaves the critical section or makes wait again), the CPU passes to thread T1 that made the signal

48

Problems in the Definition of Monitors

- Calls from one monitor to other monitors:
  - Possibility of deadlock
- Signal Semantics (the most usual one: Mesa)
  - When is invoked a signal, a thread may be surpassed by another one

49

Monitors: mailbox (in Java) (producers-consumers)

```
class mailBox {
    int MAX = 10;
    int[] buffer = new int[MAX];
    int count = 0;
    int indPro = 0;
    int indRem = 0;

    synchronized void send () {
        while (count == MAX) wait();
        tampao[indPro] = message;
        indPro++;
        if (indPro == MAX) indPro = 0;
        count ++;
        notifyAll();
    }

    synchronized void receive () {
        while (count == 0) wait();
        message = tampao[indRem] ;
        indRem++;
        if (indRem == MAX) indRem = 0;
        count --;
        notifyAll();
    }
}
```

50

End of L17v26

51

Repetition

Eric Jul and Paulo Ferreira
Programming Technology Group
Programming Section
Department of Informatics
University of Oslo

1

Lectures – what you are supposed to know

2

Starting

- Intro
- There are multi-core machines that require concurrent programming
- There are some tasks that are inherently parallel and other that are not:
  - Plant 10000 trees
  - A woman produce a baby in 9 months
- Threads concept and how they work in Java

3

Efficiency

- Speedup - Efficiency
- Efficiency and correct
- It is really simple to write a program that is not correct

- Super linear speedup (is it possible to has 10 cores and have a speedup more than 10) – can you achieve that? there must be something wrong…
- Amdahl Law
- Cost of creating threads
- Oblig1 - Find the k largest numbers in a large array

4

Time Slicing

- Time slicing
- Concurrent reads are OK but concurrent writes are not
- Synchronization (semaphores, cyclic barrier, etc.):
  - Reads are OK
  - Concurrent writes are not !
- There will be nothing on quantum computing

5

Measuring

- Measuring programs (speed, wall clock time)
- This times are not the same (median will be used):
  - example of driving taking into account rush or not
- You can be hit by a garbage collection, Internet activity (big mail coming in), effects of caching, just in time compilation (JIT), pre-fetching – different from a few years ago!
- You should warm up !

6

Caching

- Why ?
  - Slow speed of light (299,792,458 m/s)
  - Trade-off speed vs size
- The meter is defined by the speed of light - meter was defined as the fraction of distance from the equator to the north pole
- How far does electricity propagate in copper in 1 ns ?
  - Light travels 6 cm (2 clock cycles)
- Multi–level caching
- Memory takes 200 cycles and L0 caching take 4 cycles (a factor of 50)
- A mechanism that is called prefetching
- Oblig2 matrix

7

Moore´s

- Moore´s law
  - It was valid until recently
  - Switch transistors is faster
- Performance leads to caching
- There are multi-cores CPU
- How to parallelize:
  - Split (different data for several threads)
  - Program pipeline
- Divide and Conquer: Convex Hull
- Economize the creation of threads
- Pool of threads (Java has this)

8

Sieve and Factorization

- Prime number oblig3
- You parallelize (an iterative) sieve and the factorization

9

Synchronization

- Sketch
- Andes trains with blind and deaf drivers
- Chefs and waiters:
  - Busy waiting
  - Action initiated
  - wait and signal
- Hoare (1973) and Mesa monitors
- Oblig4 convex

10

More on Synchronization

- Paralellization of Convex Hull
- Oblig5 synchronization
- Synchronization
- Race conditions (two threads want to grab something)-varispeed
- Travelings salesman method - Monte Carlo
  - The optimal solution takes a long time
  - How much improvement did I get the last time
- Multiple semaphores:
  - Problem: signal a semaphore (before waiting you have to release the lock)

11

Yet More on Synchronization

- Competition
- Typical problems of synchronization (producer consumer, philosophers ´ dinner, and barber shop)
- Condition variables

12

Competition

- A proc/thread demands a resource:
  - runs acquire (SemResource)
- A proc/thread frees a resource:
  - runs release (SemResource)
- The semaphore that controls the algorithm is initialized with the number of availabe resources:
  - SemResource = new Semaphore(NUM_RESOURCES)

16

Semaphores (example)

```
#define MAX_STACK 100
char*    stack[MAX_STACK];
int      top = MAX_STACK-1;
Semaphore SemMutEx;
Semaphore SemMem;

char *ask_for_mem() {
   acquire (SemMem);
   acquire (SemMutEx);
   ptr = stack[top];
   top--;
   release (SemMutEx);
   return ptr;
}

void get_back_mem() {
   acquire(SemMutEx);
   top++;
   stack[top] = ptr;
   release (SemMutEx);
   release (SemMem);
}
```

Memory Allocator with semaphore SemMem that controls the free memory !

```
main(){
/*...*/
   semExMut = new Semaphore(1);
   semMem = new Semaphore(MAX_STACK);
}
```

The semaphore is initialized with the number of available resources for the client

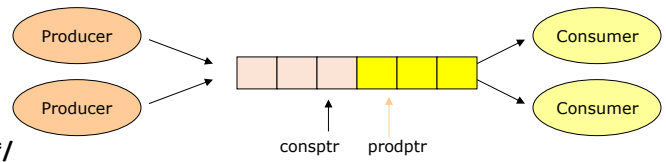
17

Typical Examples of Synchronization

- Producers/Consumers
  - Threads/procs that produce information to a buffer and threads/procs that read the information from that buffer (semaphores)
- Dining-Philosophers
  - Threads/procs that eat spaghetti and has less forks that eaters (semaphores)
- Barber
  - One or more threads/procs that serve the client thread (semaphores and conditions

18

Example: Producers - Consumers



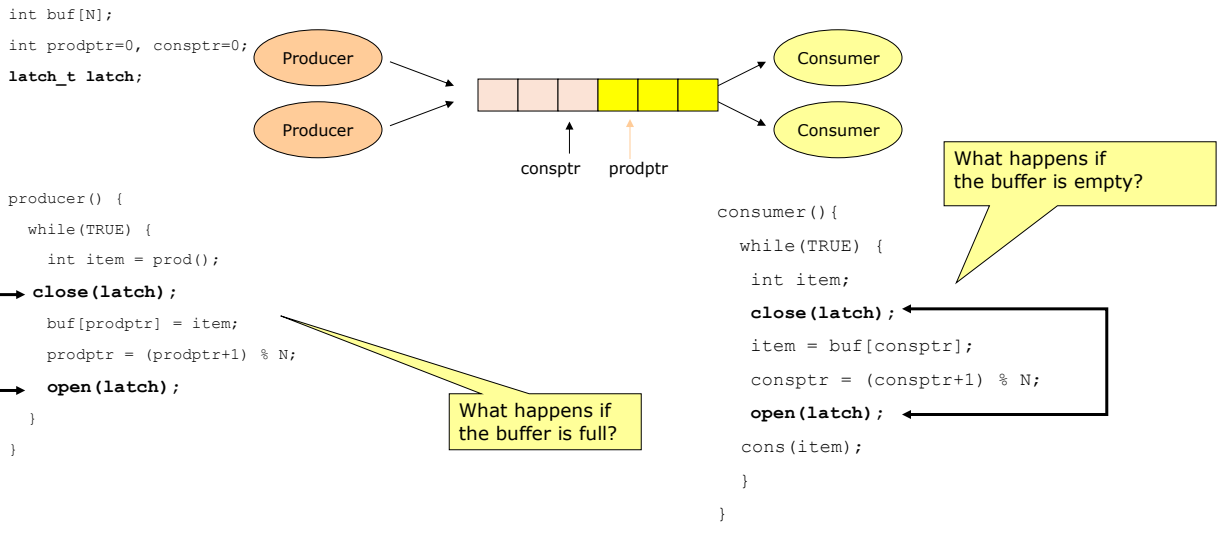

```
/* ProducerConsumer */
int buf[N];
int prodptr=0, consptr=0;

producer()
  while(TRUE) {
    int item = prod();
    buf[prodptr] = item;
    prodptr = (prodptr+1) % N;
  }
}
```

```
consumer(){
  while(TRUE) {
    int item;
    item = buf[consptr];
    consptr = (consptr+1) % N;
    cons(item);
  }
}
```

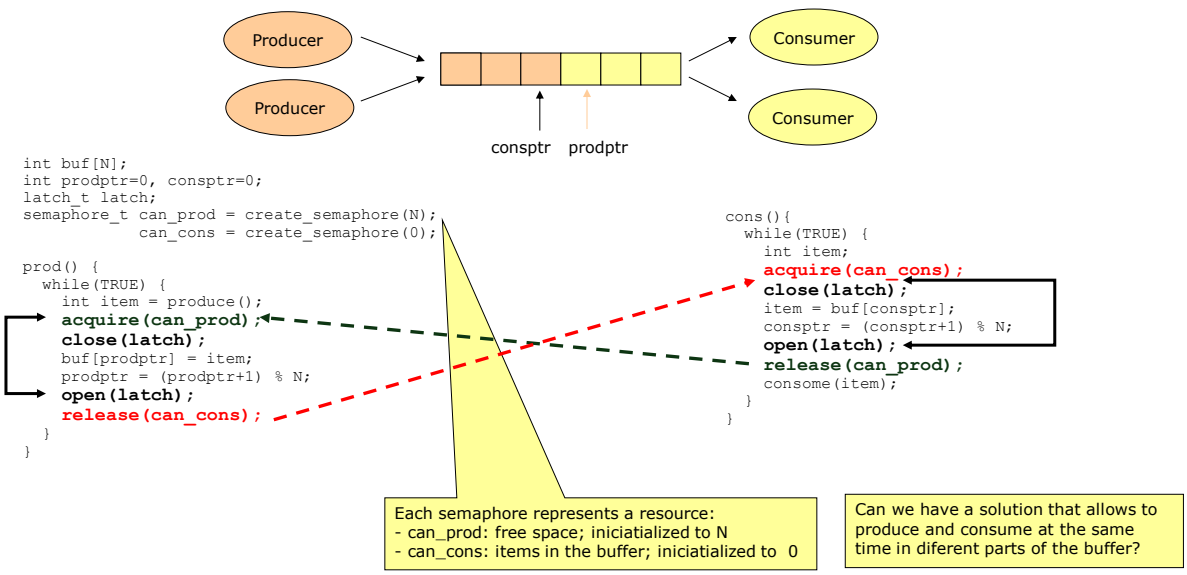
19

Example: Producers - Consumers



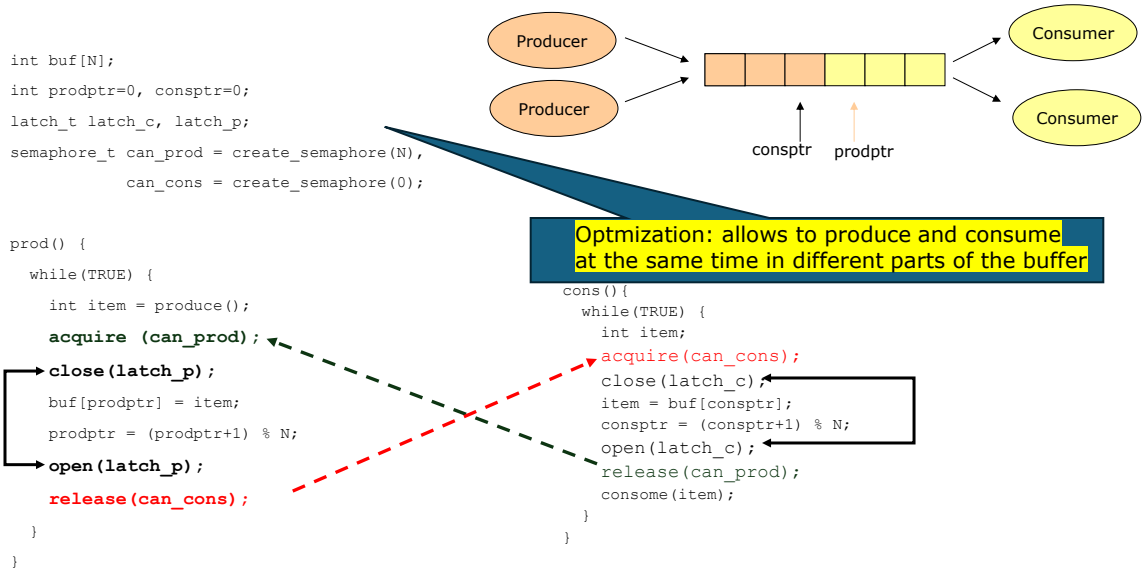
20

Example: Producers - Consumers



21

Example: Producers - Consumers




```
int buf[N];
int prodptr=0, consptr=0;
latch_t latch_c, latch_p;
semaphore_t can_prod = create_semaphore(N),
            can_cons = create_semaphore(0);

prod() {
  while(TRUE) {
    int item = produce();
    acquire (can_prod);
    close(latch_p);
    buf[prodptr] = item;
    prodptr = (prodptr+1) % N;
    open(latch_p);
    release(can_cons);
  }
}
```

```
cons(){
  while(TRUE) {
    int item;
    acquire(can_cons);
    close(latch_c);
    item = buf[consptr];
    consptr = (consptr+1) % N;
    open(latch_c);
    release(can_prod);
    consome(item);
  }
}
```

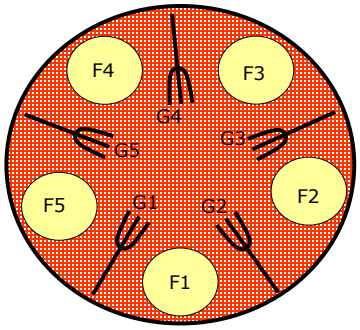
22

Philosophers Dinner

- Five philosophers are re-united to philosophate and to have dinner eating spaghetti:
  - To eat spaghetti each one needs two forks, but the table provides one fork per each
- Conditions:
  - The philosophers can be in one of the following three states: Think, Hungry, Eat.
  - Each place that a pilosopher occupies is fixed
  - Each pilosopher can only use the forks immediately on his left and right

23

Philosophers Dinner

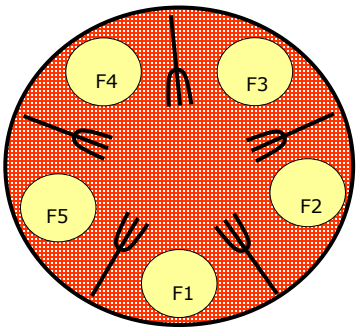


24

Philosophers Dinner

```
semaphore_t new Semaphore fork[5] = {1, 1, 1, 1, 1};

philosopher(int id)
{
    while (TRUE) {
       think();
       acquire(fork[id]);
       acquire(fork[(id+1)%5]);
       eat();
       release(fork[id]);
       release(fork[(id+1)%5]);
    }
}
```



25

Interblocking

- A situation of interblocking can occur if the four following conditions are simultaneously true:
  - At least one resource is used in a non-sharable way
  - There is at least one process that holds a resource and that is waiting to acquire mode resources
  - The resources are not “preemptible”, i.e., the resources are not voluntarily released by the process that hold them
  - There is a pattern of synchronization in which proc/thread T1 waits for a resource from T2 and circularly Tn+1 waits for a resource from T1

26

Interblocking– preventive solutions

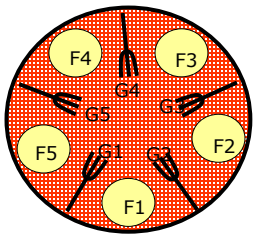
- Making sure that all the resources are acquired in the same order
- Acquiring all the resources when the thread starts
- When the acquisition of resource is not possible for the progression of the algorithm, releasing all the resources a thread holds

27

Philosophers Dinner with Semaphores, version #2

```
semaphore_t new Semaphore fork[5] =
   {1, 1, 1, 1, 1};

filosopher(int id)
{
    while (TRUE) {
        think();
        if (id%2 == 1)
          acquire(fork[(id+1)%5]);
          acquire(fork[id]);
        } else {
          acquire( fork[id]);
          acquire(fork[(id+1)%5]);
        }
        eat();
        release(fork[id]);
        release(fork[(id+1)%5]);
    }
}
```



- Always acquire the semaphores in the same order - *odd philosophers start left-hand first, and even philosophers start right-hand first*
- Other solutions:
  - Acquire resources at start of the execution with non-blocking call "trylock", release if it fails
  - Detection of interblocking: forced release by elimination of the proc/thread

28

Philosophers Dinner with Semaphores, version #3



29

Philosophers Dinner with Semaphores, version #3

```
#define THINK 0
#define HUNGRY 1
#define EAT 2
#define N 5
int state[N] = {0, 0, 0, 0, 0};
semaphore_t new Semaphore semphilo[N] =
      {0, 0, 0, 0, 0};
latch_t mutex;

Test(int k){
  if (state[k] == HUNGRY &&
      state[(k+1)%N] != EAT &&
      state[(k-1)%N] != EAT){
         state[k] = EAT;
         release(semphilo[K]);
  }
}
```

```
filosopher(int id)
{
    while (TRUE) {
        think();
        acquire(mutex);
        state[id] = HUNGRY;
        Test(id);
        release(mutex);
        acquire(semphilo[id]);
        eat();
        acquire(mutex);
        state[id] = THINK;
        Test((id-1+N)%N);
        Test((id+1)%N);
        release(mutex);
    }
}
```

- Use a semaphore to represent the conditions of blocking (and not the status of the forks/philosophers)
- Represent the status of the forks/philosophers with a set of variables that are accessed in a critical section

30

Philosophers Dinner with Semaphores, version #4

```
semaphore_t new Semaphore fork[5] =
    {1, 1, 1, 1, 1};
semaphore_t new Semaphore room = 4;

filosopher(int id)
{
    while (TRUE) {
        think();
        acquire(room);
        acquire(fork[id]);
        acquire(fork[(id+1)%5]);
        eat();
        release(fork[id]);
        release(fork[(id+1)%5]);
        release(room);
    }
}
```

- Limit the access to the room to N-1 filosophers (always be free at least a fork)

31

Barber

- In a barber shop there is a chair in which:
  - The barber cuts the hair
  - N (for clients) waiting chairs
- If there are no clients:
  - The barber sits on the chair and sleeps
- When a client arrives:
  - He must wake the sleeping barber for him to cut the client`s hair



- If, in the meantime, more client arrives, while the barber is cutting the hair to the first client:
  - They wait on a waiting chair, or
  - They leave, assuming there no more waiting chairs

32

Barber

```
void barber() {
  while(TRUE) {
    Acquire(clients);
    SeeClient();
    Acquire(mutex);
    clients_wating--;  /* remove a client */
    Release(barber); /* unblock a client */
    Release(mutex);
  }
}
```

```
#define MAX_CHAIRS N
latch_t mutex;
int clients_waiting= 0;
semaphore_t clients, barber;

main{ /*...*/
   mutex = new Semaphore(1);
   clients = new Semaphore(0);
   barber = new Semaphore(0);
/*...*/}
```

```
int customer() {
  Acquire(mutex);
  if (clients_waiting < MAX_CHAIRS) {
    client_waiting++;     /* add a client */
    Release(clients);     /* tell barber that there are clients */
    Release(mutex);
    Acquire(barber);      /* wait for the barber to be ready */
    HairCut();
  }
  else {
    Release(mutex);
    return 0;             /* give up if there are no chairs */
  }
```

33

Barber in Java

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.Random;

public class BarberShop {
    private final Lock lock = new ReentrantLock();
    private final Condition condition = lock.newCondition();
    private final int[] waiting = new int[3];
    private int waitingCount = 0;
    private static final int NUM_CHAIRS = 3;
    private Random random = new Random();

    public void barber() {
        while (true) {
            lock.lock();
            try {
                while (waitingCount == 0) {
                    try {
                        condition.await();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                int cust = waiting[0];
                for (int i = 0; i < waitingCount - 1; ++i) {
                    waiting[i] = waiting[i + 1];
                }
                waitingCount--;
                System.out.println("Barber cutting hair of customer " + cust);
            } finally {
                lock.unlock();
            }
            try {
                Thread.sleep(random.nextInt(3) + 1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("Barber finished with customer " + cust);
            lock.lock();
            try {
                condition.signal();
            } finally {
                lock.unlock();
            }
        }
    }
```

```
public void customer(int i) {
        try {
            Thread.sleep(random.nextInt(4) + 1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        lock.lock();
        try {
            if (waitingCount < NUM_CHAIRS) {
                waiting[waitingCount++] = i;
                System.out.println("Customer " + i + " waiting");
                condition.signal();
                while (waiting[0] != i) {
                    try {
                        condition.await();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                System.out.println("Customer " + i + " got haircut");
            } else {
                System.out.println("Customer " + i + " left (no seat)");
            }
        } finally {
            lock.unlock();
        }
    }
```

34

Barber in Java

```
public static void main(String[] args) {
        BarberShop barberShop = new BarberShop();
        Thread barberThread = new Thread(() -> barberShop.barber());
        barberThread.start();
        for (int i = 0; i < 5; ++i) {
            new Thread(() -> barberShop.customer(i)).start();
        }
        try {
            barberThread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

35

Condition Variables

36

Condition Variables

- A Condition Variable allow the synchronization between threads that depend on the Condition Variable boolean value
- It allows a thread to wait for a condition that depends on the actions of another thread
  - The variable condition is a boolean value that depends on shared variables
- The Condition Variables is always associated to a latch:
  - Latch that protects the critical sections that control the access to the on shared variables on which the Condition Variable depends on
  - There can be more than one single Condition Variable associated to the same latch

37

Condition Variables - operations

- **await() (or wait()):**
  - A thread calls this method to suspend its execution and release the associated lock atomically, placing itself in a wait queue until it is signalled by another thread.
  - When the thread wakes up, it re-acquires/tries to lock before continuing.
- **signal() (or notify()):**
  - A thread calls this method after changing the shared state to wake up a single thread that is waiting on the condition variable.
  - If there are threads waiting (who have invoked wait), unblocks one thread (that goes to the state "to run")
  - The thread that was unblocked can not progress if it does not have the latch
  - Signal is not memorized; so, if there are no threads waiting in the condition variable, the signal has no effect
- **signalAll() (or notifyAll()):**
  - This method wakes up all threads waiting on the condition variable
  - This is generally safer and more foolproof than signal() in complex scenarios.
- **Multiple Conditions Per Lock**:
  - A single Lock instance can have multiple Condition objects associated with it, allowing threads to wait for different, specific conditions (e.g., notFull and notEmpty in a bounded buffer).
  - The Object.wait() mechanism only has one implicit wait set per object.
- **Flexibility and Readability**:
  - The explicit Lock and Condition objects make the synchronization logic clearer and more structured, reducing the chance of errors.
- **Timed Waits**:
  - The Condition interface provides methods like await(long time, TimeUnit unit) for timed waits, which can help prevent deadlocks or handle cases where a condition never occurs.
- **wait:**
  - Wait should be invoked while the latch is closed
- **signal/broadcast:**
  - After a signal has been received by the threads (that were blocked) the latch is not automatically acquired by the same thread (Mesa)
  - It is up to the programmer to posteriorly release the latch (Mesa)

38

Typical Examples

Code that waits for a Condition

```
lock(latch);
/* ..acess to shared variables.. */
while (! conditionOverTheSharedState()) wait(ConditionVar, latch);
/* ..access to shared variables.. */
unlock(latch);
```

Code that activates the Condition

```
lock(latch);
/* ..acess to shared variables.. */

/* if the state has been modified in a way that allows other
threads to progress, it calls either signal (or broadcast)*/
signal/broadcast (ConditionVar, latch);
unlock(latch);
```

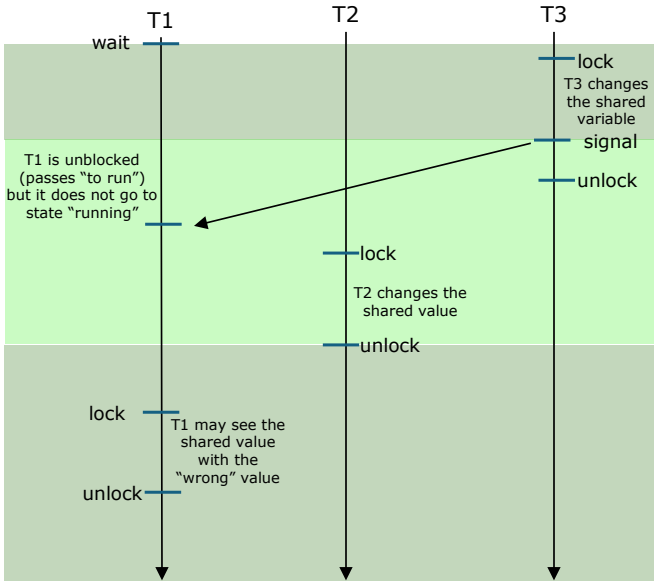
It is crucial to release the lock in a `finally` block to prevent the thread from holding the lock indefinitely in case of an exception. Also, notice the use of a `while` loop when calling `await()` (or `wait()`) to handle spurious wakeups, which can occur even if no signal was sent.

39

Condition Variables - problem

- Attention:
  - During the time that passes between the signal (T3) and a thread be unblocked (T1) acquiring the latch
  - The condition variable can be changed by another thread (T2)

Semantics Mesa (here is assumed) vs Semantics Hoare (see monitors)



40

Condition Variables - solution

- When a thread is unblocked (returns from wait):
  - It is not guaranteed that the value of the shared variables allow the thread to continue:
    - in the previous slide: the thread that is unblocked (T1) can not be the first to enter into the critical section after the thread that has performed the signal (T3)
  - **It should not be done:** if (test shared variable) wait
  - **It should be done**: while (test shared variable) wait

- Errors to avoid:
  - Do not close the latch before (outside the critical section) invoking wait: it can make the thread to not wait
  - Do not open the latch after doing signal (outside the critical section) : it can not unblock the thread waiting

41

“If vs While” in Condition Variables

- Always use `WHILE` instead of `IF`
- When you check a Condition Variable, using `while` instead of `if` is always correct
- The correction of using `if` depends on the semantics of signal
- The use of `while` also contemplates the situations in which occur spurious wakeups:
  - In some implementations, it is possible that two threads be unblocked in spite of being done just one signal

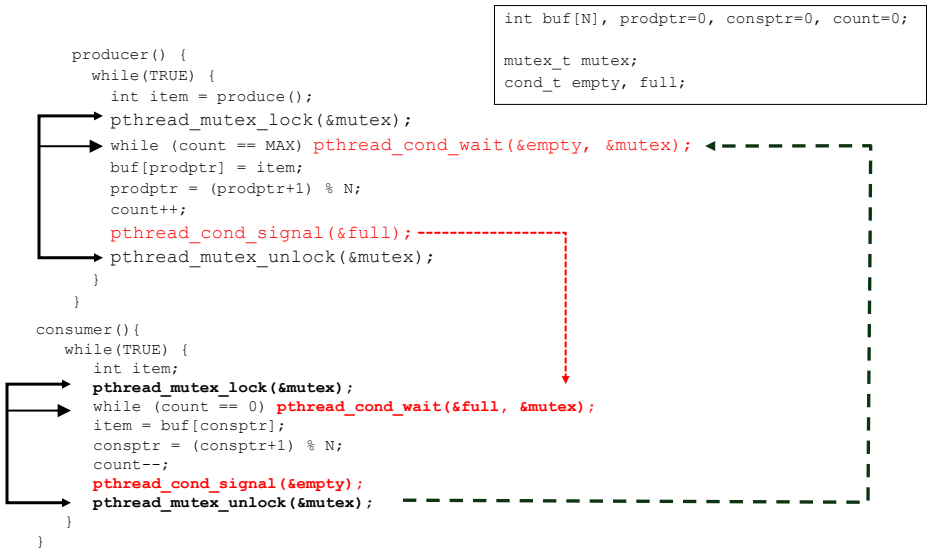
42

Condition Variables - POSIX

- Creation/destruction of Condition Variables:
  - pthread_cond_init (condition,attr)
  - pthread_cond_destroy (condition)
  - pthread_condattr_init (attr)
  - pthread_condattr_destroy (attr)
- Acquire and release in Condition Variables:
  - pthread_cond_wait (condition,mutex)
  - pthread_cond_signal (condition)
  - pthread_cond_broadcast (condition)

43

Producers – Consumers with Condition Variables



44

End of L20v26

57

Exam Hints

Eric Jul and Paulo Ferreira
Programming Technology Group
Programming Section
Department of Informatics
University of Oslo

1

Read All the Exam

- Read all the questions (find the hanging fruit…)
- Watch time for each question
- Bring some food and water
- Show up before the exam

2

End of L21v26

3

UNIVERSITY
OF OSLO

Convex Hull

What? Why? How?

Brage Isak Keiserås

17th March 2026



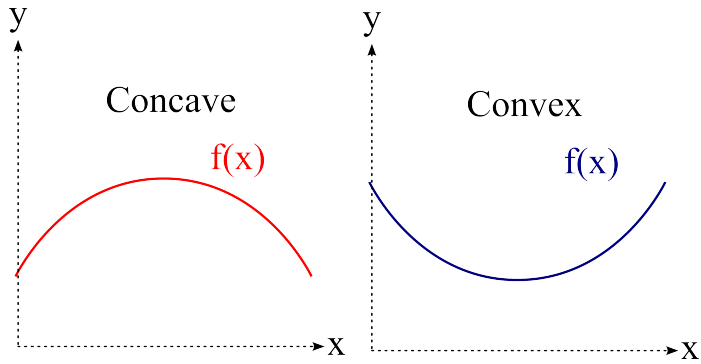
Outline

1. What?
2. Why?
3. How?

What?

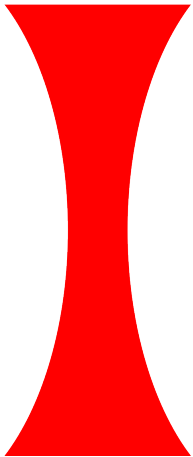
1. What?
2. Why?
3. How?

Concave/Convex Functions

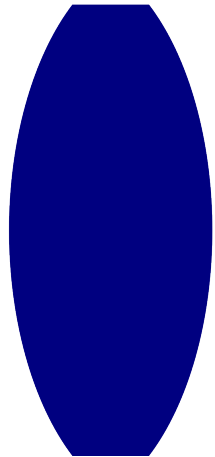


Concave/Convex Lenses

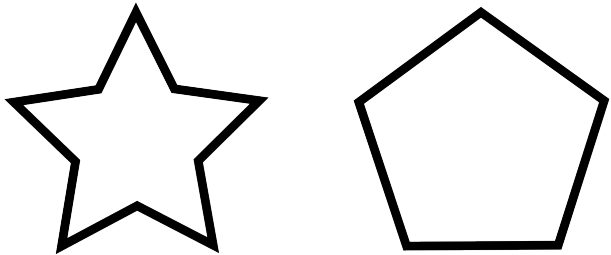
Concave



Convex



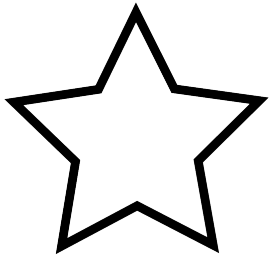
Concave/Convex

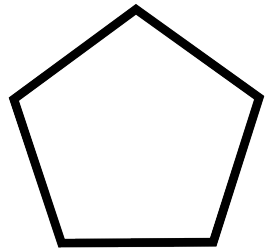


Concave/Convex

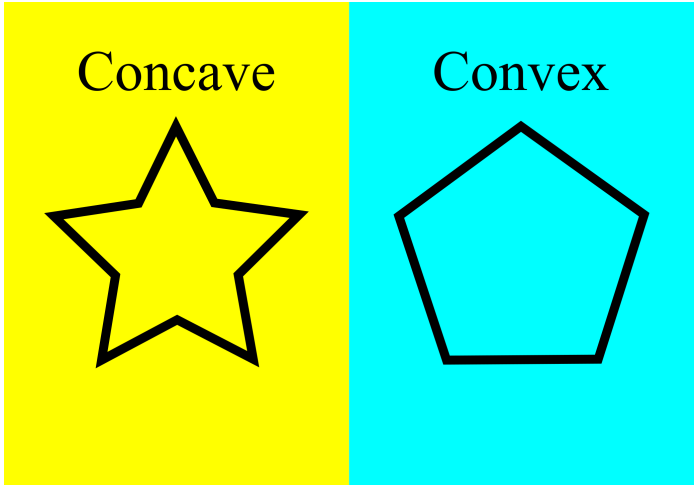
- An object is concave if it dips "inwards".
- If every line between any two points inside an object is contained inside an object then the object is convex.
  - For any points a, b inside O, ab is inside O

Concave/Convex

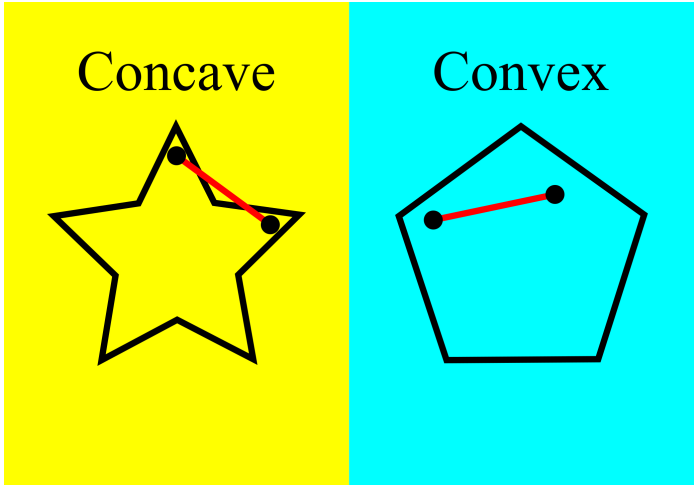




Concave/Convex



Concave/Convex



Convex Hull — What?

Convex Hull — What?

The convex hull of a set of points is the smallest convex object containing all points in the input.

Convex Hull — What?

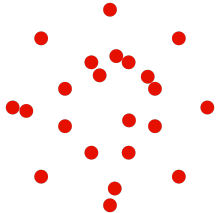
The convex hull of a set of points is the smallest convex object containing all points in the input.
If you imagine the points as nails nailed into a board, The convex hull is like a piece of string wrapped around the nails.

Convex Hull — What?

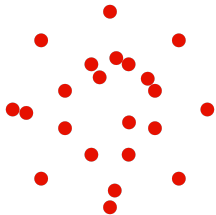
The convex hull of a set of points is the smallest convex object containing all points in the input.
If you imagine the points as nails nailed into a board, The convex hull is like a piece of string wrapped around the nails.

Convex Hull	
Input:	Set of points, P
Output:	Smallest object H such that H is convex, and all points in P are contained in H

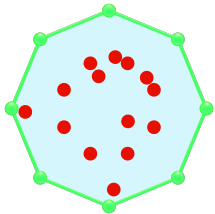
What is the convex hull of this guy?



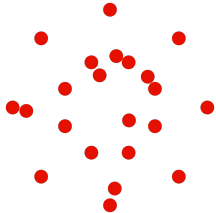
What is the convex hull of this guy?



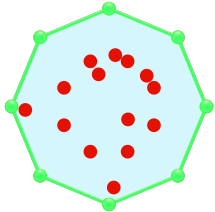
Is it this guy:



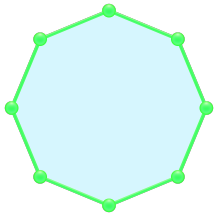
What is the convex hull of this guy?



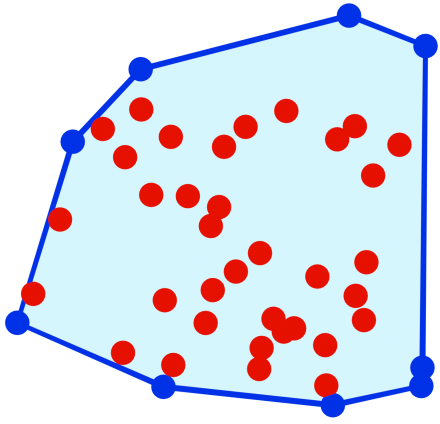
Is it this guy:



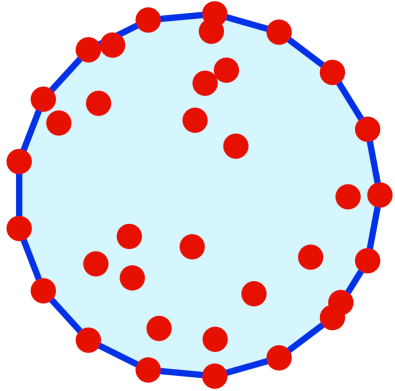
Or is it this guy:



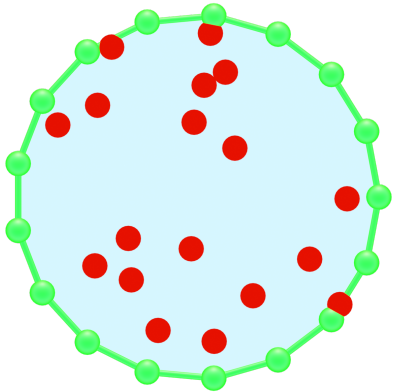
Video



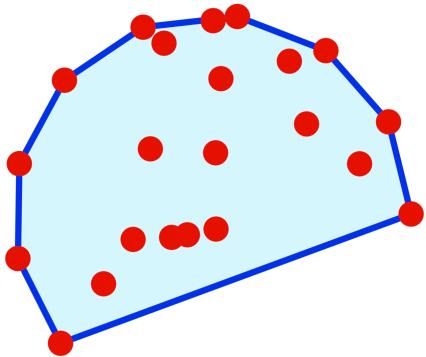
Concave or Convex?



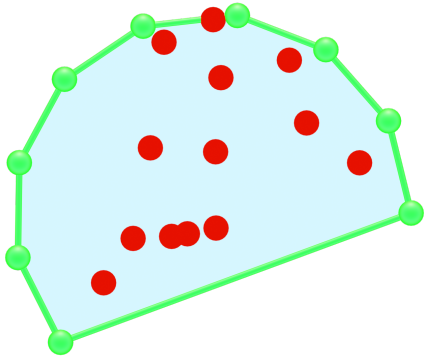
Concave or Convex?



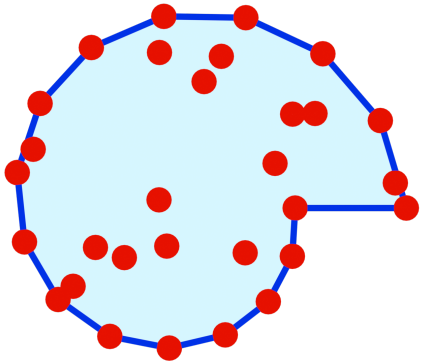
Concave or Convex?



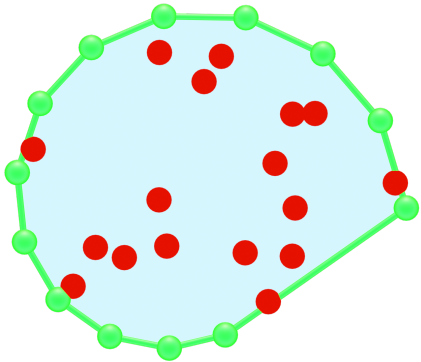
Concave or Convex?



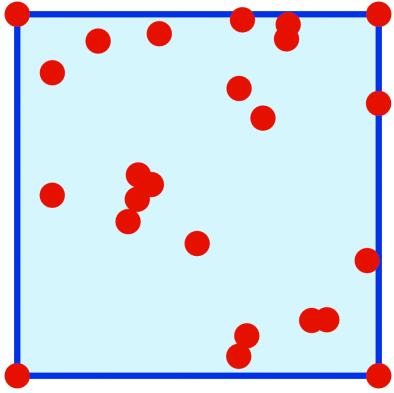
Concave or Convex?



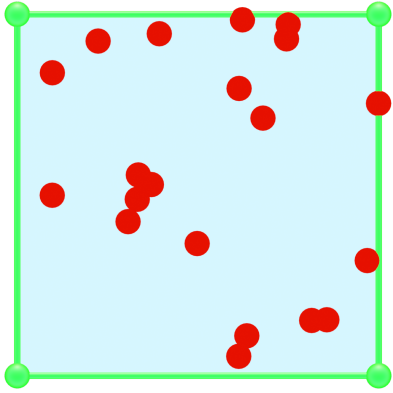
Concave or Convex?



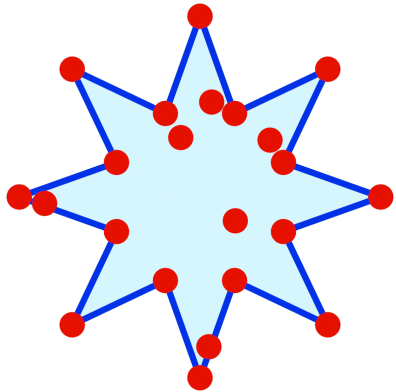
Concave or Convex?



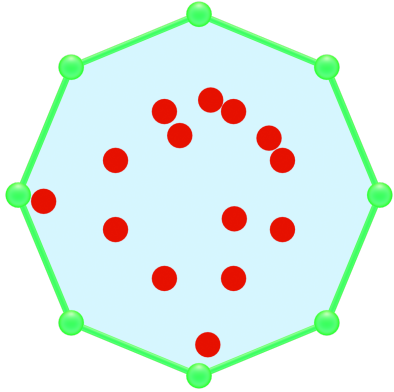
Concave or Convex?



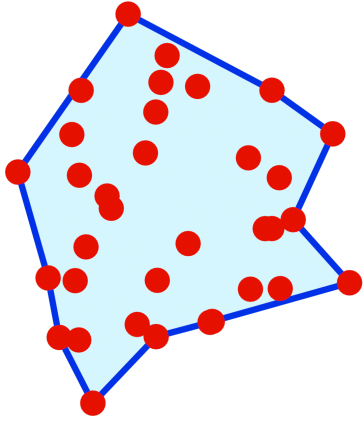
Concave or Convex?



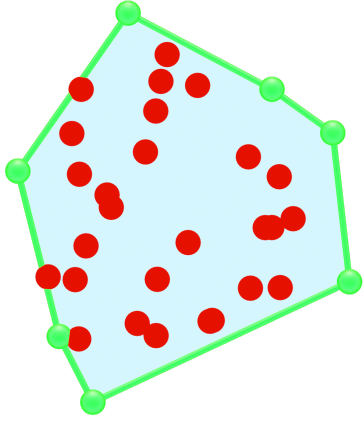
Concave or Convex?



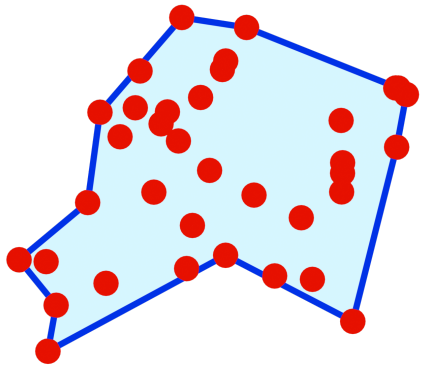
Concave or Convex?



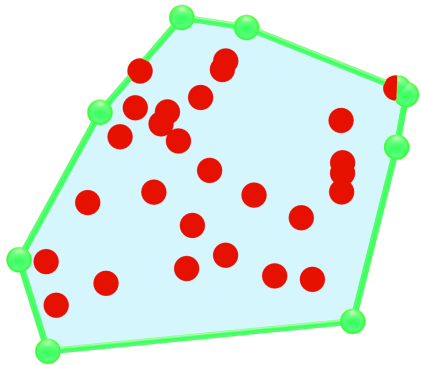
Concave or Convex?



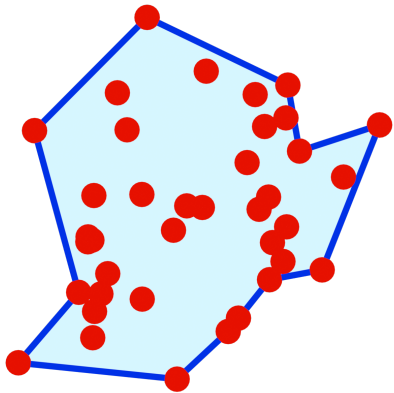
Concave or Convex?



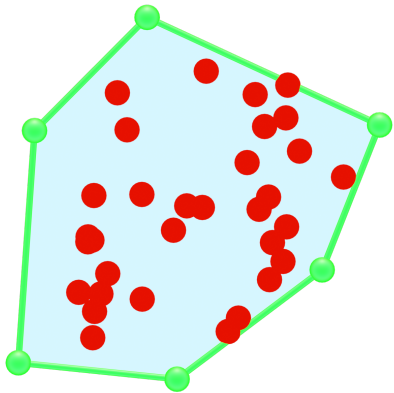
Concave or Convex?



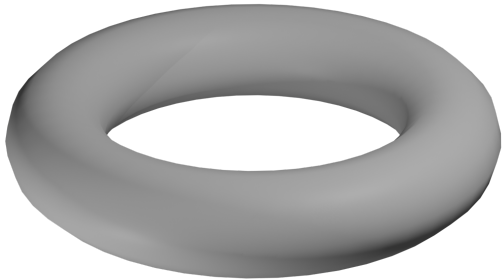
Concave or Convex?



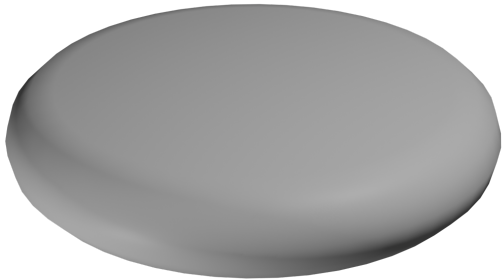
Concave or Convex?



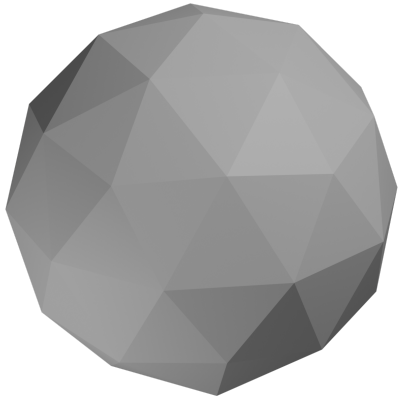
Concave or Convex?



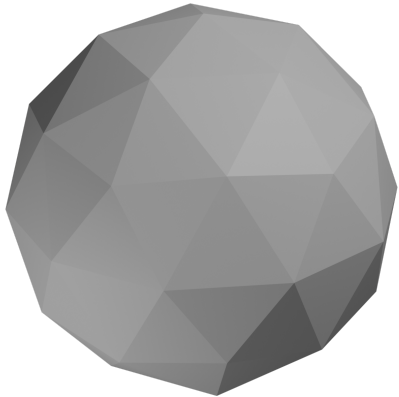
Concave



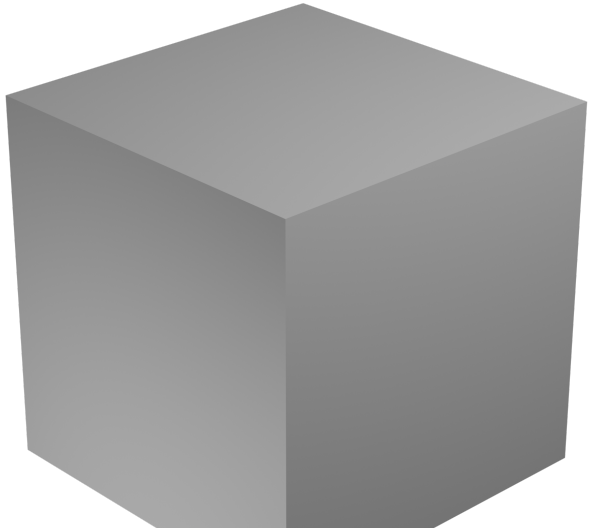
Concave or Convex?



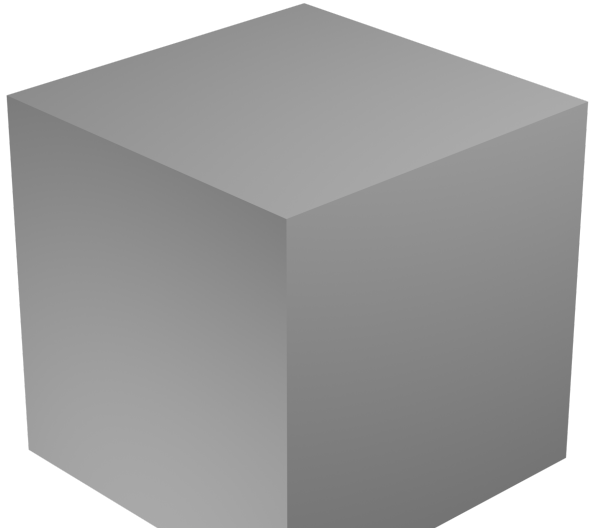
Convex



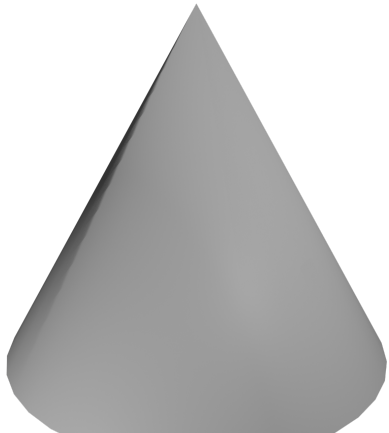
Concave or Convex?



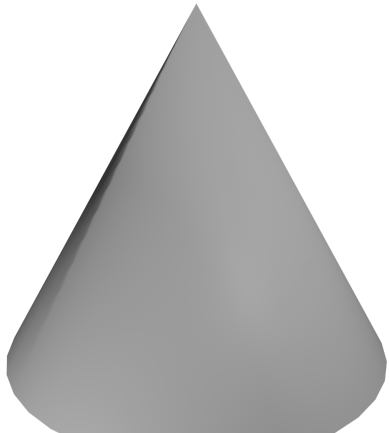
Convex



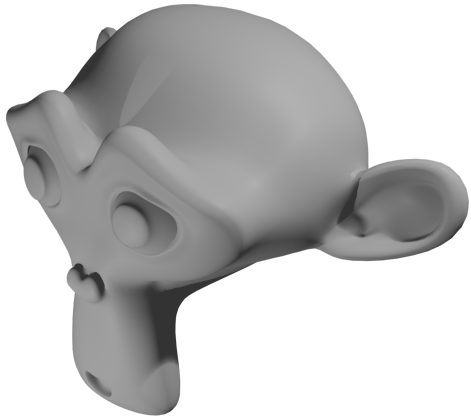
Concave or Convex?



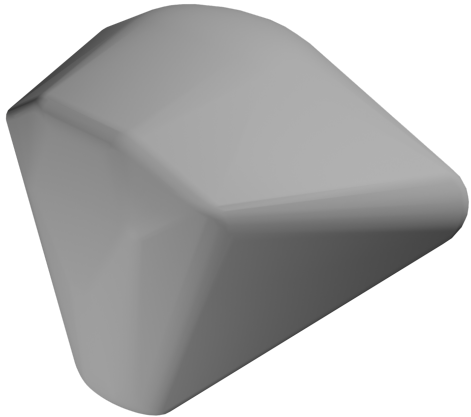
Convex



Concave or Convex?



Concave



Why?

1. What?
2. Why?
3. How?

Convex Hull — Why?

- Often easier (faster) to work with convex objects
  - First pass on convex hulls to quickly exclude objects
  - Then second pass in more detail

Convex Hull — Why?

- Often easier (faster) to work with convex objects
  - First pass on convex hulls to quickly exclude objects
  - Then second pass in more detail
- 3D models (animation and video games), simplify for
  - collision detection
  - physics simulation

Convex Hull — Why?

- Often easier (faster) to work with convex objects
  - First pass on convex hulls to quickly exclude objects
  - Then second pass in more detail
- 3D models (animation and video games), simplify for
  - collision detection
  - physics simulation
- Bézier curve lies within the convex hull of its control points
  - detecting intersections of curves

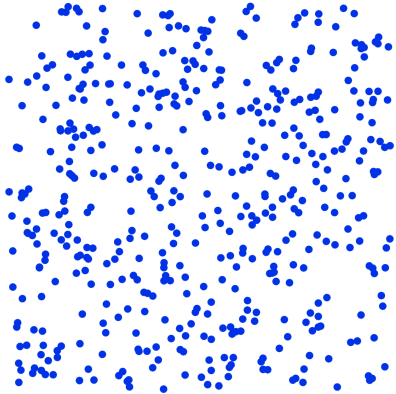
Convex Hull — Why?

- Often easier (faster) to work with convex objects
  - First pass on convex hulls to quickly exclude objects
  - Then second pass in more detail
- 3D models (animation and video games), simplify for
  - collision detection
  - physics simulation
- Bézier curve lies within the convex hull of its control points
  - detecting intersections of curves
- Calculating triangulation of points

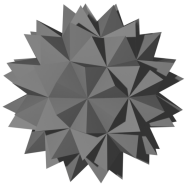
Convex Hull — Why?

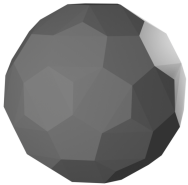
- Often easier (faster) to work with convex objects
  - First pass on convex hulls to quickly exclude objects
  - Then second pass in more detail
- 3D models (animation and video games), simplify for
  - collision detection
  - physics simulation
- Bézier curve lies within the convex hull of its control points
  - detecting intersections of curves
- Calculating triangulation of points
- Optimization

Case in point — triangulation



Case in point — collision







How?

1. What?
2. Why?
3. How?

Convex Hull — How?

Convex Hull — How?

What points are always on the hull?

Convex Hull — How?

The "boundary" points will be in the convex hull

Convex Hull — How?

The "boundary" points will be in the convex hull
Which points will be on the "boundary"?

Convex Hull — How?

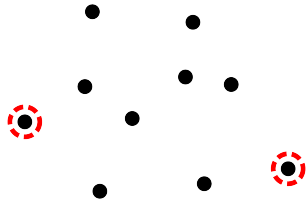
The "boundary" points will be in the convex hull
Which points will be on the "boundary"?

Observation
The points with Min_X and Max_X are both on the boundary of the hull

Convex Hull — How?

The "boundary" points will be in the convex hull
Which points will be on the "boundary"?

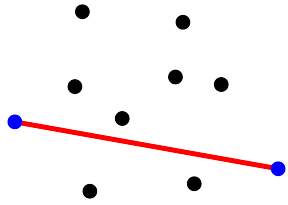
Observation
The points with Min_X and Max_X are both on the boundary of the hull



Convex Hull — How?

The "boundary" points will be in the convex hull
Which points will be on the "boundary"?

Observation
The points with Min_X and Max_X are both on the boundary of the hull

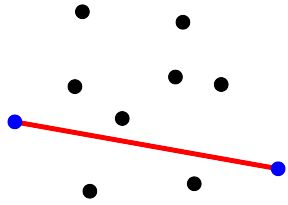


Convex Hull — How?

In-fact, since there is nothing "special" about the *X*-axis, this holds for any line

Observation

For any line, the point(s) furthest from this line is on the boundary of the convex hull

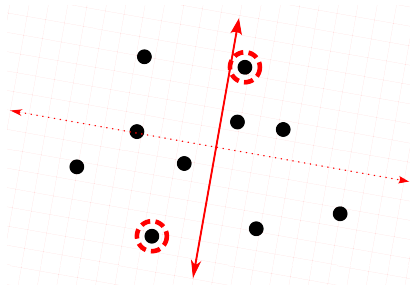


Convex Hull — How?

In-fact, since there is nothing "special" about the *X*-axis, this holds for any line

Observation

For any line, the point(s) furthest from this line is on the boundary of the convex hull



Convex Hull— How?

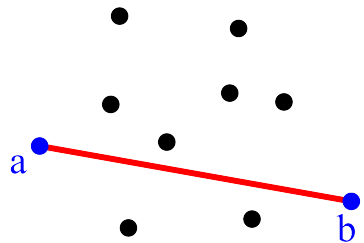
Can you utilize these two facts to create an algorithm?

Convex Hull— An idea

- For any line *ab*, if we rotate this line by 90 degrees we obtain a new line *e*
- The point *c* furthest from this line *e* is on the Convex Hull boundary
  - This gives us two new line segments *ac*, *cb*.

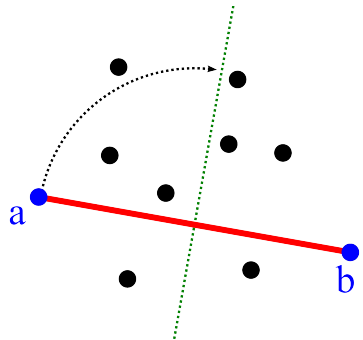
Convex Hull— An idea

- For any line *ab*, if we rotate this line by 90 degrees we obtain a new line *e*
- The point *c* furthest from this line *e* is on the Convex Hull boundary
  - This gives us two new line segments *ac*, *cb*.



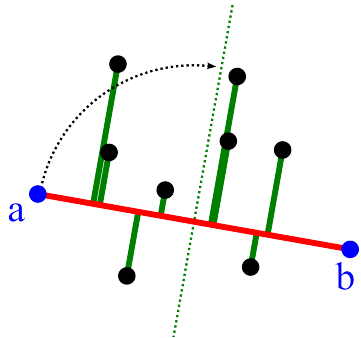
Convex Hull— An idea

- For any line *ab*, if we rotate this line by 90 degrees we obtain a new line *e*
- The point *c* furthest from this line *e* is on the Convex Hull boundary
  - This gives us two new line segments *ac*, *cb*.



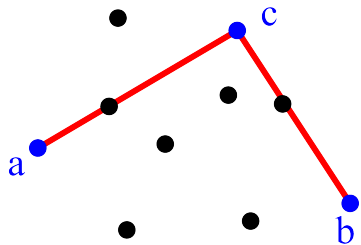
Convex Hull— An idea

- For any line *ab*, if we rotate this line by 90 degrees we obtain a new line *e*
- The point *c* furthest from this line *e* is on the Convex Hull boundary
  - This gives us two new line segments *ac*, *cb*.



Convex Hull— An idea

- For any line *ab*, if we rotate this line by 90 degrees we obtain a new line *e*
- The point *c* furthest from this line *e* is on the Convex Hull boundary
  - This gives us two new line segments *ac*, *cb*.



Convex Hull — How?

What about now, can you make an algorithm?

Convex Hull — How?

QuickHull *P* (unoptimized):

- find leftmost and rightmost points in *P*
- We then recursively find the rest of the points using
  - findHull *P*, *ab* — finding the convex hull points above *ab*
  - findHull *P*, *ba* — finding the convex hull points below *ab*

Convex Hull — How?

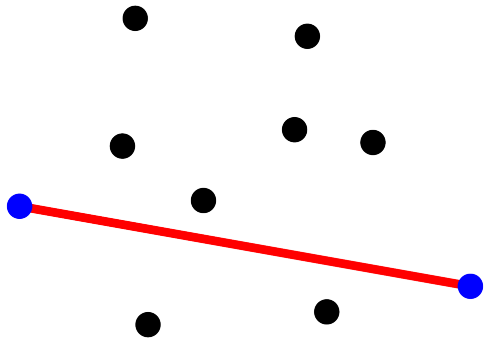
QuickHull *P* (unoptimized):

- find leftmost and rightmost points in *P*
- We then recursively find the rest of the points using
  - findHull *P*, *ab* — finding the convex hull points above *ab*
  - findHull *P*, *ba* — finding the convex hull points below *ab*

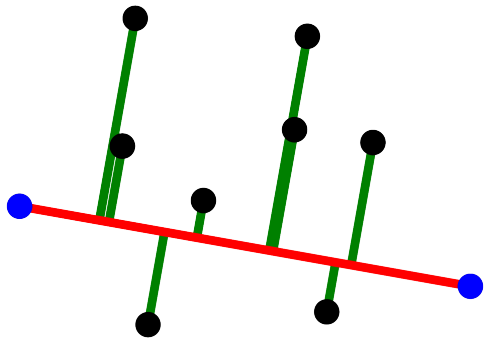
FindHull *P*, *ab* (unoptimized):

- We are adding the points on the convex hull between *a* and *b*
- Find the point *c* furthest above the line *ab*
  - We then get two new lines *ac*, *cb*
  - Recursively call findHull(*P*, *ac*), findHull(*P*, *cb*) on them

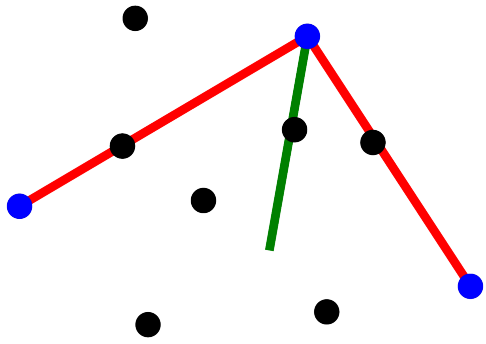
Example execution (unoptimized)



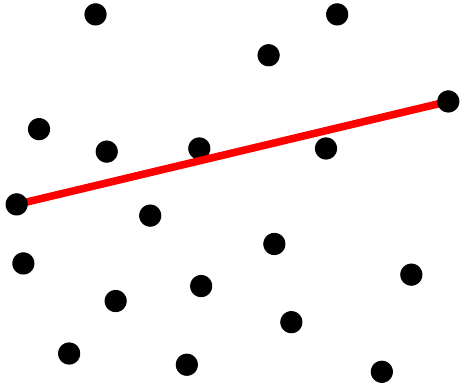
Example execution (unoptimized)



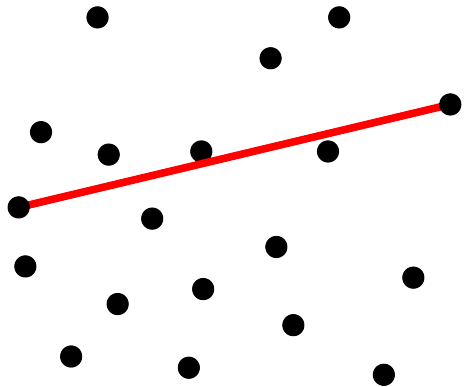
Example execution (unoptimized)



Quick-Hull Optimization

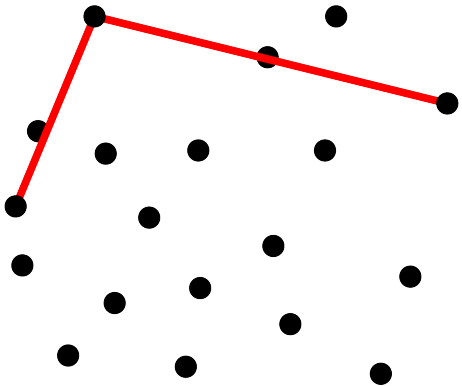


Quick-Hull Optimization



- Find the point *c* furthest above the line *ab*

Quick-Hull Optimization

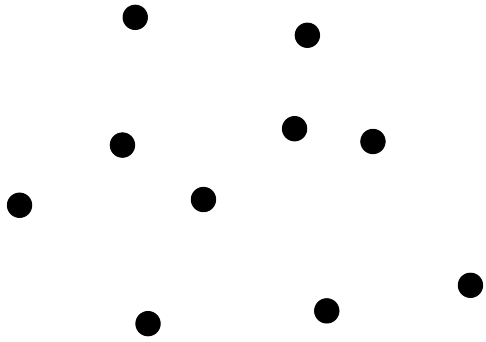


- Find the point *c* furthest above the line *ab*
  - We then get two new lines *ac*, *cb*
  - Recursively call
    - findHull(*P*, *ac*)
    - findHull(*P*, *cb*)

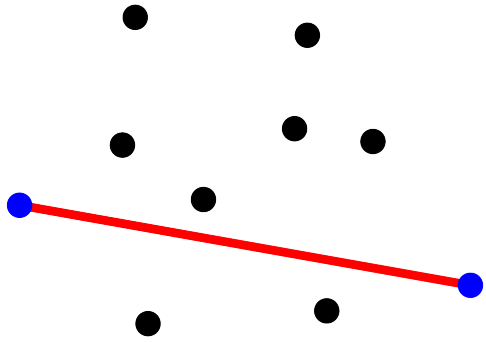
Quick-Hull Optimization

At each line we can discard all points below that line
This cleanly partitions the rest of our points

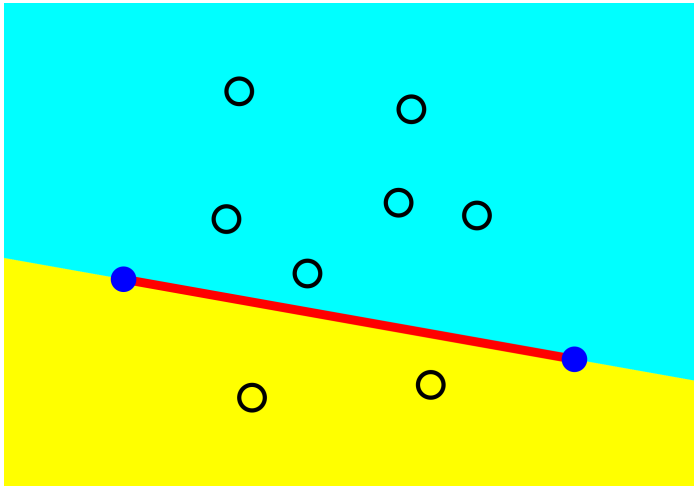
Quick-Hull



Quick-Hull — create line



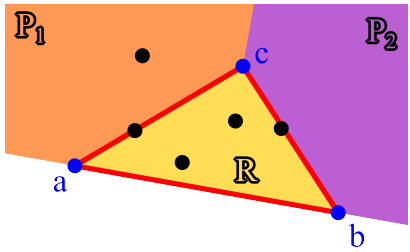
Quick-Hull — call findHull on cyan and yellow



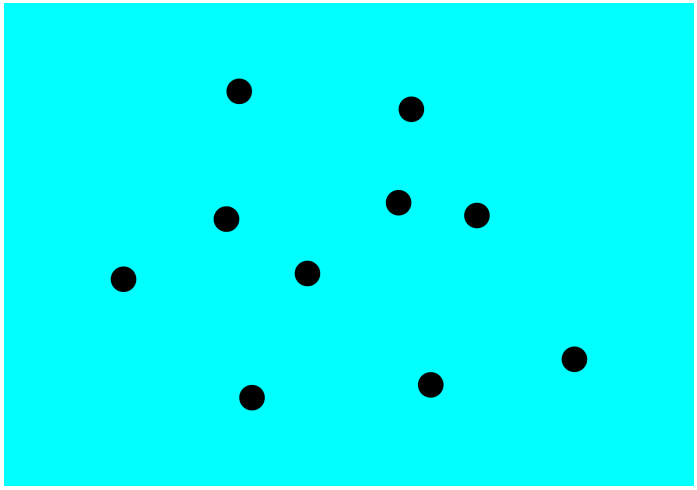
Convex Hull — How?

Find-Hull *ab*:

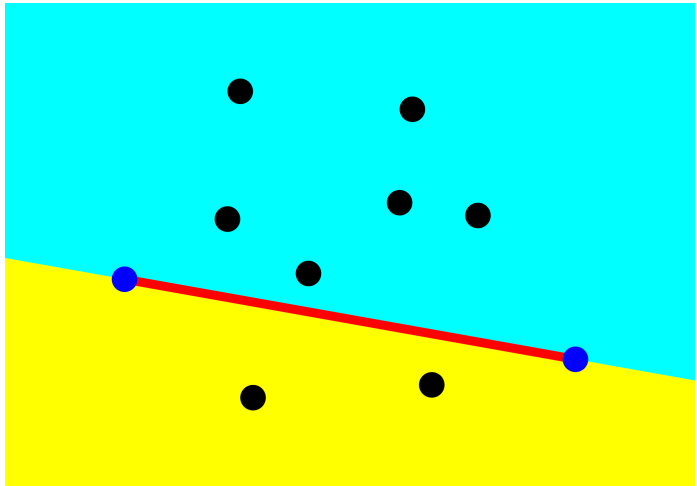
- find point $c \in P$ furthest from line *ab*
- *c* splits (parititions) *P* in three: P_1, P_2, and *R*
  - line segments *ac*, *bc*, and *ba* form a triangle
    - *R*: The points inside the triangle
    - P_1: The points "left" of the triangle
    - P_2: The points "right" of the triangle
- recursively findHull on *ac* in P_1 and *cb* in P_2



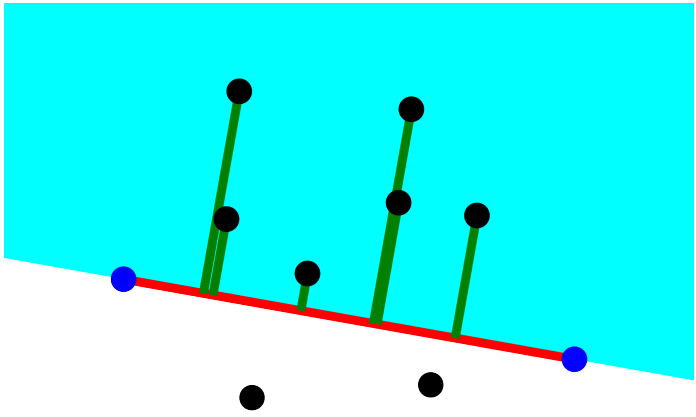
Example – start



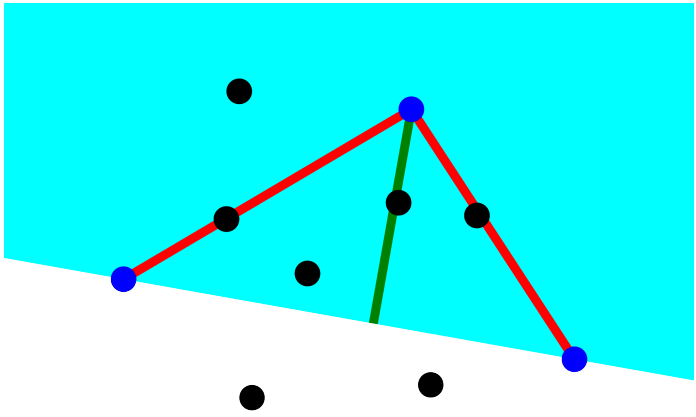
Example – minX maxX



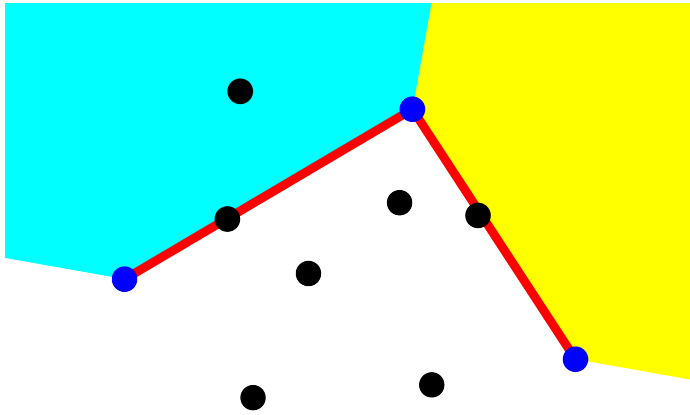
Example – recursion depth = 0



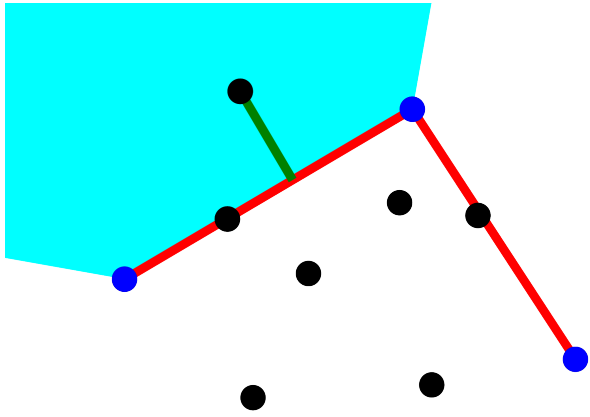
Example – recursion depth = 0



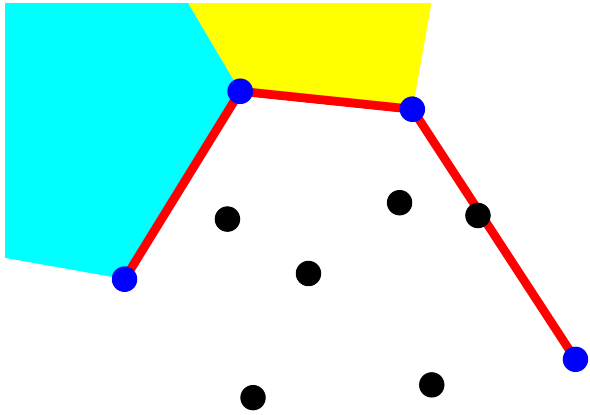
Example – recursion depth = 1



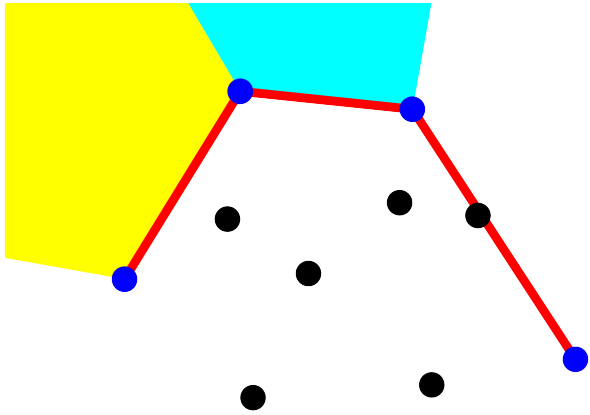
Example – recursion depth = 1



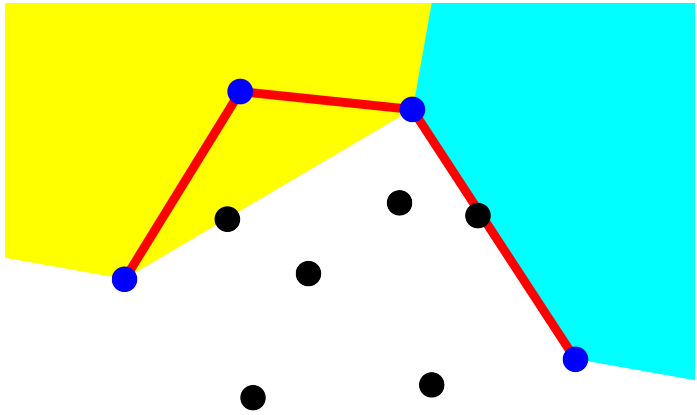
Example – recursion depth = 2



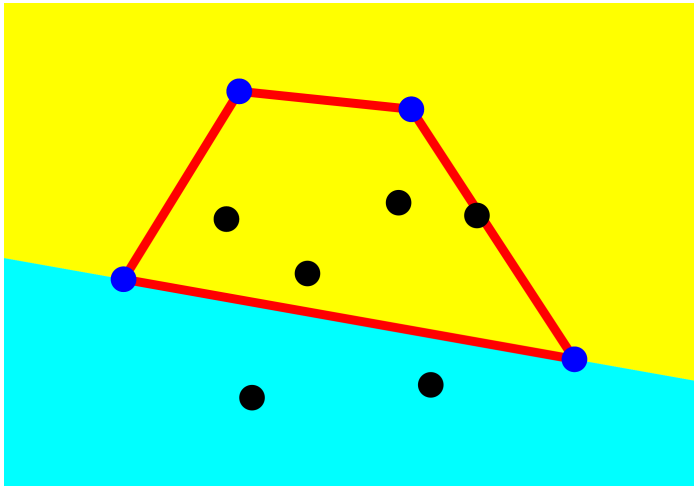
Example – recursion depth = 2



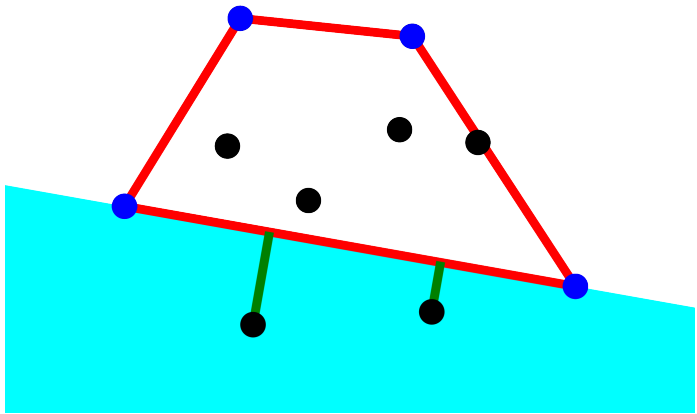
Example – recursion depth = 1



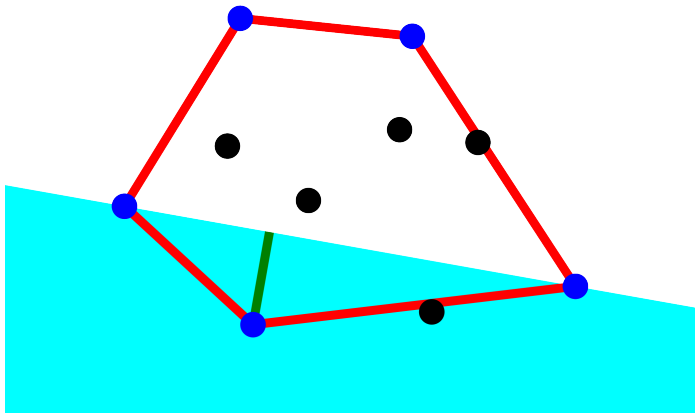
Example – recursion depth = 0



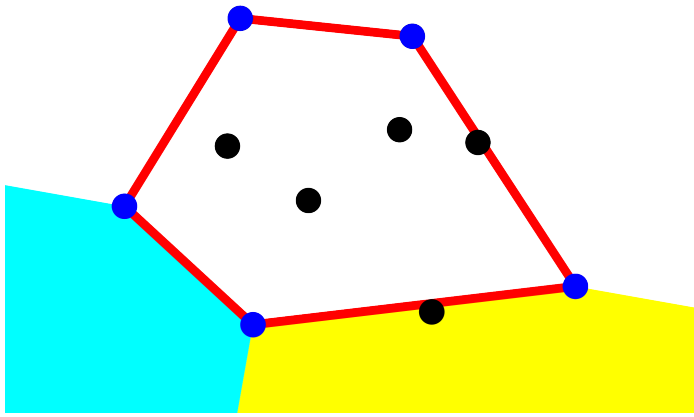
Example – recursion depth = 0



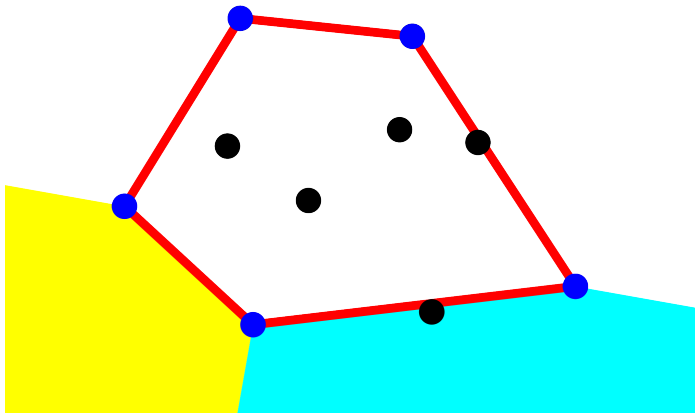
Example – recursion depth = 0



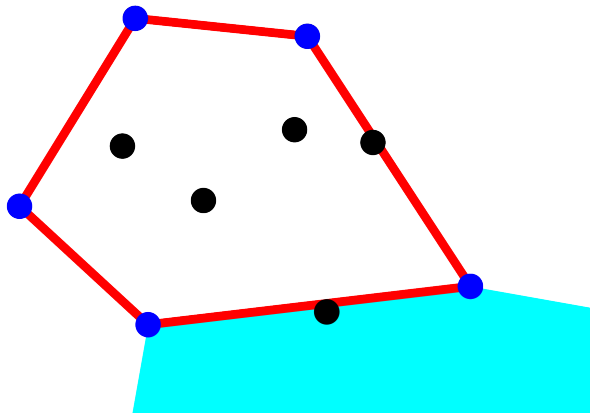
Example – recursion depth = 1



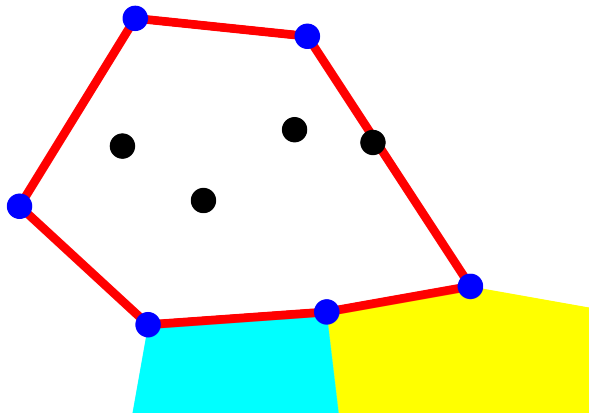
Example – recursion depth = 1



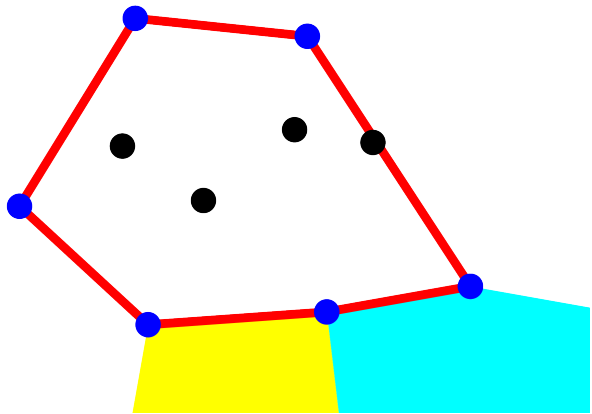
Example – recursion depth = 1



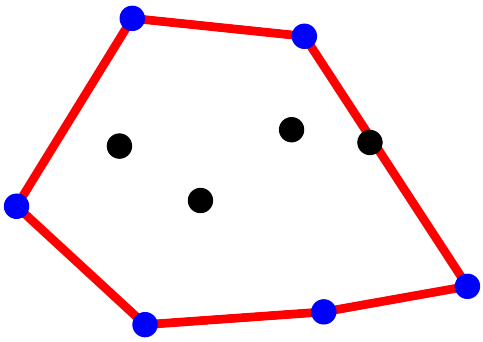
Example – recursion depth = 2



Example – recursion depth = 2



Example – finished



QuickHull

```
1: Function quickHull(S)
2:     a = point in P with min x;
3:     b = point in P with max x;
4:     addToHull(a); addToHull(b);
       // S1 : points above line ab
5:     S1 = pointsToLeft(S, ab);
       // S2 : points below line ab
6:     S2 = pointsToLeft(S, ba);
7:     findHull(S1, a, b);
8:     findHull(S2, b, a);
```

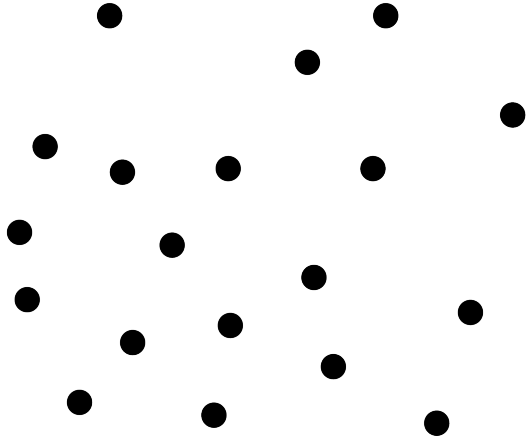
```
1: Function findHull(S, a, b)
2:     if S == ∅ then
3:         return;
4:     c = point in S furthest from ab;
5:     addToHull(c);
6:     S1 = pointsToLeft(S, ac);
7:     S2 = pointsToLeft(S, cb);
8:     findHull(S1, a, c);
9:     findHull(S2, c, b);
```

Utilities

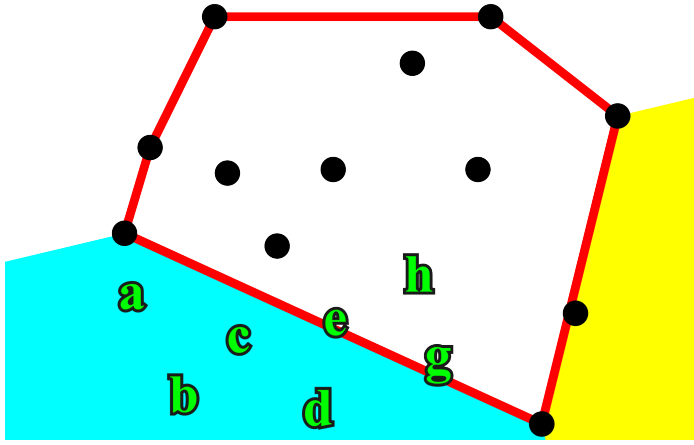
Function ***pointsToLeft****(S, ab)*
- $S_{left} = \{\}$;
- **foreach** $p \in S$ **do**
  - $d =$ ***distanceToLine***(p, ab);
  - **if** $d \leq 0$ **then**
    - $S_{left}.add(p)$;
- **return** S_{left};

Function *distanceToLine(P, P_1P_2)*
- $P = (x, y)$;
- $P_1 = (x_1, y_1)$;
- $P_2 = (x_2, y_2)$;
- $a = y_1 - y_2$;
- $b = x_2 - x_1$;
- $c = x_1 \cdot y_2 - x_2 \cdot y_1$;
- **return** $\frac{a \cdot x + b \cdot y + c}{\sqrt{a^2 + b^2}}$;

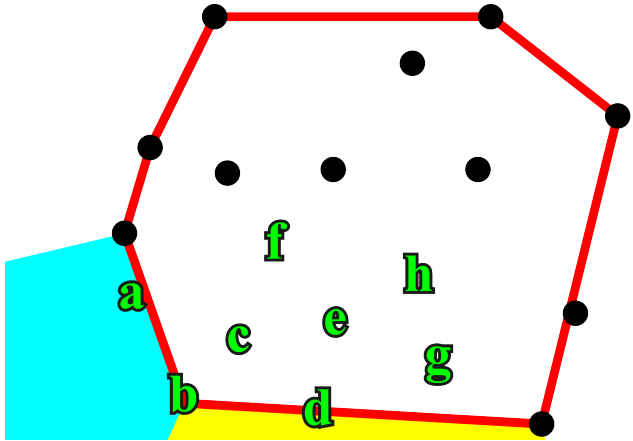
Guess the execution



Guess the execution



Guess the execution



Running time

Running time

worst-case:

Running time

worst-case: $O(n^2)$
average-case:

Running time

worst-case: $O(n^2)$
average-case: $O(n \cdot lg(n))$

- quickHull divides input in roughly two
- findHull divides input in roughly two

Running time

Running time

worst-case:

Running time

worst-case: $O(n^2)$

- If the input almost is a convex hull

Running time

worst-case: $O(n^2)$

- If the input almost is a convex hull

Best-case:

- The input has most points inside the hull, not close to the boundary

A note on implementation

Take note that the order in which the points are added to the hull matters for how we construct the edges.

A note on the name

QuickHull ~ QuickSort?

A note on the name

QuickHull ~ QuickSort?

- Both partition the input in two parts at each step

A note on the name

QuickHull ~ QuickSort?

- Both partition the input in two parts at each step
- Both are average-case $O(n \cdot lg(n))$, worst-case $O(n^2)$

A note on the name

QuickHull ~ QuickSort?

- Both partition the input in two parts at each step
- Both are average-case $O(n \cdot lg(n))$, worst-case $O(n^2)$
- Both are in-place

A note on the name

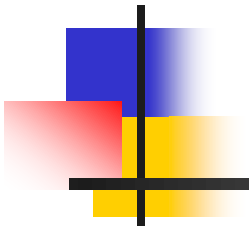
QuickHull $\sim$ QuickSort?

- Both partition the input in two parts at each step
- Both are average-case $O(n \cdot lg(n))$, worst-case $O(n^2)$
- Both are in-place

Implies the existance of ...?

Generalization - 3D?

Generalization - parallel



IN3030/IN4330
Effektiv parallellprogrammering
L01, (Uke 4) våren 2026

Eric Jul
Professor Emeritus
Programmeringsteknologi
Institutt for Informatikk
Universitetet i Oslo
Norge

Recording of Lectures

- **Recording lectures: I will attempt BUT I make no promises – a few times last spring my recording did not work ☹**
- **Lecture slides are mostly in Norwegian (perhaps a little Danish and English intermixed in a few places…)**

Do not be shy...

- **Ask any question at ANY TIME during the lecture – I shall try to respond fairly quickly ☺**
- **Also, REMEMBER, in education:**

THERE IS NO STUPID QUESTION, *only* STUPID ANSWERS!

English

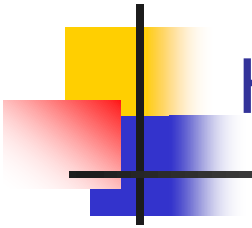
- **I am Danish: so my Danwegian sounds like a terrible dialect of Norwegian – spoken only about 200 km southeast of Kristianssand.**
- **Many Norwegians understand what I say, but, alas, many that have Norwegian as a second language have trouble doing so**
- **And there are a few international students in the course.**
- **So I will lecture in English.**

Hvem

- **Meg**
- **Paulo Ferreira** – professor in PT
- **Michael Kirkedal Thomsen –** førsteamanuensis in PT
- **Brage Isak Keiserås** – Ph.D. student

- **2 Teaching Assistants**

- **About 110 students signed up (usually somewhat fewer show up)**



Hvorfor dette kurs?

- **Love of *Learning by Doing***
- **Oppleve progammer, der gjør en forskjell!!**
- **Uttnytte *multicore – multikjerne***
- ***Lage cache-friendly programs***
- ***Teori er bra, MEN praksis essentiel!***
- ***Bli bedre programmør!!***
- ***… And have fun doing so!!!***

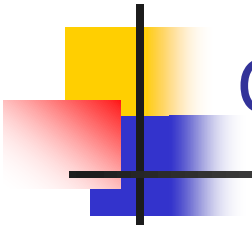
Litt om Eric

- **Ph.D. University of Washington, 1989**
- **Dansk-amerikaner**
- **Bor i Danmark – pendler til Oslo ca 1 gange/måned**
- **1989-2000: Førsteamauensis Københavns Universitet**
- **2000-2009: Professor Københavns Universitet**
- **2009-2015: Bell Labs, Dublin & Professor II, IfI**
- **2016-2025 Professor IfI**
- **2019-2024 Gruppeleder *Programming Technology***
- **2025- : Professofr Emeritus IfI**

- ***Favorite hobbies: Skiing and Gliding (Seilfly) and a little bicycling***

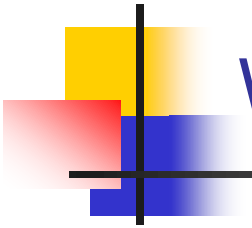
Litt mere om Eric





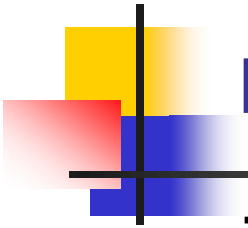
Communication

- **Please keep up with the messages on the web site: The course website is THE source of information**
- **Use the Hjeplelærer ☺**
- **Ask questions when in doubt**
- **IN4330 students – use the IN3030 web site**



Why are YOU here??

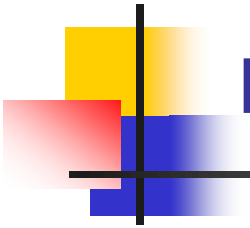
- **What do you expect from the course?**



Let's get started on parallelism

Two examples:

- **Plant a tree**
  - **Embarrassingly paralizable!**
- **Make a baby**
  - **Inherently sequential!**



Motivation for Parallelle Programmer

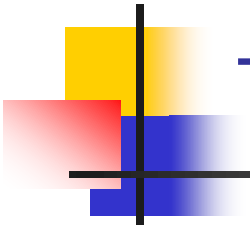
- **Maskiner kan bestå av flere CPU-er («cores»)**
  - Hver CPU kan utføre programmer uavhengig av de andre CPUer
- **Hver CPU kan have flere kjerner**
  - Hver kjerne kan utføre programmer
- Vi vil gjerne utnytte disse muligheter for parallellisme

Hva vi skal lære om i dette kurset:

Lage parallelle programmer (algoritmer) som er:

- **Riktige**
  - Parallelle programmer er klart vanskeligere å lage enn sekvensielle løsninger på et problem.
- **Effektive**
  - dvs. raskere enn en sekvensiell løsning på samme problem

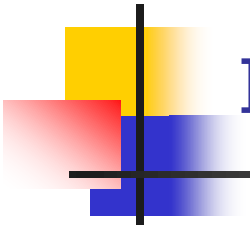
- Lære hvordan man parallelliserer et riktig, sekvensielt program + lage egne parallelle algoritmer som ikke bare er en slik parallellisering;
- Lære om caching – og dets store effekt på programmer
- Lære de mange problemene vi støter på og hvordan disse kan takles.
- Kurset er **empirisk** (med tidsmålinger), ikke bare basert på en teoretisk *modell* av parallelle beregninger,
- Presenterer en klassifikasjon av parallelle algoritmer.



Tre grunner til å lage parallelle programmer

1) Skille ut aktiviteter som går **langsommere** i en egen tråd.
   - Eks: Tegne grafikk på skjermen, lese i databasen, sende melding på nettet. Asynkron kommunikasjon.
2) Av og til er det **lettere** å programmere løsningen som flere parallelle tråder. Naturlig oppdeling.
   - Eks: Kundesystem over nettet hvor hver bruker får en tråd.
   - Hele operativsystemet har mye parallellitet – cirka 6000 aktive tråder i Mac OS X Sequoia.
3) Vi ønsker **raskere** programmer, raskere algoritmer.
   - Eks: Tekniske beregninger, søking og sortering.

Dette kurset legger nesten all vekt på *raskere* algoritmer



IN3030/IN4330

- Et relativt UNIKT kurs – set internationalt.
- Planlagt fem obliger - ca. 1-4 uker per oblig.
  - De individuelle innleveringer : Man kan samarbeide om algoritmer, men **ikke** ha helt lik eller delvis felles kode med andre. Rapport skal skrives selv.
- En oblig er ikke bare innlevering av ett eller flere parallelle programmer, men **også en liten rapport** om de testene man har gjort: hastighetsmålinger på disse for ulike størrelse av data med konklusjoner – f.eks speedup + O( ) og en forklaring på resultatene .
- Gruppetimer:
  - Jobbing med ukeoppgaver og obligene

Pensum

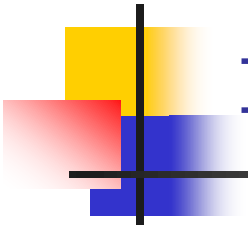
- Ingen dekkende lærebok er funnet, men:
  - **Kompendium av Arne Maus:** legges opp på web
  - **Det som foreleses (Powerpoint slides) + oppgavene (obliger + ukeoppgaver) er pensum.**

  - **Bra bok:** Brian Goetz, T.Perlis, J. Bloch, J. Bobeer, D. Holms og Doug Lea::"Java Concurrency in practice", Addison Wesley 2006
  - Kap. **18 og 19** i A. Brunland, K. Hegna, O.C. Lingjærde, A. Maus:"Rett på Java" 3.utg. Universitetsforlaget, 2011.
  - I tillegg leses *fra en maskin på Ifi* kap 1 til 1.4, hele 2 og 3.1 til 3.7 (hopp over programeksemplene) i :



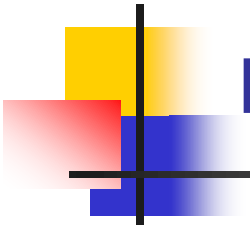
(Hvis leses utenfor Ifi, så koster det $!)

I dag – teori og praksis



- What is in this course?
- Reasons for multi-core machines – and parallel programs
- Parallel program solutions to a given problem are often much more complicated and require more code than a sequential solution
- Main reason for parallel programs: They go *FASTER* because they utilize all available cores



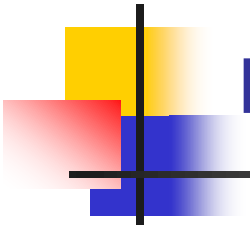


Main Performance goal: speedup!

- In this course, the main goal is to rewrite a sequential program into a corresponding parallel program that runs *faster*
- The main **metric** is speedup:

Speedup = time(sequential) / time(parallel)

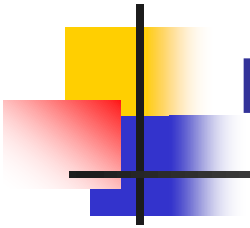
Time is *wall-clock time*



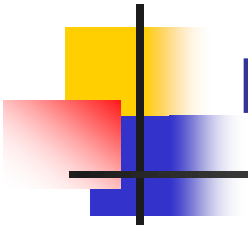
Lineær speedup ?

- Selvsagt ønsker vi lineær speedup – dvs. bruker vi *k* kjerner skulle det helst gå *k* ganger fortere enn med 1 kjerne.
- Meget sjelden at det kan oppnås (mer om det siden)
- Kan superlineær speedup oppnås?
  - *PLEASE THINK ABOUT THIS!*
- Speedup: a central metric.

Lineær speedup analogier

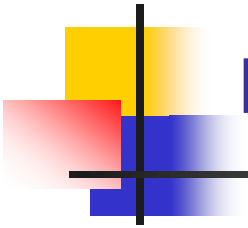


- Selvsagt ønsker vi lineær speedup, MEN:
- Noen problem er **nemme** at parallellisere: Eksempel: 10 personer kan plante 10 treer ca 10 gange fortere end 1 person kan plante 10 treer.
- Andre problemer er *vanskeligere* at parallellisere: Eksempel: hvis 1 person kan samle et Lego-set på 10 timer, så vil det være vanskelig for 10 personer at samle det på 1 time – der er avhengigheter mellom delene.
- Andre problemer er nærmest ***umulig*** at parallellisere: Eksempel: hvis 1 kvinne kan lage et barn på 9 måneder, kan 9 kvinner så lage et barn på 1 måned?



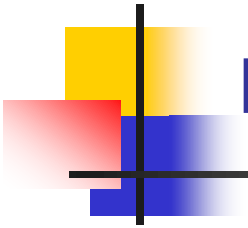
Dette kurset handler om multikjernemaskiner

- Nå har vi multikjerne maskiner!
  - Din 2500 NOK bærbar: 4-8 kjerner
  - Min 2023 MacBook Pro: 19 kjerner
  - Stor server: 64-512 kjerner
- Nå har vi maskiner med multi-level caches
- Hvordan uttnytter vi dem??
- Svar: vi bruker tråde!



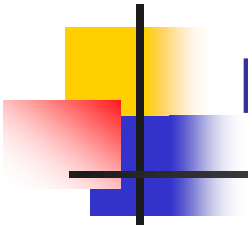
Dette kurset handler om tråder og effektivitet

- Hva er tråder
  - Se litt på maskinen
  - Se på operativsystemet
  - Hvordan skal vi oppfatte en tråd i et Java-program
  - Senere se på kompileringen og kjøring av Java-kode
- Hvordan måle effektivitet
  - Hvordan ta tiden på ulike deler av et program; både:
    - Den sekvensielle algoritmen
    - En eller flere parallelle løsninger
  - I dag: Enkel tidtaking
  - Neste gang: Bedre tidtaking
- Praktisk i dag
  - Standard måte å starte programmet med tråder



Dette kurset handler også om caching

- Caching:
  - Why have caching?
  - What is the effect on our programs?
  - How do we make cache-friendly programs?

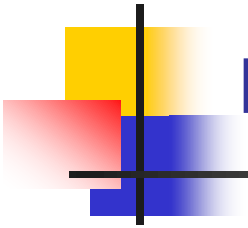


Flere mulige synsvinkler

Mange nivåer i parallellprogrammering:

1. Maskinvare
2. Programmeringsspråk
3. Programmeringsabstraksjon
4. Hvilke typer problem egner seg for parallelle løsninger?
5. Empiriske eller formelle metoder for parallelle beregninger

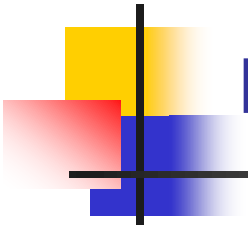
IN3030/IN4330: Parallellprogrammering av ulike algoritmer med tråder på multikjerne CPU i Java – empirisk vurdert ved tidsmålinger.



Learning-by-doing

Dette kurs er et Learning-by-doing kurs!

Utbyttet **ditt** er avhengig av **DIN** innsats!



DIN innsats

Dette kurs er et Learning-by-doing kurs!

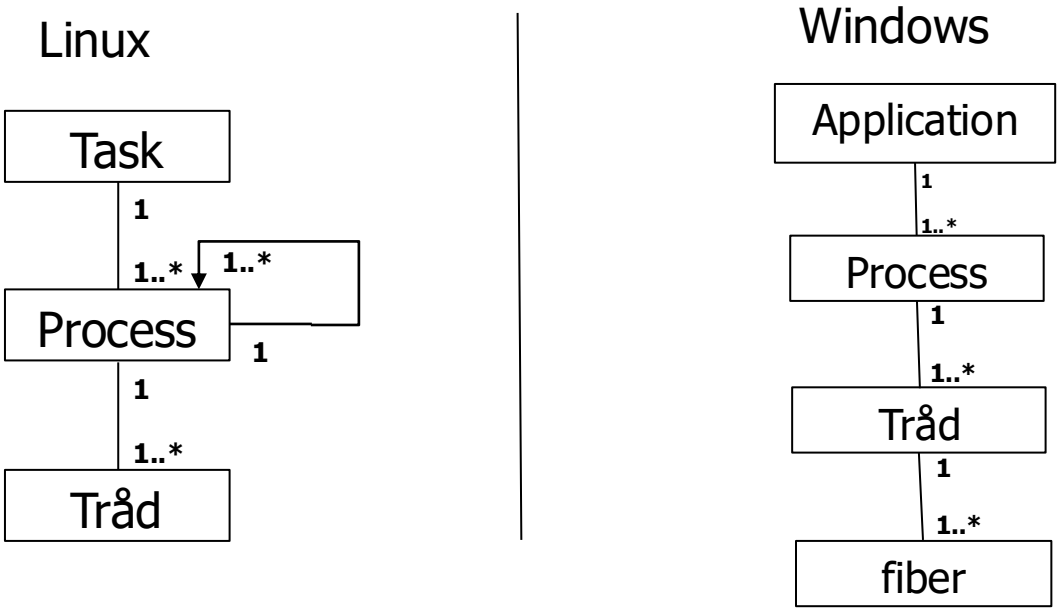
Go program lots of parallel programs

AND

make them ***FAST***

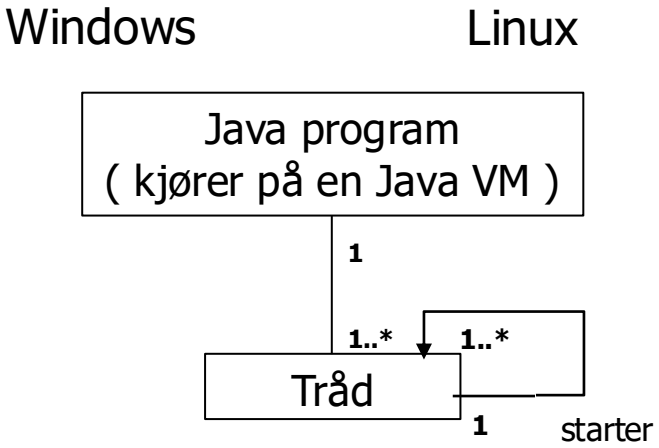
Operativsystemet og tråder

- De ulike operativsystemene (Linux, Windows) har ulike begreper for det som kjøres; mange nivåer (egentlig flere enn det som vises her)

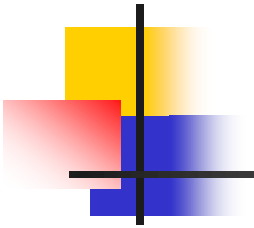


Heldigvis forenkler Java dette

Java forenkler dette ved å velge to nivåer



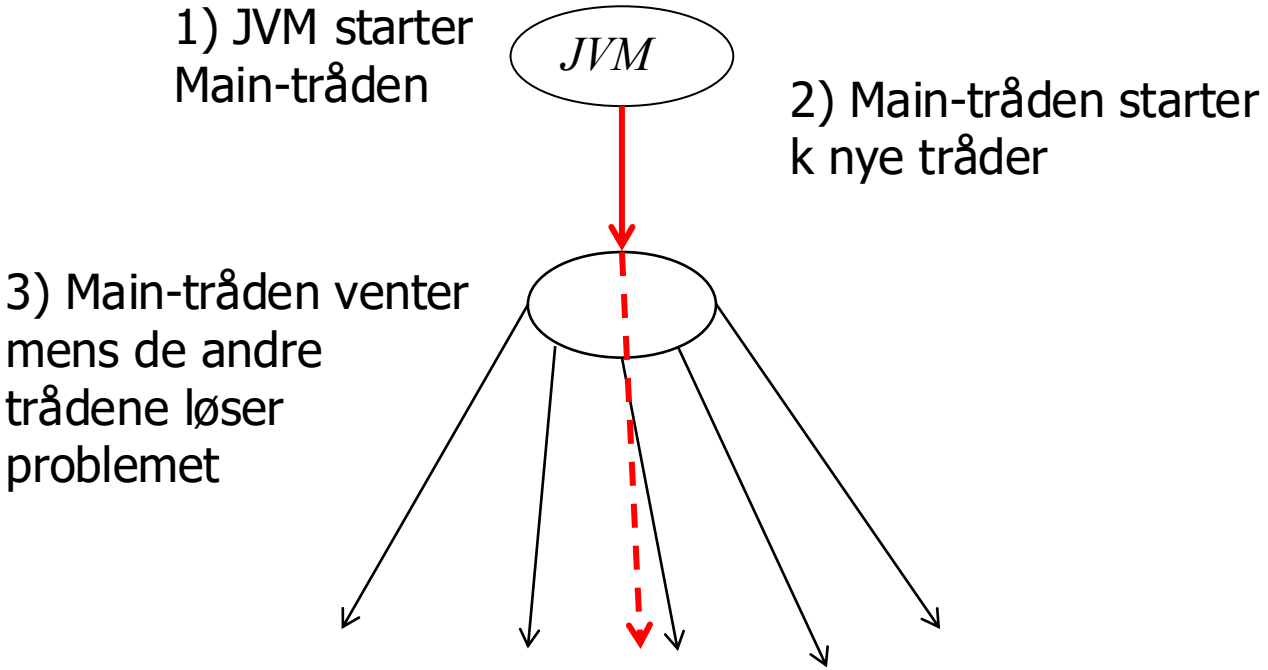
- **Alle trådene i et Java-program deler samme adresserom** (= samme plasser i hovedhukommelsen). Alle trådene kan lese og skrive i de variable (objektene) programmet har og ha adgang til samme kode (metodene i klassene).



Hva er tråder i Java ?

- I alle programmer kjører minst en tråd – main tråden (starter og kjører i `public static void main`).
- Main-tråden kan starte en eller flere andre, nye tråder.
- Enhver tråd som er startet, kan stoppes midlertidig eller permanent av:
  - Av seg selv ved kall på synkroniseringsobjekter hvor den må vente
  - Den er ferdig med sin kode (i metoden run), terminerer da
- Main-tråden og de nye trådene går i parallell ved at:
  - De kjører enten på hver sin kjerne
  - Hvis vi har flere tråder enn kjerner, vil klokka i maskinen sørge for at trådene av og til avbrytes og en annen tråd får kjøretid på kjernen.
- Vi bruker tråder til å parallellisere programmene våre

>java (også kalt JVM) starter main-tråden som igjen starter nye tråder



Tråder i Java er objekter av klassen Thread.

Konstruktør til Thread-klassen

Thread

`public Thread(Runnable target)`

Allocates a new `Thread` object. This constructor has the same effect as `Thread (null, target, gname)`, where `gname` is a newly generated name. Automatically generated names are of the form `"Thread-"`+*n*, where *n* is an integer.

Parameters:

`target` - the object whose `run` method is invoked when this thread is started. If `null`, this classes `run` method does nothing.

- **`Runnable target`** er:
  - En klasse som implementerer grensesnittet 'Runnable'
- Det er en annen måte å starte er tråd hvor vi lager en subklasse av Thread (ikke fullt så fleksibel).

Tråder i Java

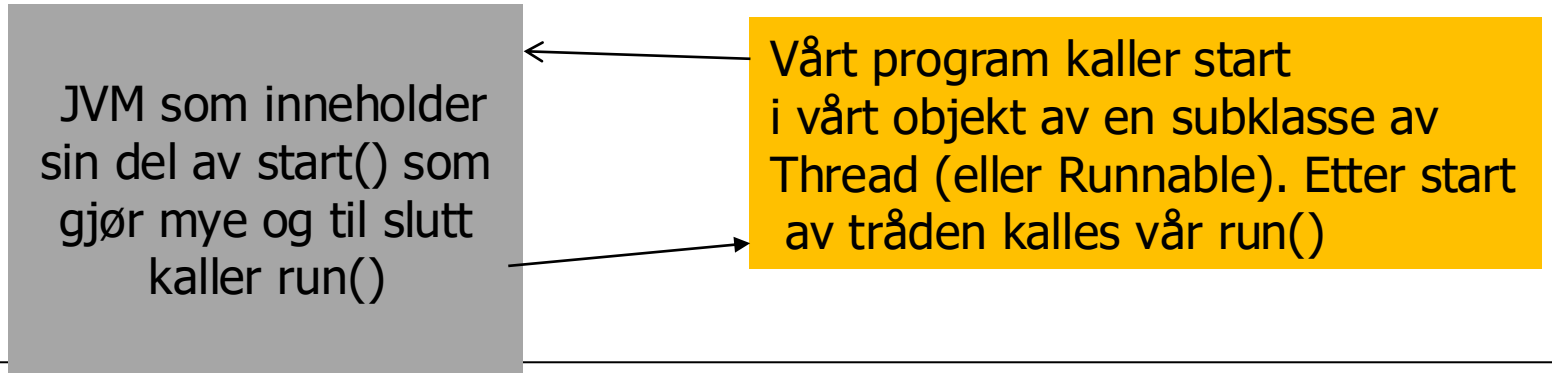
- Er én programflyt, dvs. en serie med instruksjoner som oppfører seg som ett vanlig, sekvensielt program – og kjører på én kjerne
- Det kan godt være (langt) flere tråder enn det er kjerner.
- En tråd er ofte implementert i form av en indre klasse i den klassen som løser problemet vårt (da får trådene greit aksess til **felles data**):

```
import java.util.concurrent.*;
class Problem { int [] fellesData ; // dette er felles, delte data for alle trådene
        public static void main(String [] args) {
                Problem p = new Problem();
                        p.utfoer();
                }
                void utfoer () { Thread  t = new Thread(new Arbeider());
                    t.start();
                 }

            class Arbeider implements Runnable {
                int i,lokalData; // dette er lokale data for hver tråd
                    public void run() {
                            // denne kalles når tråden er startet
                    }
             } // end indre klasse Arbeider
} // end class Problem
```

Tråder i Java

- En tråd er enten subklasse av Thread eller får til sin konstruktør et objekt av en klasse som implementerer Runnable.
- Poenget er at begge måtene inneholder en metode:
  - `'public void run()'`
- Vi kaller metoden `start()` i klassen `Thread`. Det sørger for at JVM starter tråden og at 'run()' i vår klasse deretter kalles.



Konstruktør til Thread-klassen

Thread

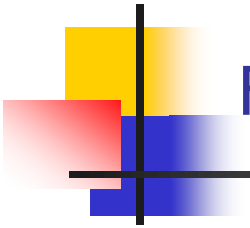
`public Thread(Runnable target)`

Allocates a new `Thread` object. This constructor has the same effect as `Thread (null, target, gname)`, where `gname` is a newly generated name. Automatically generated names are of the form `"Thread-"`+*n*, where *n* is an integer.

Parameters:

`target` - the object whose `run` method is invoked when this thread is started. If `null`, this classes `run` method does nothing.

- **`Runnable target`** er:
  - En klasse som implementerer grensesnittet 'Runnable'
- Det er en annen måte å starte er tråd hvor vi lager en subklasse av Thread (ikke fullt så fleksibel).



Flere problemer med parallellitet og tråder i Java

1. Operasjoner blandes (oppdateringer går tapt).
2. Oppdaterte verdier til felles data er ikke alltid synlig fra alle tråder (oppdateringer er ikke synlige når du trenger dem).
3. Synlighet har ofte med cache å gjøre.
4. The Java memory model (= hva skjer 'egentlig' når du kjører et Java-program).

- Vi må finne på 'skuddsikre' måter å programmere parallelle programmer
  - De er kanskje ikke helt tidsoptimale
  - Men de er lettere å bruke !!
  - Det er vanskelig nok likevel.
- Bare oversiktelige, 'enkle' måter å programmere parallelt er mulig i praksis

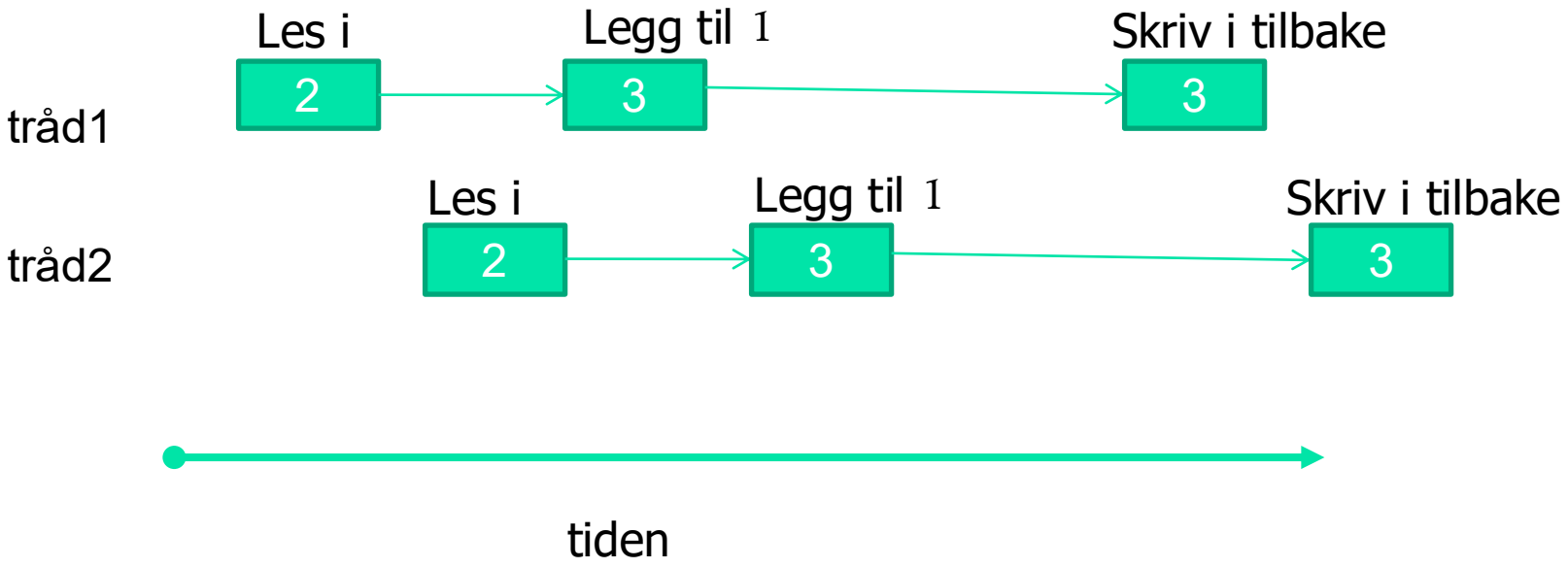
Example: Concurrent update of a variable Let's try! (Spoiler: we will fail!)

```
import java.util.concurrent.*;
class Problem { int i ; // dette er felles, delte data for alle trådene
        public static void main(String [] args) {
                Problem p = new Problem();
                        p.utfoer();
                }
                void utfoer () {
                    int j, k;
                    j=10;
                    for (k=0; k< j; k++)  {
                        new Thread(new Arbeider()).start();
                 }

          class Arbeider implements Runnable {
               int i,lokalData; // dette er lokale data for hver tråd
                  public void run() { // denne kalles når tråden er startet
                        i++;
                  }
           } // end indre klasse Arbeider
} // end class Problem
```

1) Ett problem i dag: operasjoner blandes ved samtidige oppdateringer

- Samtidig oppdatering - flere tråder sier gjentatte ganger: **i++ ;** der i er en felles int.
  - **i++** er 3 operasjoner: a) les i, b) legg til 1, c) skriv i tilbake
  - Anta i =2, og to tråder gjør i++
  - Vi kan få svaret 3 eller 4 (skulle fått 4!)
  - Dette skjer i praksis !



Test på i++; parallell

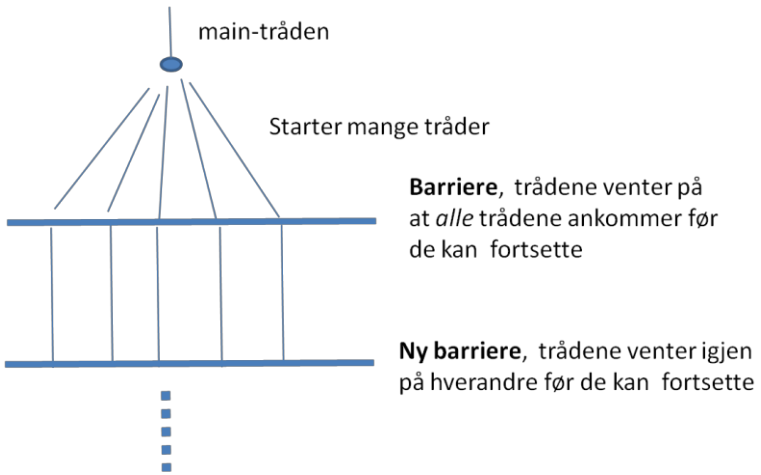
- Setter i gang **n tråder** (på en 2-kjerner CPU) som alle prøver å øke med 1 en felles variabel int i; 100 000 ganger uten synkronisering;

```
for (int j =0; j< 100000; j++) {
            i++;
}
```

- Vi fikk følgende feil - antall og %, (manglende verdier).
  Merk: Resultatene *varierer også mye* mellom hver kjøring :

Antall tråder n		**1**	**2**	**20**	**200**	**2000**
Svar	1.gang	100 000	200000	1290279	16940111	170127199
	2.gang	100 000	159234	1706068	16459210	164954894
Tap	1.gang	0 %	0%	35,5%	15,3%	14,9%
	2. gang	0%	20,4%	14,6%	17,7%	17,5%

Kommende program bruker CyclicBarrier. Hva gjør den?

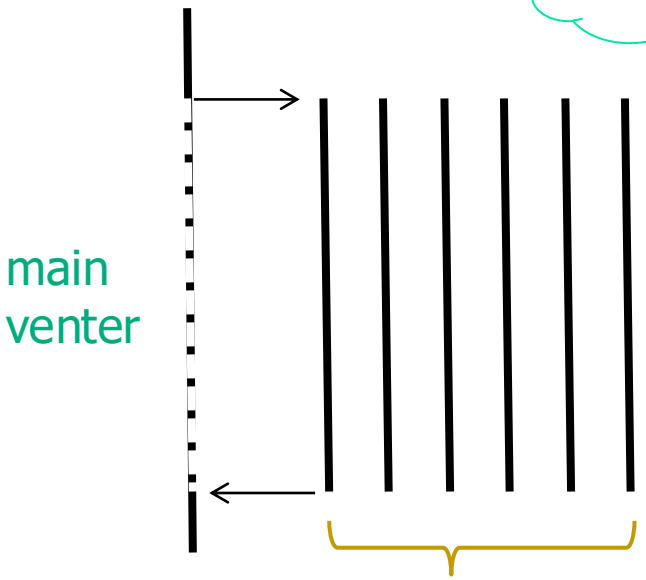


- Man lager først ett, felles objekt `b` av klassen CyclicBarrier med et tall: `ant` til konstruktøren = det antall tråder den skal køe opp før alle trådene slippes fri 'samtidig'.
- Tråder (også main-tråden) som vil køe opp på en CyclicBarrier sier await() på den.
- De `ant-1` første trådene som sier await(), blir lagt i en kø.
- Når tråd nummer `ant` sier await() på `b`, blir alle trådene sluppet ut av køen 'samtidig' og fortsetter i sin kode.
- Det sykliske barriere objektet `b` er da med en gang klar til å være kø for nye, `ant` stk. tråder.

Husk: Vanligste oppsett av main-tråden + k tråder

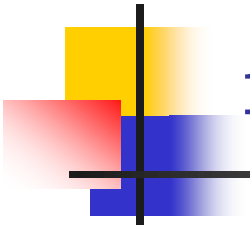
main, lager k nye tråder





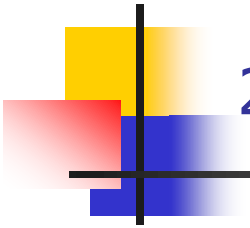
k tråder, leser og skriver i egne og i felles data og løser problemet

Hver av trådene (main + k nye) er sekvensielle programmer.
Problemet er at de samtidig *ikke kan skrive* på felles data



1) Avslutning med en CyclicBarrier

- En CyclicBarrier (cb= new CyclicBarrier (n+1))
  - Er tenkt som et ventested, en bom/grind for et antall (i dette tilfellet for n+1) tråder - de n 'nye' trådene + main. Alle må vente når de sier sier cb.await() til sistemann ankommer køen, og **da** kan alle fortsette.
  - Trådene kan da være ferdige med en beregning kan selv avslutte med å bli ferdige med sin run() -kode. Main-tråden forsetter, og vet at de andre trådene er ferdige. Main-tråden kan da bruke resultatene fra trådene.
  - Den sykliske barrieren cb er da strakt klar til å køe nye n tråder som sier cb.await() , .. osv
  - cb.await() sies inne i en try-catch blokk



2) Avslutning med en Semaphore

- En Semaphore (sf = new Semaphore(-n+1))
  - Administrerer (i dette tilfellet) –n+1 stk. **tillatelser**.
  - To sentrale primitiver:
    - sf.acquire() – ber om **en** tillatelse. Antall tillatelser i sf blir da 1 mindre hvis antallet er >0. Hvis det ikke er noen ledig tillatelse, må tråden vente i en kø (inne i en try-catch blokk)
    - sf.release() – gir **én** tillatelse tilbake til semaforen sf. Ikke try-catch blokk (Den tillatelsen som gis tilbake behøver ikke vært 'fått' ved hjelp av aquire() ; den er bare et tall).
  - Avlutning med Semaphore sf:
    - Maintråden sier sf.acquire() – og må vente på at det er minst en tillatelse i sf.
    - Alle de n nye trådene sier sf.release() når de terminerer, og når den siste sier sf.release() blir det 1 tillatelse ledig og main fortsetter.
    - Ikke syklisk.

3) Avslutning med join() – bare EN gang

- Logikken er her at i den rutinen hvor alle trådene lages, legges de også inn i en array. Main-tråden legger seg til å vente på den tråden som den har peker til skal terminere selv.
  Venter på alle trådene etter tur at de terminerer:

```
 // main –tråden i konstruktøren
Thread [] t = new Thread[n];
for (int i = 0; i < n; i++) {
   t[i] = new Thread (new Arbeider(..));
   t[i].start();
 }
  ……….
  // main vil vente her til trådene er ferdige
for(int i = 0; i < n; i++) {
       try{ t[i].join();
        }catch (Exception e){return;};
}     ……..……..
```

II) Mange ulike synkroniserings primitiver
Vi skal bare lære noen få !

- java.util.concurrent

Classes

AbstractExecutorService
ArrayBlockingQueue
ConcurrentHashMap
ConcurrentLinkedDeque
ConcurrentLinkedQueue
ConcurrentSkipListMap
ConcurrentSkipListSet
CopyOnWriteArrayList
CopyOnWriteArraySet
CountDownLatch
CyclicBarrier
DelayQueue
Exchanger
ExecutorCompletionService
ecutor
eadPoolExecutor
ThreadPoolExecutor.AbortPolicy
ThreadPoolExecutor.CallerRunsPolicy
ThreadPoolExecutor.DiscardOldestPolicy
ThreadPoolExecutor.DiscardPolicy

Semaphore
SynchronousQueue
ThreadLocalRandom
ThrExecutors
ForkJoinPool
ForkJoinTask
ForkJoinWorkerThread
FutureTask
LinkedBlockingDeque
LinkedBlockingQueue
LinkedTransferQueue
Phaser
PriorityBlockingQueue
RecursiveAction
RecursiveTask
ScheduledThreadPoolEx

Interfaces

BlockingDeque
BlockingQueue
Callable
CompletionService
ConcurrentMap
ConcurrentNavigableMap
Delayed
Executor
ExecutorService
ForkJoinPool.ForkJoinWorkerThreadFacto
ry
ForkJoinPool.ManagedBlocker
Future
RejectedExecutionHandler
RunnableFuture
RunnableScheduledFuture
ScheduledExecutorService
ScheduledFuture
ThreadFactory
TransferQueue

java.util.concurrent.atomic

De har samme virkning (semantikk) som volatile variable (forklares senere), men kan gjøre mer sammensatte operasjoner. Mye raskere enn sychronized methods.

Classes

AtomicBoolean
AtomicInteger
AtomicIntegerArray
AtomicIntegerFieldUpdater
AtomicLong
AtomicLongArray
AtomicLongFieldUpdater
AtomicMarkableReference
AtomicReference
AtomicReferenceArray
AtomicReferenceFieldUpdater
AtomicStampedReference

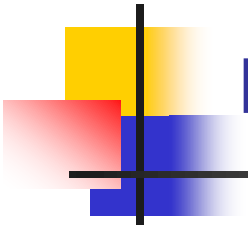
Eksempel på operasjoner i **AtomicIntegerArray:**

int	**get**(int i) Gets the current value at position i.
int	**getAndAdd**(int i, int delta) Atomically adds the given value to the element at index i.
int	**getAndDecrement**(int i) Atomically decrements by one the element at index
void	**set**(int i, int newValue) Sets the element at position i to the given value.

Vi skal bare lære ett fåtall av dette

- Her er de vi skal konsentrere oss om:
  - new Thread – join()
  - synchronized method
  - Semaphore – acquire() og release()
  - CyclicBarrier – await()
  - ExecutorService pool = Executors.newFixedThreadPool(k);
    med Futures - forklares senere
  - AtomicIntegerArray – get(), set(), getAndAdd(),..
  - ReentrantLock (i pakken: **java.util.concurrent.locks**)
  - volatile variable - forklares senere

- Alle de synkroniseringer vi trenger, kan gjøres med disse!
- De fleste andre har sine måter å gjøre det på, men man har neppe tid til å lære seg alle.
- Bedre å bli flink i et lite og tilstrekkelig sett av synkroniseringsprimitiver, enn halvgod i de fleste.



Resume

- Learning-by-doing course:
  - Your learning is dependent on YOUR doing!
- Focus on writing programs that are FAST
- Not much THEORY – but LOTS of practice
- Get to know parallel programming – and the difficult task of synchronizing parallel activities
- Get to know CACHING – and WHY it is there and HOW it impacts performance
- Have FUN!



IN3030/IN4330, L02, vår, 2026
Cache

Michael Kirkedal Thomsen and Paulo Ferreira
Programming Technology Group
Programming Section
Department of Informatics
University of Oslo

1



Summary of first Lecture L01v25

- Motivation
  - Utilization of Multi-core – by changing sequential programs into parallel programs
  - Multi-core hardware driving this trend
- Purpose
  - Convert sequential program into parallel versions
  - Achieve speedup
- Requirements
  - Correctness – Effective!
  - Efficient – MUST be FASTER – measured by speedup
- Approach
  - Empirical – the proof of the pudding is in the eating!
  - Timings have the ULTIMATIVE SAY!
- (Theory is fine; but practise is essential!)

2

2



Summary of first Lecture L01v25 (second page)

- Central metric: SPEEDUP
  - Speedup: sequential time / parallel time
  - Want speedup > 1
  - Really want speedup = number of cores
- How?
  - Parallel threads in Java
  - Must synchronize
- Measuring programs - how?
  - Real-time clock times!!
- Threads in Java
- Template for parallel programs
- Synchronization
  - Cyclic Barrier in Java
  - Other synchronization methods in Java

3

3



Plan for L02v25

- What is caching – intro
- The effect of cache on execution time
- Presentation of Oblig1

4

4



Caching

- What is caching?
- The principle of caching
- Multi-level caching
- Prefetching

5

5



What is caching ?

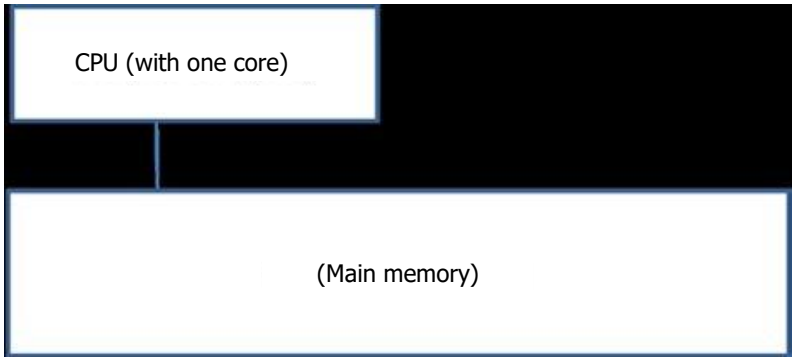
- Principle of a cache
  - CPUs fast – memory access slow
  - Move data into a fast cache
  - Make room: kick out Least Recently Used (LRU Strategy)
- Multiple level cache
- Speed of memory vs size
- Further performance enhancement: Prefetching

6

6

Machine 1980 (without a cache)

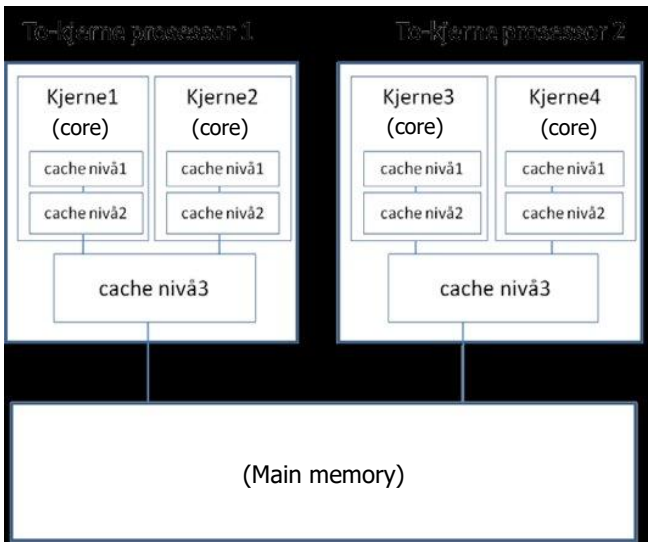
Sketch of a computer in about 1980 where there was only one computing unit, a CPU, which read its instructions and both wrote and read data (variables) directly into main memory. Intel 8080: 1 MHz CPU



7

7

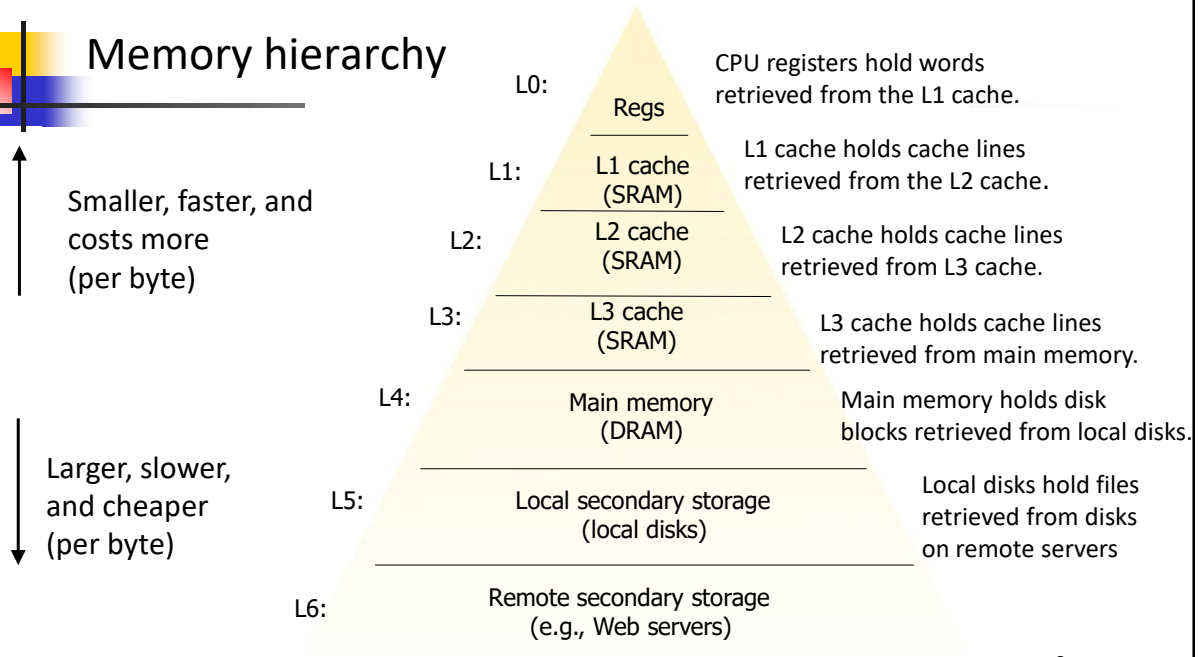
Machine in about 2010 with two dual-core CPUs



8

8

Memory hierarchy



9

9

Caching

- The L1, L2 and L3 caches are implemented with a hardware technology known as static random access memory (SRAM)
- The idea behind caching is that a system can get the effect of both a very large memory and a very fast one by exploiting locality, the tendency for programs to access data and code in localized regions.
- By setting up caches to hold data that are likely to be accessed often, we can perform most memory operations using the fast caches.
- The disk drive on a typical system might be 1 000 times larger than the main memory, but it might take the processor 10 000 000 times longer to read a word from disk than from memory.
- However, the processor can read data from the register file almost 100 times faster than from memory.
- It might take 5 times longer for the processor to access the L2 cache than the L1 cache, but this is still 5 to 10 times faster than accessing the main memory.

10

10



Memory Sizes and Times

Type	What cached	Where cached	Latency (cycles)	Managed by
CPU registers	4-byte or 8-byte words	On-chip CPU registers	0	Compiler
TLB	Address translations	On-chip TLB	0	Hardware MMU
L1 cache	64-byte blocks	On-chip L1 cache	4	Hardware
L2 cache	64-byte blocks	On-chip L2 cache	10	Hardware
L3 cache	64-byte blocks	On-chip L3 cache	50	Hardware
Virtual memory	4-KB pages	Main memory	200	Hardware + OS
Buffer cache	Parts of files	Main memory	200	OS
Disk cache	Disk sectors	Disk controller	100,000	Controller firmware
Network cache	Parts of files	Local disk	10,000,000	NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

Figure 6.23 **The ubiquity of caching in modern computer systems.** Acronyms: TLB: translation lookaside buffer; MMU: memory management unit; OS: operating system; NFS: network file system.

11

11



Unix

- The 1960s was an era of huge, complex operating systems, such as IBM's OS/360 and Honeywell's Multics systems.
- While OS/360 was one of the most successful software projects in history, Multics dragged on for years and never achieved wide-scale use.
- Bell Laboratories was an original partner in the Multics project but dropped out in 1969 because of concern over the complexity of the project and the lack of progress.
- In reaction to their unpleasant Multics experience, a group of Bell Labs researchers—Ken Thompson, Dennis Ritchie, Doug McIlroy, and Joe Ossanna—began work in 1969 on a simpler operating system for a Digital Equipment Corporation PDP-7 computer, written entirely in machine language.
- Many of the ideas in the new system, such as the hierarchical file system and the notion of a shell as a user-level process, were borrowed from Multics but implemented in a smaller, simpler package.

12

12



Unix

- In 1970, Brian Kernighan dubbed the new system “Unix” as a pun on the complexity of “Multics.”
- The kernel was rewritten in C in 1973, and Unix was announced to the outside world in 1974.
- Because Bell Labs made the source code available to schools with generous terms, Unix developed a large following at universities.
- The most influential work was done at the University of California at Berkeley in the late 1970s and early 1980s, with Berkeley researchers adding virtual memory and the Internet protocols in a series of releases called Unix 4.xBSD (Berkeley Software Distribution).
- Bell Labs was releasing their own versions, which became known as System V Unix.
- Versions from other vendors, such as the Sun Microsystems Solaris system, were derived from these original BSD and System V versions.

13

13



Unix

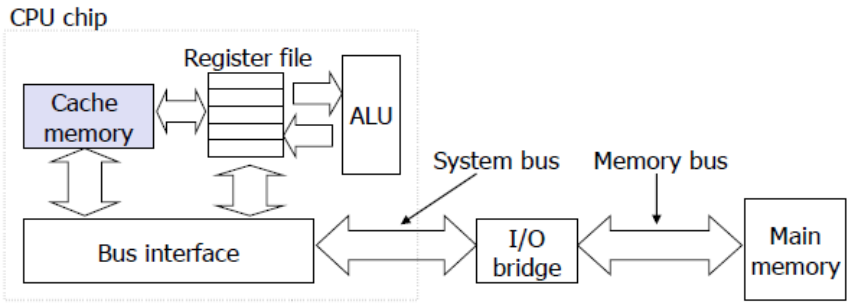
- Trouble arose in the mid 1980s as Unix vendors tried to differentiate themselves by adding new and often incompatible features.
- To combat this trend, IEEE (Institute for Electrical and Electronics Engineers) sponsored an effort to standardize Unix, later dubbed “Posix” by Richard Stallman.
- The result was a family of standards, known as the Posix standards, that cover such issues as the C language interface for Unix system calls, shell programs and utilities, threads, and network programming.
- More recently, a separate standardization effort, known as the “Standard Unix Specification,” has joined forces with Posix to create a single, unified standard for Unix systems.
- As a result of these standardization efforts, the differences between Unix versions have largely disappeared.

14

14

Cache Memories

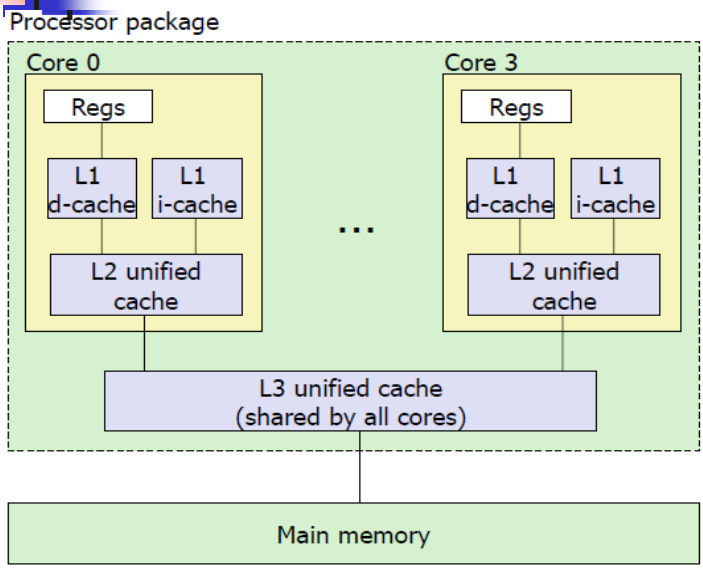
- Cache memories are small, fast SRAM-based memories managed automatically in hardware
  - Hold frequently accessed blocks of main memory
- CPU looks first for data in cache
- Typical system structure:



15

15

Intel Core i7 Cache Hierarchy



L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 10 cycles

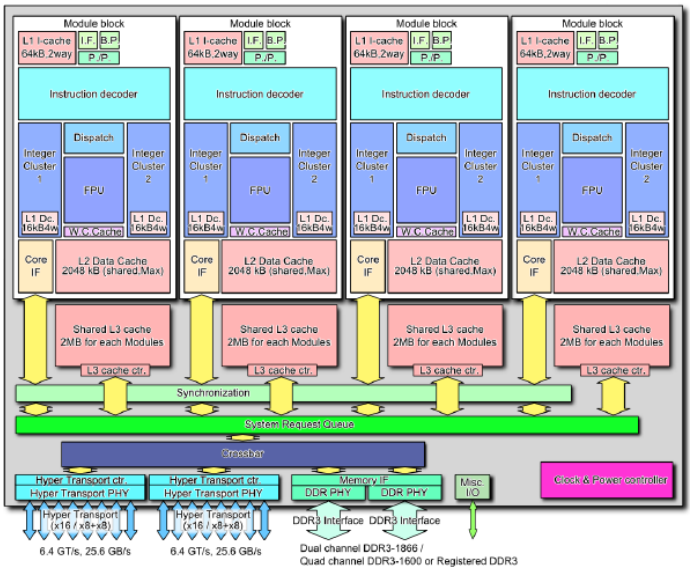
L3 unified cache:
8 MB, 16-way,
Access: 40-75 cycles

Block size: 64 bytes for all caches.

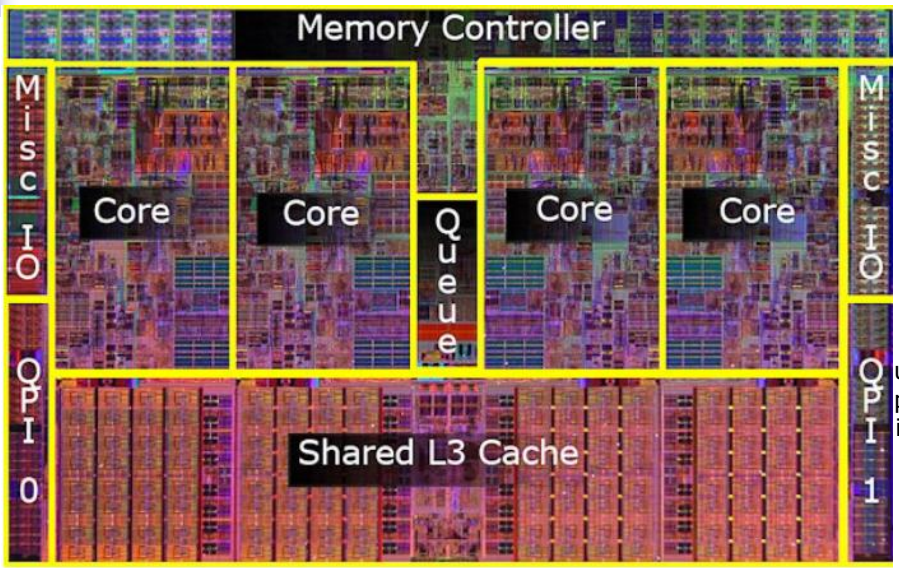
16

16

The memory system in a 4 core CPU



Intel Core i7 - 4-core CPU



uick path interconnect



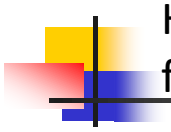
Memory

As a programmer, you need to understand the memory hierarchy because it has a big impact on the performance of your applications. If the data your program needs are stored in a CPU register, then they can be accessed in 0 cycles during the execution of the instruction. If stored in a cache, 4 to 75 cycles. If stored in main memory, hundreds of cycles. And if stored in disk, tens of millions of cycles!

For example, on our Core i7 system, the running times of different matrix multiplication kernels that perform the same number of arithmetic operations, but have different degrees of locality, can vary by a factor of almost 40!

19

19



How do we take the cache system into account to get faster programs?

- We only look at the data caches (little to fetch on the instructions)
- Important to know is that every time we fetch data in the main memory, we get a cache line = 64 bytes = e.g. 8 integers (int)
- There is very limited space in the caches, and a cache line that has not been used for a 'long time' will be 'thrown out' (overwritten by another, newer) cache line
- This is the fastest way:
  - Work on few data (short parts of an array) 'long' at a time – do not jump around
  - Preferably go forwards or backwards through the data (arrays)

  (i, i+1,.. or: i, i-1,..)
- We must create such cache-friendly programs!

20

20



Summary – the idea that we have uniform access time in memory is completely wrong

- Memory system in a multicore CPU, Intel Core i5-459 3.3 GHz, – many layers (typical access time in instruction cycles):
  - Registers in the core – 8/32 registers
  - L1 cache – 32 Kb
  - L2 cache – 256 kb
  - L3 cache – 8Mb
  - Main memory (virtual memory) – 8-64 GB
  - Disk (15 000 000 rotating) = 5 ms – 1000 GB

    1-5 TB FlashDisk (approx. 2 000 000 reads, approx. 10 000 000 writes)

    = approx. 1 ms

21

21



Absolutely crucial for us – cache memory

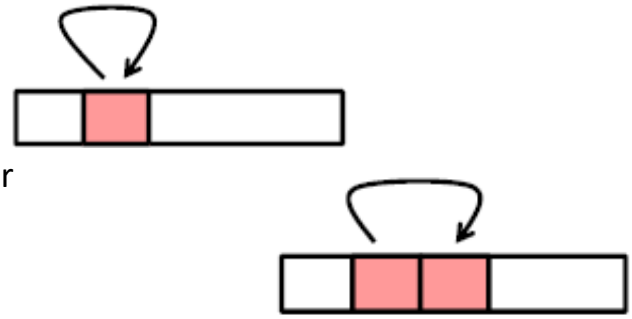
- What is cache:
  - Faster (but also more expensive) memory between the main memory and the cores.
  - We need cache because there are such large speed differences between a CPU core and the main memory (‘main memory’)
  - Often now 3-4 layers of cache memories + a number of registers in the core (even faster than the cache memories) that hold data or instructions
  - When a core needs data or a new instruction (and it doesn’t have it in a register) it looks down in the cache memories. First cache level 1 (L1), then the L2 cache, .. , before going to the main memory for data or instructions.
  - There are several techniques to do this quickly (such as pre-fetch, i.e. the system fetches the next data/instruction without the core having explicitly requested it)

22

22

Locality

- Principle of Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently
- Temporal locality: Recently referenced items are likely to be referenced again in the near future



- Spatial locality: Items with nearby addresses tend to be referenced close together in time

23

Writing Cache Friendly Code

- Make the common case go fast
  - Focus on the inner loops of the core functions
- Minimize the misses in the inner loops
  - Repeated references to variables are good (temporal locality)
  - Stride-1 reference patterns are good (spatial locality)
- Key idea:
  - Our qualitative notion of locality is quantified through our understanding of cache memories

24

The Memory Mountain

- Read throughput (read bandwidth)
  - Number of bytes read from memory per second (MB/s)
- Memory mountain: Measured read throughput as a function of spatial and temporal locality
  - Compact way to characterize memory system performance.

25

The Memory Mountain test function

```
long data[MAXELEMS];  /* Global array to traverse */

/* test - Iterate over first "elems" elements of
 *        array "data" with stride of "stride", using
 *        using 4x4 loop unrolling.
 */
int test(int elems, int stride) {
    long i, sx2=stride*2, sx3=stride*3, sx4=stride*4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems, limit = length - sx4;

    /* Combine 4 elements at a time */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i+stride];
        acc2 = acc2 + data[i+sx2];
        acc3 = acc3 + data[i+sx3];
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 + data[i];
    }
    return ((acc0 + acc1) + (acc2 + acc3));
}
```

mountain/mountain.c

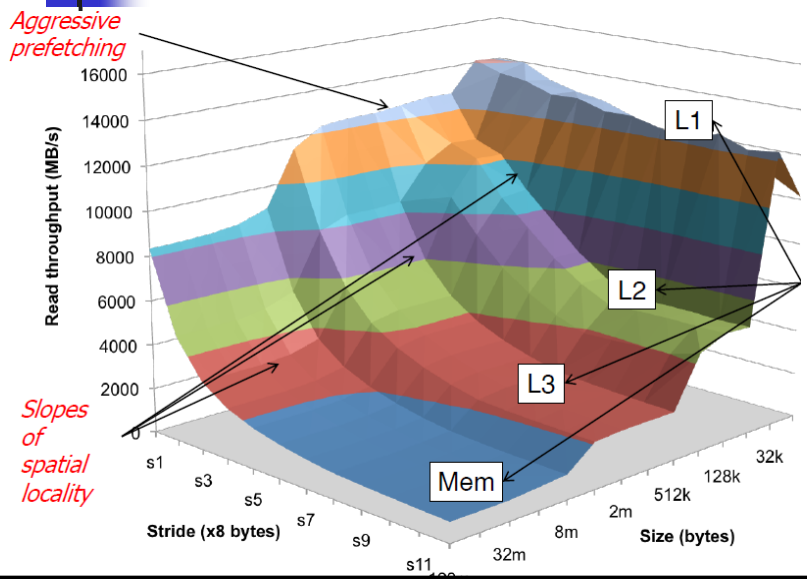
- Call test() with many combinations of elems and stride.
- For each elems and stride:

1. Call test() once to warm up the caches.
2. Call test() again and measure the read throughput(MB/s)

26

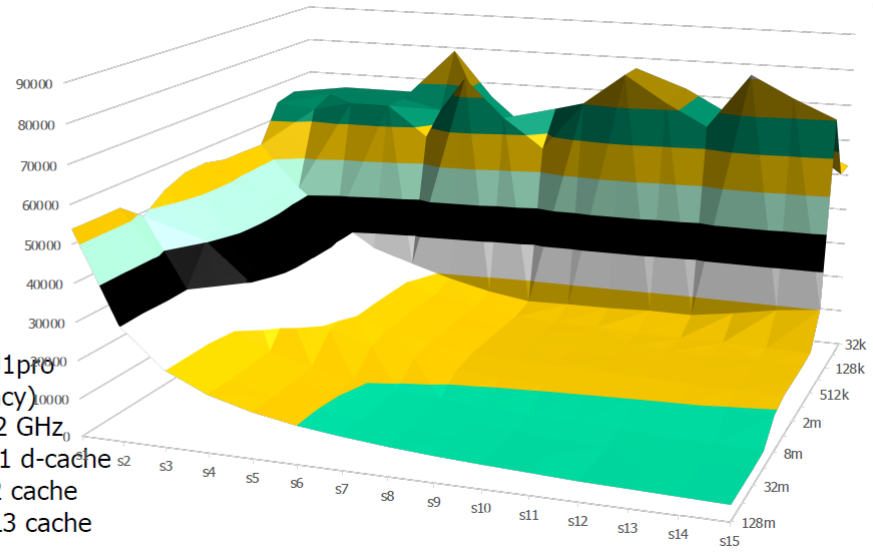
The Memory Mountain

- Core i7 Haswell
- 2.1 GHz
- 32 KB L1 d-cache
- 256 KB L2 cache
- 8 MB L3 cache
- 64 B block size



27

The memory mountain – Apple M1pro



Apple M1pro (efficiency)
2.4 - 3.2 GHz
64 KB L1 d-cache
4 MB L2 cache
32 MB L3 cache

28

Matrix Multiplication

- $O(n^3)$ total operations
- Identical number of adds and multiplies

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

- Each array is an $n \times n$ array of `double`, with `sizeof(double)` $= 8$.
- There is a single cache with a 32-byte block size ($B = 32$).
- The array size n is so large that a single matrix row does not fit in the L1 cache.
- The compiler stores local variables in registers, and thus references to local variables inside loops do not require any load or store instructions.

29

Matrix Multiplication (ijk + jik)

(a) Version *ijk*

code/mem/matmult/mm.c

```
1  for (i = 0; i < n; i++)
2      for (j = 0; j < n; j++) {
3          sum = 0.0;
4          for (k = 0; k < n; k++)
5              sum += A[i][k]*B[k][j];
6          C[i][j] += sum;
7      }
```

load

code/mem/matmult/mm.c

(b) Version *jik*

code/mem/matmult/mm.c

```
1  for (j = 0; j < n; j++)
2      for (i = 0; i < n; i++) {
3          sum = 0.0;
4          for (k = 0; k < n; k++)
5              sum += A[i][k]*B[k][j];
6          C[i][j] += sum;
7      }
```

code/mem/matmult/mm.c

The inner loops of the class *AB* routines (Figure 6.44(a) and (b)) scan a row of array *A* with a stride of 1. Since each cache block holds four 8-byte words, the miss rate for *A* is 0.25 misses per iteration. On the other hand, the inner loop scans a column of *B* with a stride of *n*. Since *n* is large, each access of array *B* results in a miss, for a total of 1.25 misses per iteration.

Stride 1 (k in bytes); words are 8 bytes; (B) cache block are 32 bytes; min (1, (word size x k) / B) 30

Matrix Multiplication (jki + kji)

(c) Version *jki*

code/mem/matmult/mm.c

```
1   for (j = 0; j < n; j++)
2       for (k = 0; k < n; k++) {
3           r = B[k][j];
4           for (i = 0; i < n; i++)
5               C[i][j] += A[i][k]*r;
6       }
```

load

code/mem/matmult/mm.c

(d) Version *kji*

code/mem/matmult/mm.c

```
1   for (k = 0; k < n; k++)
2       for (j = 0; j < n; j++) {
3           r = B[k][j];
4           for (i = 0; i < n; i++)
5               C[i][j] += A[i][k]*r;
6       }
```

code/mem/matmult/mm.c

The inner loops in the class *AC* routines (Figure 6.44(c) and (d)) have some problems. Each iteration performs two loads and a store (as opposed to the class *AB* routines, which perform two loads and no stores). Second, the inner loop scans the columns of *A* and *C* with a stride of *n*. The result is a miss on each load, for a total of two misses per iteration. Notice that interchanging the loops has decreased the amount of spatial locality compared to the class *AB* routines.

Stride 1 (k in bytes); words are 8 bytes; (B) cache block are 32 bytes; min (1, (word size x k) / B) 31

31

Matrix Multiplication (kij +ikj)

(e) Version *kij*

code/mem/matmult/mm.c

```
1   for (k = 0; k < n; k++)
2       for (i = 0; i < n; i++) {
3           r = A[i][k];
4           for (j = 0; j < n; j++)
5               C[i][j] += r*B[k][j];
6       }
```

load

code/mem/matmult/mm.c

(f) Version *ikj*

code/mem/matmult/mm.c

```
1   for (i = 0; i < n; i++)
2       for (k = 0; k < n; k++) {
3           r = A[i][k];
4           for (j = 0; j < n; j++)
5               C[i][j] += r*B[k][j];
6       }
```

code/mem/matmult/mm.c

The *BC* routines (Figure 6.44(e) and (f)) present an interesting trade-off: With two loads and a store, they require one more memory operation than the *AB* routines. On the other hand, since the inner loop scans both *B* and *C* row-wise with a stride-1 access pattern, the miss rate on each array is only 0.25 misses per iteration, for a total of 0.50 misses per iteration.

Stride 1 (k in bytes); words are 8 bytes; (B) cache block are 32 bytes; min (1, (word size x k) / B) 32

32



Summary of Matrix Multiplication

Matrix multiply version (class)	Per iteration					
	Loads	Stores	*A* misses	*B* misses	*C* misses	Total misses
ijk & *jik* (*AB*)	2	0	0.25	1.00	0.00	1.25
jki & *kji* (*AC*)	2	1	1.00	0.00	1.00	2.00
kij & *ikj* (*BC*)	2	1	0.00	0.25	0.25	0.50

this is the fastest

Stride 1 (k in bytes); words are 8 bytes; (B) cache block are 32 bytes; min (1, (word size x k) / B) 33



Summary of Matrix Multiplication

- Focus your attention on the inner loops, where the bulk of the computations and memory accesses occur.
- Try to maximize the spatial locality in your programs by reading data objects sequentially, with stride 1, in the order they are stored in memory.
- Try to maximize the temporal locality in your programs by using a data object as often as possible once it has been read from memory.

34

Summary of Matrix Multiplication

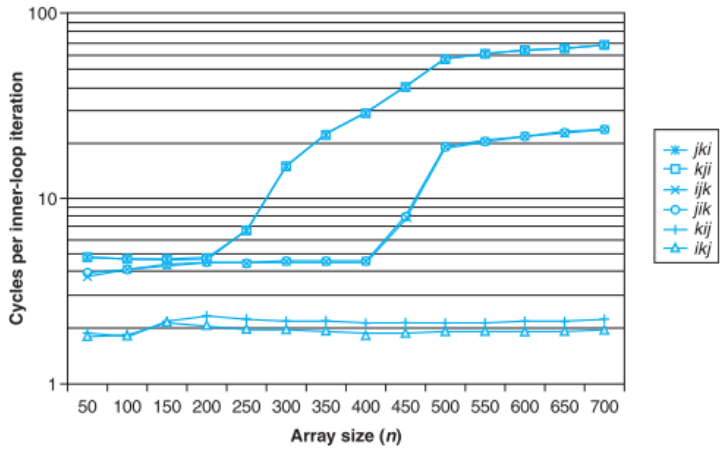


Figure 6.46 Core i7 matrix multiply performance.

35

Summary of Matrix Multiplication

- For large values of n, the fastest version runs almost 40 times faster than the slowest version, even though each performs the same number of floating-point arithmetic operations.
- Pairs of versions with the same number of memory references and misses per iteration have almost identical measured performance.
- The two versions with the worst memory behavior, in terms of the number of accesses and misses per iteration, run significantly slower than the other four versions, which have fewer misses or fewer accesses, or both.

36

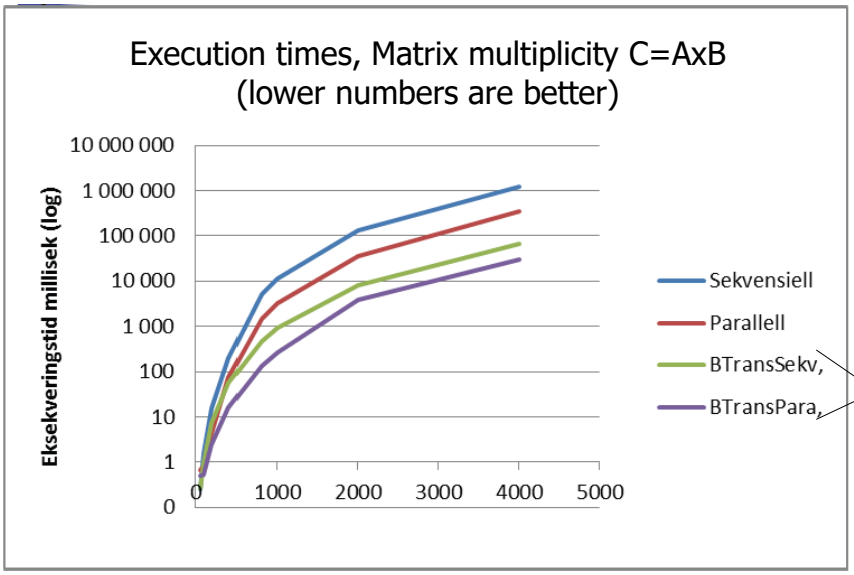
Summary of Matrix Multiplication

- Miss rate, in this case, is a better predictor of performance than the total number of memory accesses. For example, the class BC routines, with 0.5 misses per iteration, perform much better than the class AB routines, with 1.25 misses per iteration, even though the class BC routines perform more memory references in the inner loop (two loads and one store) than the class AB routines (two loads).
- For large values of n, the performance of the fastest pair of versions (kij and ikj) is constant. Even though the array is much larger than any of the SRAM cache memories, the prefetching hardware is smart enough to recognize the stride-1 access pattern, and fast enough to keep up with memory accesses in the tight inner loop. This is a stunning accomplishment by the Intel engineers who designed this memory system, providing even more incentive for programmers to develop programs with good spatial locality.

37

37

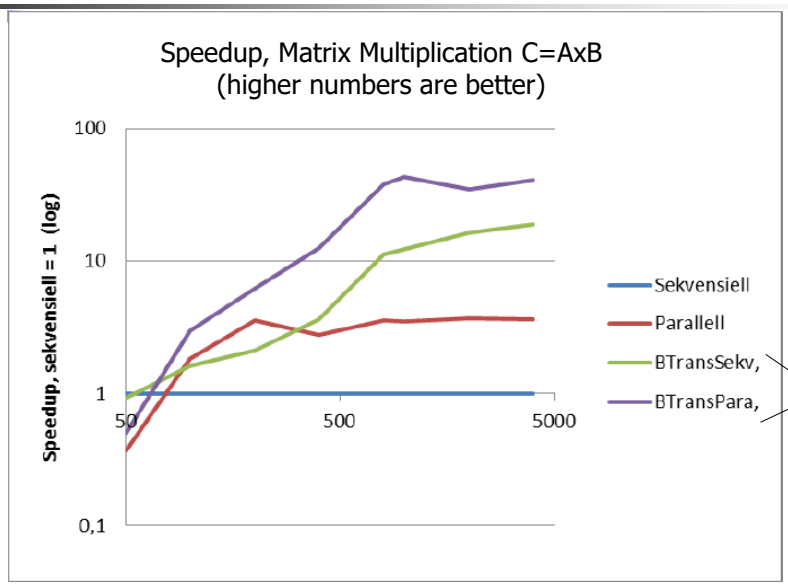
Runtimes – in ms (y-axis logarithmic)



38

38

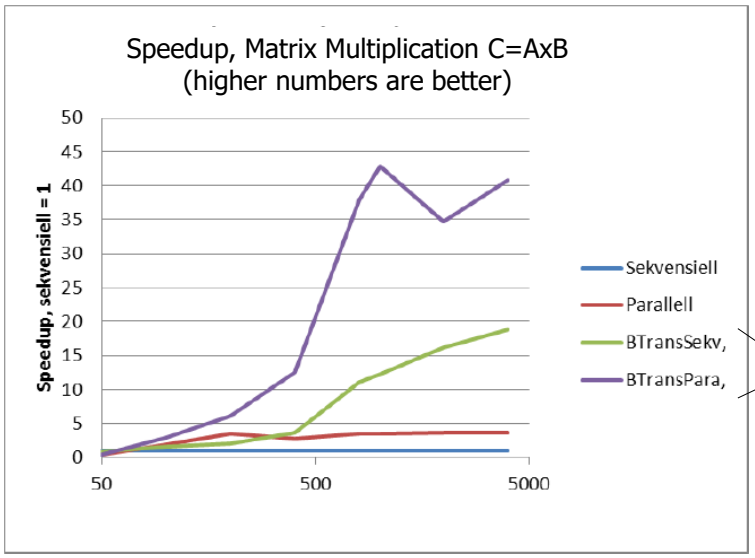
Runtime Results – Speedup, y-axis logarithmic



39

39

Speedup - with linear y-axis



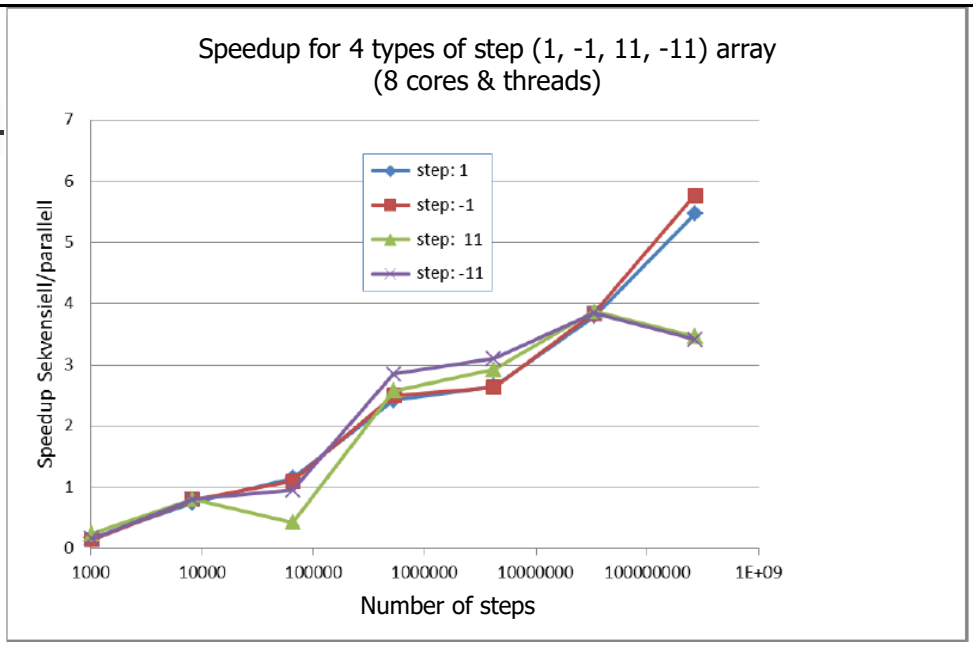
40

40

The on-chip prefetch mechanism

- Consists in the fact that if we access (read or write) element k in an array a[] and then element k+m, the electronics tries to fetch element a[k+2m], a[k+3m],…, before we have requested it
- The addition m can be both positive and negative
- If the elements a[k+2m] are in the same cache line, this is particularly fast
- First graphs

41



- Conclusion: Especially step 1,-1 has good speedup due to the same cache line

42



The prefetch mechanism helps a lot

- Not as important as the cache system
- Not as important as JIT-compilation but helps, and in no way gets in the way of the two most important mechanisms

43

43



The effect on execution times of cache

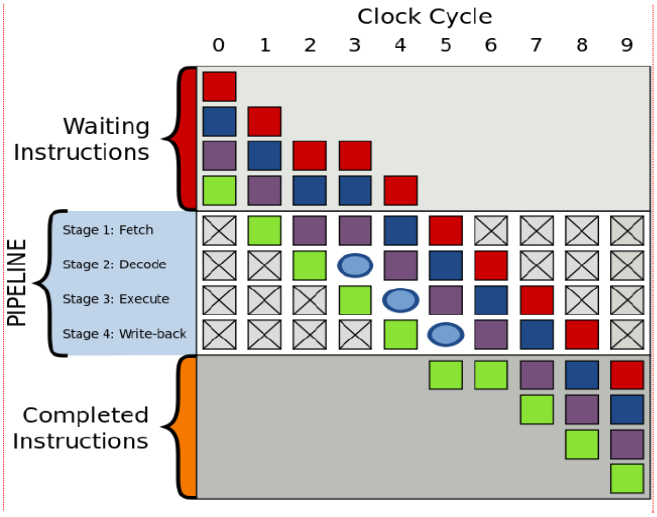
- How long does it take to execute n times (n=100, 1000, 10000,….,100 million): a[b[i]]=i;
- 2) Depends on what b[] contains:
  - 1) If b[i] = i (sequential) , then a[b[i]] = a[i] and we have ‘everything’ in the cache
  - 2) If the contents of b[] are randomly drawn between 0:n-1, then each read/write in the memory is a hop back and forth in a[] – no benefit of the cache
- 3) We can show how many times longer it takes to execute times the two ways of filling b[]– either b[i] = i, or b[i] = random(0..n-1)

44

44

Instruction parallelism in a CPU core

- Pipeline – multiple instructions are executed simultaneously in the fastest possible order.



Note, that this in an extremely simplified model of modern machines.

45

45

Conclusion – nested access a[b[i]]

- For ‘small’ values of n <1000, the cache provides good access to both the entire a[] (most importantly), and to b[]
- For large values of n > 1 000 000 it becomes much slower, and we can get between 12 – 240 times slower code (due to cache misses) when the content of b[] is ‘random’

46

46



Presentation of Oblig1

47



End L02v2

- End of lecture L02v25

48

IN3030/IN4330, L04, våren 2026
Moore's law & Speed of light, Latency
Oblig2

Eric Jul and Paulo Ferreira
Programming Technology Group
Programming Section
Department of Informatics
University of Oslo

1

Summary of second Lecture L02v26

- What is caching?
- The principle of caching
- Multi-level caching
- Prefetching

2



Summary of second Lecture L02v26

- What is caching?
- The principle of caching
- Multi-level caching
- Prefetching

3



Plan for L04v26

- Moore's Law
- Performance improvements in processor power
- Speed of light
- Effect of the speed of light
- How to present your timing results
- Presentation of Oblig2 Matrix Multiplication

4

Moore's «Law»

- Who knows this law?
- Used to be known as: «CPU speeds double every 2 years»

- Original Law: Transistors per square cm doubles
  - 1965 Article: Every year
  - 1975 Article: Every two years

5

Moore's Law

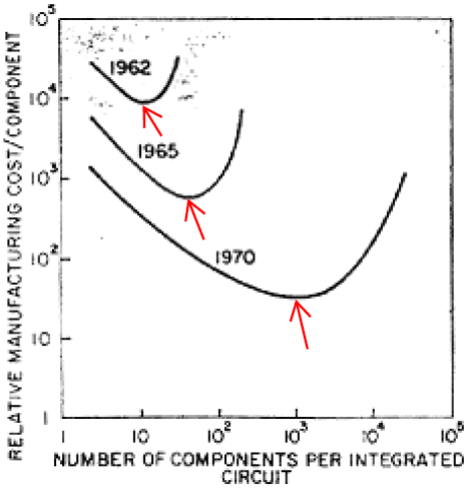
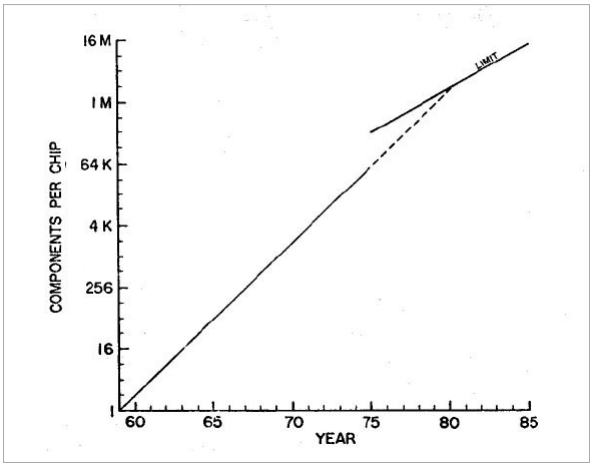
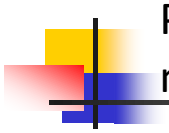


Figure 5 Projection of the complexity curve reflecting the limit on increased density through invention.



6

6



Perspective: Developments in CPU, memory, networking and more about Moore's law

- 1980 CPU: Intel 8080 8-bit processor
  - «int»: one byte
  - «long»: two bytes
  - CPU clock frequency: 1 MHz
  - Memory: 64 kilobytes max, i.e., 16 bit addresses (2 bytes)
  - 128 kB floppy disk
  - Data transmission 1.2 kbit/s == 150 bytes/s
- A look at Moore’s law
  - Moore’s original prediction 1965
  - Moore’s revisited prediction 1975

7



Newer Machines

- 2023: A Macbook Pro w M2 processor
  - 2.4 up-to 3.2 GHz – 12 x CPU cores, 19 GPU cores
  - 16 Gbyte Memory
  - 10 Gbit/s network
  - 1 TB Solid State Disk (SSD)
  - Liquid Retina XDR Display 14" 3024x1964 pixels

8

Moore's Law

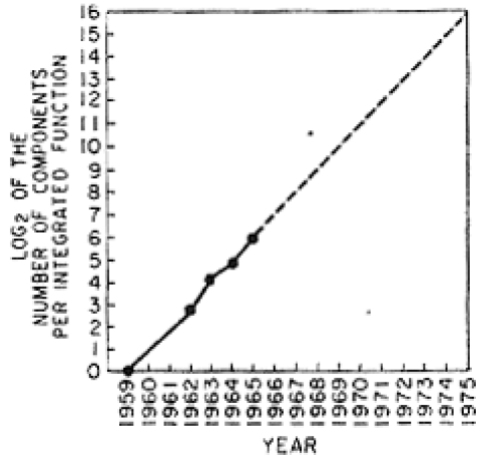
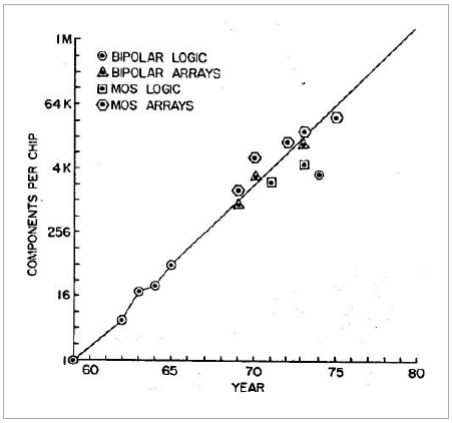


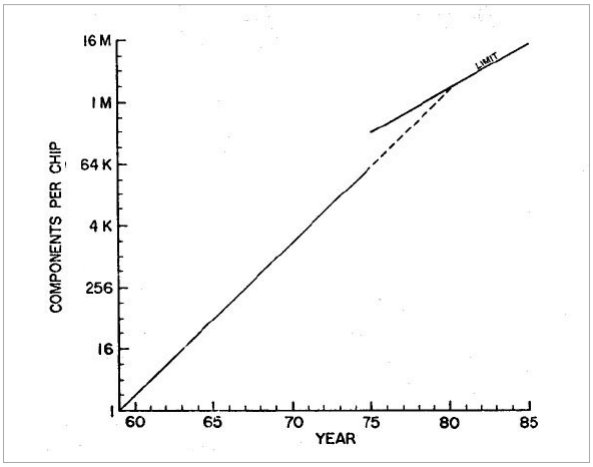
Figure 1 Approximate component count for complex integrated circuits vs. year of Introduction.



9

Moore's Law

Figure 5 Projection of the complexity curve reflecting the limit on increased density through invention.



10



Moore's Law Summary of Effects

- 1958-1975
  - Doubling of transistors every year
  - Derived effect: doubling of CPU speeds
- 1975-2005
  - Doubling of transistors every two years
  - Derived effect: doubling of CPU speeds – screeching halt at about 3 GHz
- 2005-2020
  - Doubling of transistors every two years
  - Derived effect: doubling of number of CORES or doubling of the amount of on-chip cache

11



Performance 1980 vs 2023

- 2023: Intel 8080 vs M2
  - 1 MHz CPU vs 8 x 3.2 GHz ~ factor 60,000
  - 16 Gbyte Memory vs. 64 kbytes ~ factor 250,000
  - 10 Gbit/s network vs 1 kbit/s ~ factor 10,000,000
  - 1 TB SSD vs 128 kB floppy disk ~ factor 8,000,000
  - 10 Gbit/s Ether vs 1200 baud serial line ~ factor 8,000,000
- ☺

12



What is ping?

- `Ping`: low-level internet messaging: sends an empty message from one computer to another.
- The other computer returns it
- Sending an empty message from Oslo to Seattle & back
- In 1988, when the internet was first "opened" in Scandinavia: a ping took a little under 200 ms
- How long does this take 38 years later – in 2026?

- *ping in the computer*

13



What about ping time?

- Ping: sending an empty message from Oslo to Seattle & back
- In 1988 it took 180-200 ms
- How long does this take 1988 vs 2022?
  - 1,000,000x faster?
  - 100,000x faster?
  - 10,000x faster?
  - 1,000x faster?
  - 100x faster?
  - 10x faster?
  - 10% faster?
  - Same time?
  - Slower?

14



Latency has **not** changed: dominated and limited by the incredibly slow speed of light!

- How fast is the speed of light approximately?
- Great circle route over and back 16,000 km
- Speed of light in fibre 11/15, copper 3/5
- Great Circle round trip time: minimum of about 100 ms
- So, ping time will **NEVER** go below 100 ms!

15



Speed of Light

- How fast is the speed of light approximately?
  - 300,000 km/s
- Speed of light
  - in fibre 11/15 (70%): about 220,000 km/s
  - in copper 3/5 (60%): about 180,000 km/s

16



Speed of Light

- **1 ns = 10^{-9} seconds**
  That's **one billionth of a second**.

- A few quick ways to picture it:
  - Light travels ~**30 cm** (about 1 foot) in 1 ns
  - A **3 GHz** CPU clock cycle ≈ **0.33 ns**, so 1 ns ≈ 3 cycles

  - In electronics, 1 ns is a *very* fast signal edge

17

17



Speed of Light

- Time scale
  - 1 ns = 0.000 000 001 s
  - There are 1,000,000,000 ns in a second

- Physics intuition
  - Speed of light: ~299,792,458 m/s → in 1 ns, light travels ~29.98 cm (about the length of a ruler)

  - Electric signals in copper travel ~15–20 cm per ns
    (slower than light because of the medium)

18

18



Speed of light revisited

- How fast is the speed of light approximately?
  - 300,000 km/s
- How far does light travel in 1 ns?
- How far in copper in 1 ns?
- When a 3 GHz core executes one cycle, how far does light travel?
  - 3 GHz CPU
    - 1 cycle ≈ 0.333 ns
    - So, 1 ns ≈ 3 CPU cycles

19

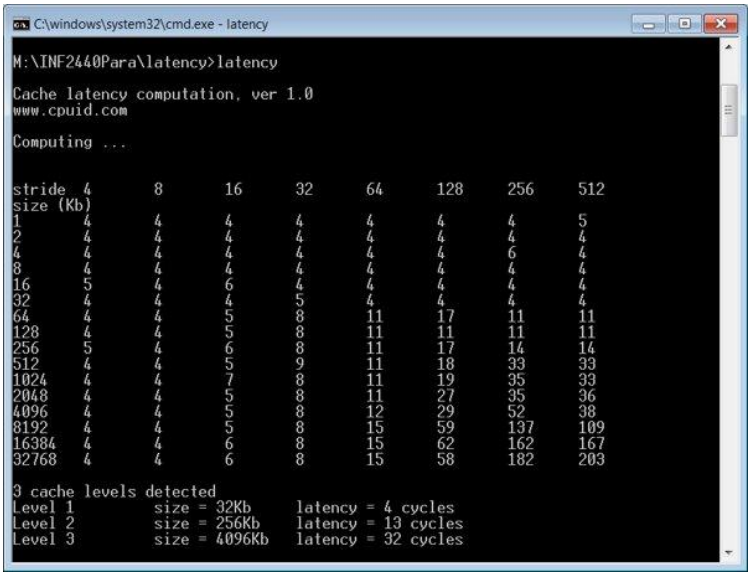


Speed of light: answers to questions

- How fast is the speed of light?
  - About 300,000 km/s
  - 299,792,458 m/s EXACTLY – by definition
- How far does light travel in 1 ns?
  - About 30 cm – call it a Lightfoot
- How far in copper in 1 ns?
  - About – 15 to 20 cm
- When a 3 GHz core executes one cycle, how far does light travel in vacuum and in copper?
  - About 10 cm (vacuum) and 6 cm (copper)

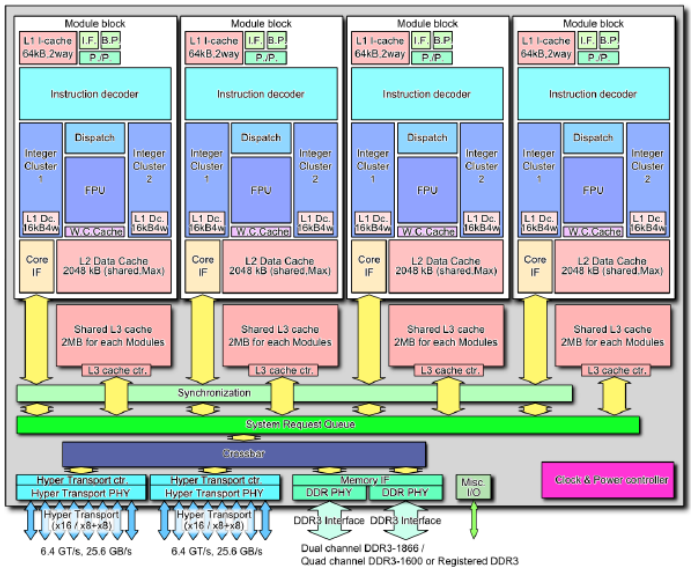
20

Latency

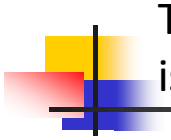


21

The memory system in a 4 core CPU



22



The idea that we have uniform access time in memory is completely wrong

- Memory system in a multicore CPU, Intel Core i5-459 3.3 GHz, – many layers (typical access time in instruction cycles):
  - Registers in the core – 8/32 registers
  - L1 cache – 32 Kb
  - L2 cache – 256 kb
  - L3 cache – 8Mb
  - Main memory (virtual memory) – 8-64 GB
  - Disk (15 000 000 rotating) = 5 ms – 1000 GB

    1-5 TB FlashDisk (approx. 2 000 000 reads, approx. 10 000 000 writes) = approx. 1 ms

23



The PRAM model for parallel computations

- PRAM (Parallel Random Access Memory) assumes two things:
  - You have infinitely many cores for the calculations
  - Every access to the memory takes the same amount of time, - ignoring the advantage of the cache memory, for example (a factor of 50)
- This makes many algorithms easy to calculate and program
- The problem is that both assumptions are completely wrong.
- What PRAM does is count the number of instructions executed
  - We have seen that this is completely wrong (Radix and Matrix-multiple)
- A great many courses and textbooks are based on PRAM
- PRAM means that the two sequential algorithms (with and without transposition) passed the un-transposed test faster!
- This course does not use the PRAM model!



Presentation of Oblig 2

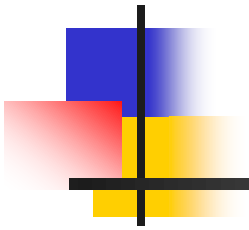
25



End of L04v25

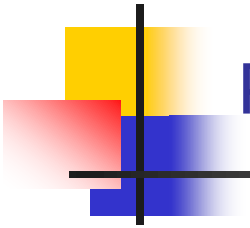
- End of lecture L04v25

26



IN3030 L05v26 – Prime Numbers

Eric Jul
Professor Emeritus
Programming Technology Group
Department of Informatics
University of Oslo

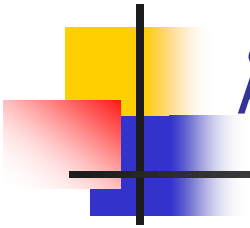


Plan for F05v26

I. Paralelisering hints
II. Amdahl's laws
III. Primtall
IV. Tidtagning
    I. JIT compilation
    II. Operativsystem?
    III. Søppel/Garbage Collection

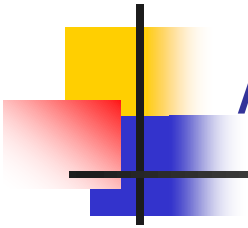
Om å parallelliser et problem

- Utgangspunkt: Vi har en sekvensiell effektiv og riktig sekvensiell algoritme som løser problemet.
- Vi kan dele opp både koden og data (hver for seg?)
- Vanligst å dele opp data
  - Som oftest deler vi opp data, og lar 'hele' koden virke på hver av disse data-delene (en del til hver tråd).
  - Eks: Matriser
    - radvis eller kolonnevis oppdeling av C til hver tråd
    - Omforme data slik at de passer bedre i cachene (transponere B)
  - Rekursiv oppdeling av data ('lett')
    - Eks: Quicksort
- Også mulig å dele opp koden:
  - Alternativ Oblig3 i INF1000: Beregning av Pi (3,1415..) med 17 000 sifre med tre ArcTan-rekker
  - Primtalls-faktorisering av store tall N for kodebrekking:
    - $N = p_1 * p_2$



Å dele opp algoritmen

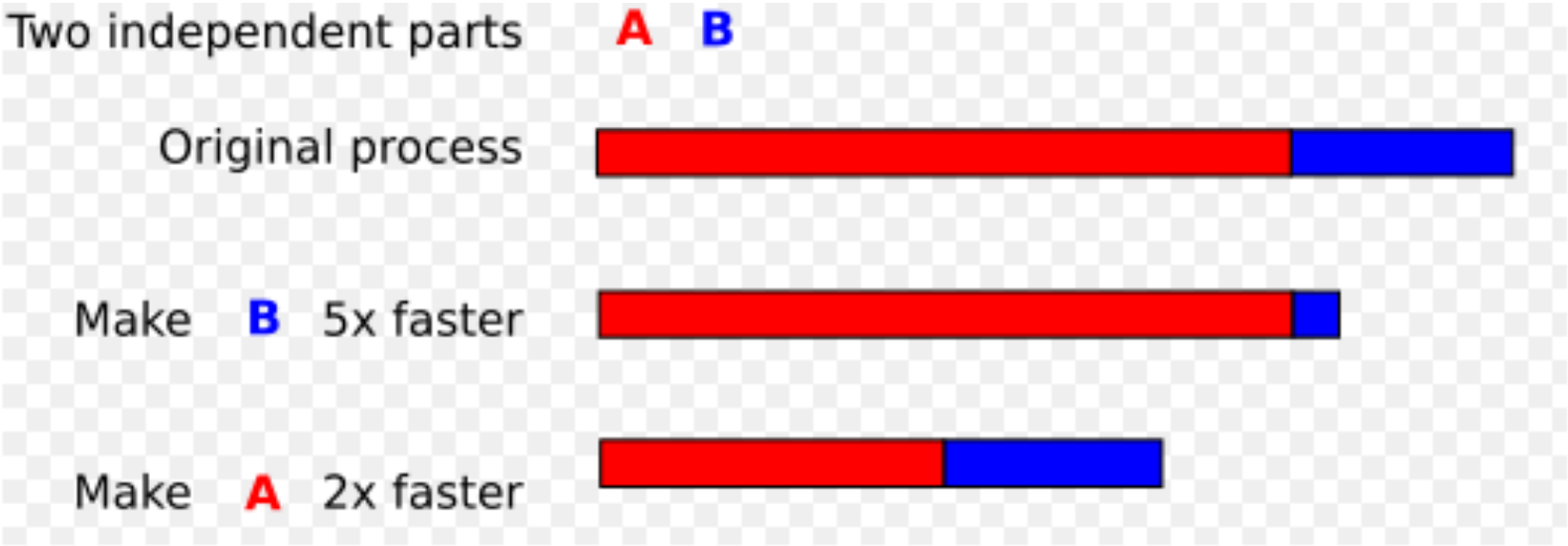
- Koden består en eller flere steg; som oftest i form av en eller flere samlinger av løkker (som er enkle, doble, triple..)
- Vi vil parallellisere med k tråder, og hver slikt steg vil få hver sin parallellisering med en CyclickBarrier-synkronisering mellom hver av disse delene + en synkronisert avslutning (join(), ..).
- Eks:
  - finnMax – hadde ett slikt steg: for (int i = 0 …n-1) -løkke
  - MatriseMult hadde ett slikt steg med trippel-løkke
  - Flere steg mulig: Eksempler senere i kurs (Radix)



Å dele opp data – del 2

- For å planlegge parallellisering av ett slikt steg må vi finne:
  - Hvilke data i problemet er lokale i hver tråd?
  - Hvilke data i problemet er felles/delt mellom trådene?
- Viktig for effektiv parallell kode.
  - Hvordan deler vi opp felles data (om mulig)
  - Kan hver tråd beregne hver sin egen, disjunkte del av data
  - Færrest mulig synkroniseringer (de tar 'mye' tid)

Amdahl – Important to parallelize big part



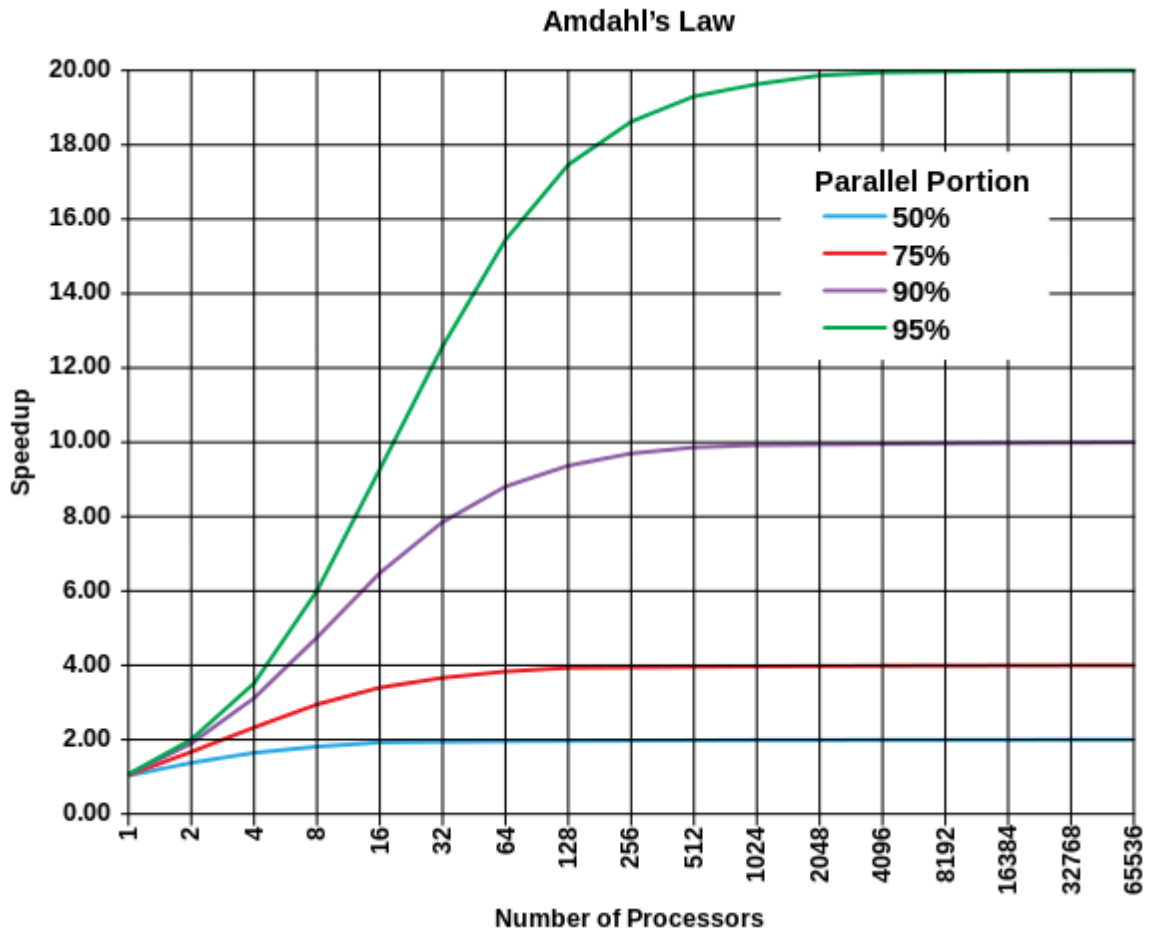
Amdahl lov for parallelle beregninger

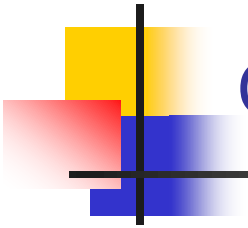
- «Hvis du først har brukt 10% av tida på en sekvensiell del, så kan resten av programmet ikke gå fortere enn 0.00 sekunder uansett hvor mange prosessorer du bruker på det. Dvs. at speedup ≤ 10»
- Amdahl lov: Har du **seq** andel sekvensiell kode og da **p** andel parallelliserbar kode i et parallelt program, **seq+p=1**, er den største speedup S du kan få med k kjerner:

$$S = \frac{tid(sekvensiell)}{tid\ (parallell)} = \frac{1}{seq+p/k} = \frac{1}{1-p+p/k}$$

- Når k $\rightarrow\ \infty$, vil S $\rightarrow \frac{1}{1-p}$.
- Er p=0.9, så er S ≤ 10 uansett hvor mange kjerner du har, og har du 'bare' 50, er S = $\frac{1}{1-0.9+0,9/50}$ = 8,5.
- Amdahls lov er pessimistisk- antar fast størrelse på problemet

Amdahl for ulike verdier av p





Om primtall

- Primtall og faktorisering av ikke-primtall.
- Et primtall er:
  Et heltall som bare lar seg dividere med 1 og seg selv.
  - 1 er ikke et primtall (det mente mange på 1700-tallet, og noen mener det fortsatt)

Om primtall og faktorisering av heltall

- Ethvert heltall N > 1 lar seg faktorisere som et produkt av primtall:
  - $N = p_1 * p_2 * p_3 * \ldots\ldots\ldots * p_k$
  - Denne faktoringen er entydig (pånær rækkefølge)
  - gjøres entydig hvis tall i faktoriseringen sorteres
  - Hvis det bare er ett tall i denne faktoriseringen, er N selv et primtall
- Eksempler:
  - 2 = 2
  - 3 = 3
  - 4 = 2*2
  - 5 = 5
  - 6 = 2 * 3 = 3 * 2
  - 7 = 7
  - 8 = 2 * 2 * 2

2 måter å lage primtall

Ønsker at finne alle primtal $p_i < N$

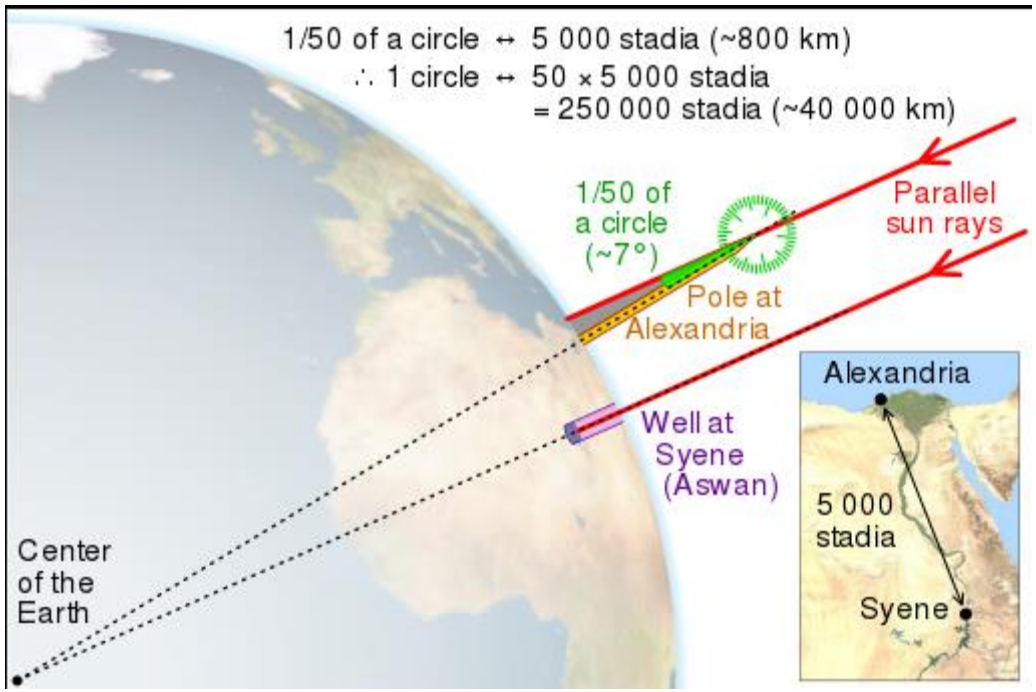
- Dividere alle tall < N med alle tall < N
  - Divisjonsmetoden
  - Bare oddetall (2 spesiel)
  - Bare opp til $\sqrt{N}$ -- hvorfor?
  - Bare primtall opp til $\sqrt{N}$ -- hvorfor?

- Lage en tabell over alle de primtallene vi trenger
  - Eratosthene sil

Litt mer om Eratosthenes

Eratosthenes, matematikker, laget også et estimat på jordas radius som var < 1,5% feil, grunnla geografi som fag, fant opp skuddårsdagen + at han var sjef for Biblioteket i Alexandria (den tids største forskningsinstitusjon).





Hvad er raskest?

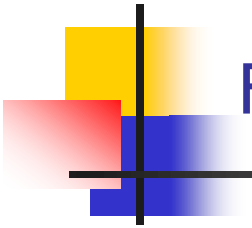
- A) Med Eratosthenes sil:

```
Z:\INF2440Para\Primtall>java PrimtallESil 2000000000
 max primtall m:2000000000
Genererte alle primtall <= 2000000000  paa     18 949 millisek
med Eratosthenes sil  og det største primtallet er:1999999973
```

- Med gjentatte divisjoner

```
Z:\INF2440Para\Primtall>java PrimtallDiv 2000000000
Genererte alle primtall <=2000000000  paa   1 577 302  millisek med
 divisjon , og det største primtallet er:1999999973
```

- Å lage primtallene p og finne dem ved divisjon (del på alle oddetall < SQRT(p), p = 3,5,7,..) er ca. 100 ganger langsommere enn Eratosthenes avkryssings-tabell (kalt Eratosthenes sil).



Finne primtall -- Eratosthenes sil

- Hvordan?
- (Blackboard)

Om primtall og faktorisering av heltall

- Ethvert heltall N > 1 lar seg faktorisere som et produkt av primtall:
  - $N = p_1 * p_2 * p_3 * \ldots\ldots\ldots * p_k$
  - Denne faktoringen er entydig (pånær rækkefølge)
  - gjøres entydig hvis tall i faktoriseringen sorteres
  - Hvis det bare er ett tall i denne faktoriseringen, er N selv et primtall
- Eksempel: faktorisering av 532
  - 532 = 2 * 266
  - 532 = 266*2 = 2 * 2 * 133
  - 532 = 266*2 = 2 * 2 * 7 * 19

Å lage og lagre primtall (Erotosthenes sil)

- Som en bit-tabell (1- betyr primtall, 0-betyr ikke-primtall)
  - Påfunnet i jernalderen av Eratosthenes (ca. 200 f.kr)
  - Man skal finne alle primtall < M
  - Man finner da de første primtallene og krysser av alle multipla av disse (N.B. dette forbedres/endres senere):
    - Eks: 3 er et primtall, da krysses 6, 9,12,15,.. Av fordi de alle er ett-eller-annet-tall (1,2,3,4,5,..) ganger 3 og følgelig selv ikke er et primtall. 6=2*3, 9 = 3*3,
      12 =2*2*3, 15 = 5*3, ..osv
    - De tallene som *ikke blir* krysset av, når vi har krysset av for alle primtallene vi har, er primtallene
- Vi finner 5 som et primtall fordi, etter at vi har krysset av for 3, finner første ikke-avkryssete tall: 5, som da er et primtall (og som vi så krysser av for, …finner så 7 osv)

Litt mer om Eratothenes sil

- Vi representerer ikke partallene på den tallinja som det krysses av på fordi vi vet at 2 er et primtall (det første) og at alle andre partall er ikke-primtall.
- Har vi funnet et nytt primtall p, for eksempel. 5, starter vi avkryssingen for dette primtallet først for tallet p*p (i eksempelet: 25), men etter det krysses det av for p*p+2p, p*p+4p,.. (i eksempelet 35,45,55,…osv.). Grunnen til at vi kan starte på p*p er at alle andre tall t < p*p slik det krysses av i for eksempel Wikipedia-artikkelen har allerede blitt krysset av andre primtall < p.
- Det betyr at for å krysse av og finne alle primtall < N , behøver vi bare å krysse av på denne måten for alle primtall p ≤ sqrt(N). Dette sparer svært mye tid.

Vise at vi trenger bare primtallene <10 for å finne alle primtall < 100, avkryssing for 3 (3*3, 9+2*3,9+4*3, ….)

1	**3**	**5**	**7**	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75	77	79
81	83	85	87	89
91	93	95	97	99

1	**3**	**5**	**7**	**9**
11	13	**15**	17	19
21	23	25	**27**	29
31	**33**	35	37	**39**
41	43	**45**	47	49
51	53	55	**57**	59
61	**63**	65	67	**69**
71	73	**75**	77	79
81	83	85	**87**	89
91	**93**	95	97	**99**

Avkryssing for 5 (starter med 25, så 25+2*5, 25+4,5,..):

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75	77	79
81	83	85	87	89
91	93	95	97	99

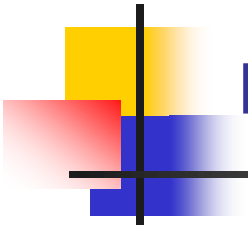
1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45 45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75 75	77	79
81	83	85	87	89
91	93	95	97	99

Avkryssing for 7 (starter med 49, så 49+2*7,49+4*7,.):

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45 45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75 75	77	79
81	83	85	87	89
91	93	95	97	99

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45 45	47	49
51	53	55	57	59
61	63 63	65	67	69
71	73	75 75	77	79
81	83	85	87	89
91	93	95	97	99

Er nå ferdig fordi neste primtall vi finner: 11, så er 11*11=121 utenfor tabellen

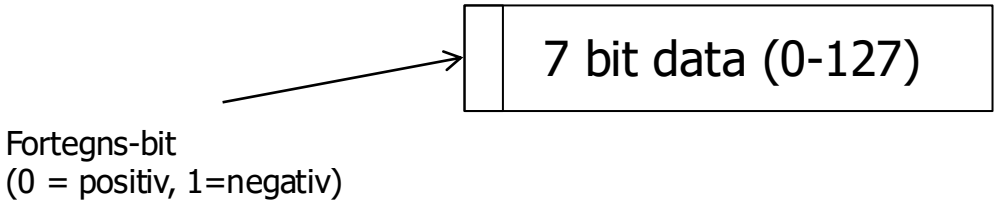


Hvordan representeres tallene?

- Kun oddetall – 2 kjenner vi!
- Array of Boolean?
  - Problem: 32 bit per primtall
- Kompakter bitarray
    - Kun 1 bit per oddetall

Hvordan bruke 8 eller 7 bit i en **byte-array** for å representere primtallene

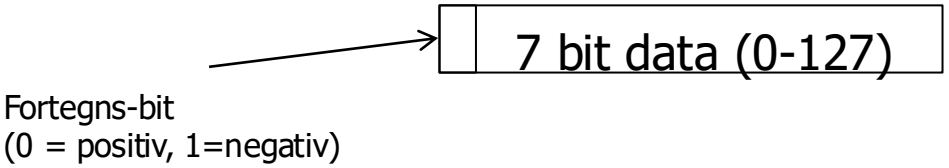
En byte = 8 bit heltall:



- Vi representer alle oddetallene (1,3,5,,,) som ett bit (0= ikke-primtall, 1 = primtall)
- Bruke alle 8 bit :
  - Fordel: mer kompakt lagring og litt raskere(?) adressering
  - Ulempe: Kan da ikke bruke verdien i byten direkte (f.eks som en indeks til en array), heller ikke +,-,* eller /-operasjonene på verdien
- Bruke 7 bit:
  - Fordel: ingen av ulempene med 8 bit
  - Ulempe: Tar litt større plass og litt langsommere(?) adressering

Hvordan representere 8 (eller 7) bit i en byte-array

byte = et 8 bit heltall



- Bruker alle 8 bitene til oddetallene:
  - Anta at vi vil sjekke om tallet ***k*** er et primtall, sjekk først om ***k*** er 2, da ja, hvis det er et partall (men ikke 2) da nei – ellers sjekk så tallets bit i byte-arrayen
    - Byte nummeret til ***k*** i arrayen er da:
      - Enten: ***k***/16, eller: ***k*** >>>4 (shift 4 høyreover uten kopi av fortegns-bitet er det samme som å dele med 16)
    - Bit-nummeret er i denne byten er da enten (***k*** %16)/2 eller (***k*** &15)>>1
  - Hvorfor dele på 16 når det er 8 bit
    - fordi vi fjernet alle partallene – egentlig 16 tall representert i første byten, for byte 0: tallene 0-15
  - Om så å finne bitverdien – se neste lysark.

Bruke 7 bit i hver byte i arrayen

- Anta at vi vil sjekke om tallet ***k*** er et primtall sjekk først om ***k*** er 2, da ja, ellers hvis det er et partall (men ikke 2) da nei – ellers:
- Sjekk da tallets bit i byte-arrayen
  - Byte nummeret til ***k*** i arrayen er da: ***k***/14
  - Bit-nummeret er i denne byten er da: (***k*** %14)/2
- Nå har vi byte-nummeret og bit-nummeret i den byten. Vi kan da ta AND (&) med det riktige elementet i en av de to arrayene som er oppgitt i skjelett-koden og teste om svaret er 0 eller ikke.
- Hvordan sette alle 7 eller 8 bit == 1 i alle byter)
  - 7 bit: hver byte settes = 127 (men bitet for 1 settes =0)
  - 8 bit: hver byte settes = -1 (men bit for 1 settes = 0)

- Konklusjon: bruk 8 eller 7 bit i hver byte (valgfritt)

Faktorisering av et tall M i sine primtallsfaktorer

- Vi har laget og lagret ved hjelp av Erotosthanes sil alle (unntatt 2) primtall < N i en bit-array over alle odde-tallene.
  - 1 = primtall, 0=ikke-primtall
  - Vi har krysset ut de som ikke er primtall
- Hvordan skal vi så bruke dette til å faktorisere et tall M < N*N ?
- **Svar:** Divider M med alle primtall $p_i<\sqrt{M}$ (p_i =2,3,5,..), og hver gang en slik divisjon M%p_i ==0, så er p_i en av faktorene til M. Vi forsetter så med å faktorisere ett mindre tall M'=M/p_i.
- Faktoriseringen av M = p_i *..* p_k er da produktet av alle de primtall som dividerer M uten rest.
- HUSK at en p_i kan forekommer flere ganger i svaret. eks: 20= 2*2*5, 81 = 3*3*3*3, osv
- Finner vi ingen faktorisering av M, dvs. ingen $p_i \leq \sqrt{M}$ som dividerer M med rest == 0, så er M selv et primtall.

Hvordan parallellisere faktorisering ?

1. Gjennomgås neste uke - denne uka viktig å få på plass en effektiv sekvensiell løsning med om lag disse kjøretidene for N = 2 mill:

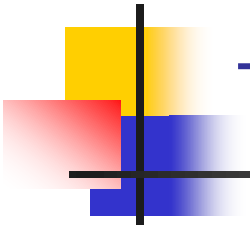
```
M:\INF2440Para\Primtall>java PrimtallESil 2000000
 max primtall m:2 000 000
Genererte primtall <= 2000000  paa        15.56 millisek
med Eratosthenes sil (  0.00004182 millisek/primtall)
.....
3999998764380 = 2*2*3*5*103*647248991
3999998764381 = 37*108108074713
3999998764382 = 2*271*457*1931*8363
3999998764383 = 3*19*47*1493093977
3999998764384 = 2*2*2*2*2*7*313*1033*55229
3999998764385 = 5*13*59951*1026479
3999998764386 = 2*3*3*31*71*100964177
3999998764387 = 1163*1879*1830431
3999998764388 = 2*2*11*11*17*23*293*72139
 100 faktoriseringer beregnet paa:     422.0307ms -
 dvs:      4.2203ms. per faktorisering
```

Faktorisering av store tall med 18-19 desimale sifre

```
Uke5>java PrimtallESil 2140000000
max primtall m:2 140 000 000
bitArr.length:133 750 001
Genererte primtall <= 2 140 000 000  paa    11030.36 millisek
med Eratosthenes sil ( 0.00010530 millisec/primtall)
 antall primtall < 2 140 000 000 er: 104 748 779, dvs:  4.89% ,
 og det største primtallet er: 2 139 999 977

4 579 599 999 999 999 900 = 2*2*3*5*5*967*3673*19421*221303
4 579 599 999 999 999 901 = 45795999999999901
4 579 599 999 999 999 902 = 2*22897999999999951
4 579 599 999 999 999 903 = 3*31*13188589*3733758839
4 579 599 999 999 999 904 = 2*2*2*2*2*19*71*106087842846553
4 579 599 999 999 999 905 = 5*7*1308457142857142 83
.............
4 579 599 999 999 999 997 = 11*4163272727272727
4 579 599 999 999 999 998 = 2*121081*18911307306679
4 579 599 999 999 999 999 = 3*17*19*6625387*713333333
100 faktoriseringer beregnet paa:  333481.4427ms
 dvs:   3334.8144ms. per faktorisering

 largestLongFactorizedSafe: 4 579 599 841 640 001 173= 2139999949*2139999977
```



Tidtagning

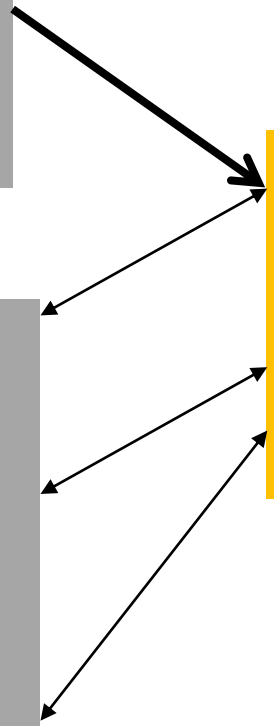
- JIT –kompilering
  - Hvor mye betyr det egentlig
- Operativsystemet (Windows eller Linux)
  - Er de like raske?
- Søppeltømming i Java
  - Skjer under kjøring (med i tidene)

Tidsmålinger og JIT (Just In Time) -kompilering

- Tilbake til kompileringen av et Java-program:

javac kompilerer først vårt java-program til en .class fil. som består av **byte-kode**

java (JVM) starter vår program i 'main()', men følger med.
1.Kalles en metode flere ganger, kompileres den over fra bytekode til **maskinkode**.
2. Kalles den enda mange ganger kan denne koden igjen **optimaliseres** (flere ganger)



main( ).
Vårt program kjører først interpretert (byte-koden tolkes).
Blir JIT-kompilert (mens koden kjører) en eller flere ganger. Går mye raskere

Optimaliserng – ett eksempel

Original kode

```
class A {
  B b;
  public void newMethod() {
    y = b.get();
    ...do stuff...
    z = b.get();
    sum = y + z;
  }
}
class B {
   int value;
   final int get() {
      return value;
   }
}
```

1) Inline get

```
public void
newMethod() {
    y = b.value;
    ...do stuff...
    z = b.value;
    sum = y + z;
}
```

2) Fjern overflødige les

```
public  void
newMethod() {
    y = b.value;
    ...do stuff...
    z = y;
    sum = y + z;
}
```

3) Fjern overflødige variable

```
public  void
newMethod() {
    y = b.value;
    ...do stuff...
    y = y;
    sum = y + y;
}
```

4) Fjern død kode

```
public  void
newMethod() {
    y = b.value;
    ...do stuff...
    sum = y + y;
}
```

Mediantider for finnMax fra ukeoppgavene:

n= 10 000

Vi ser at kjøretidene (sekv og para) synker dramatisk fra 1.ste til neste kjøring.
Pga JIT-optimalisering

```
M:\INF2440Para\FinnMax>java FinnMaxMulti 10000 7

Kjøring:0, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     6.30 msek. , nanosek/n:  630.46
Max sekv = a:9853, paa:     0.28 msek. , nanosek/n:   28.38

Kjøring:1, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.57 msek. , nanosek/n:   56.87
Max sekv = a:9853, paa:     0.27 msek. , nanosek/n:   26.95

Kjøring:2, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.35 msek. , nanosek/n:   35.07
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.36

Kjøring:3, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.57 msek. , nanosek/n:   56.87
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    0.66

Kjøring:4, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.43 msek. , nanosek/n:   43.47
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.33

Kjøring:5, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.49 msek. , nanosek/n:   49.20
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.36

Kjøring:6, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.48 msek. , nanosek/n:   47.84
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.43

Median seq time:   0.014, median para time:   0.569,
 Speedup:    0.03, n = 10000
```

```
M:\INF2440Para\FinnMax>java FinnMaxMulti 10000000 5

Kjøring:0, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:    14.08 msek. , nanosek/n:    1.41
Max sekv = a:9999216, paa:     6.98 msek. , nanosek/n:    0.70

Kjøring:1, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:     3.17 msek. , nanosek/n:    0.32
Max sekv = a:9999216, paa:     4.75 msek. , nanosek/n:    0.47

Kjøring:2, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:     2.79 msek. , nanosek/n:    0.28
Max sekv = a:9999216, paa:     5.04 msek. , nanosek/n:    0.50

Kjøring:3, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:     2.87 msek. , nanosek/n:    0.29
Max sekv = a:9999216, paa:     5.05 msek. , nanosek/n:    0.51

Kjøring:4, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:     2.92 msek. , nanosek/n:    0.29
Max sekv = a:9999216, paa:     5.03 msek. , nanosek/n:    0.50

Median seq time:   5.052, median para time:   3.173,
 Speedup:    1.59, n = 10 000 000
```

n= 10 mill

```
M:\INF2440Para\FinnMax>java -Xint FinnMaxMulti 10000000 5

Kjøring:0, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:    67.24 msek. , nanosek/n:    6.72
Max sekv = a:9999216, paa:   179.40 msek. , nanosek/n:   17.94

Kjøring:1, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:    64.00 msek. , nanosek/n:    6.40
Max sekv = a:9999216, paa:   175.12 msek. , nanosek/n:   17.51

Kjøring:2, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:    51.42 msek. , nanosek/n:    5.14
Max sekv = a:9999216, paa:   176.23 msek. , nanosek/n:   17.62

Kjøring:3, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:    64.95 msek. , nanosek/n:    6.49
Max sekv = a:9999216, paa:   173.17 msek. , nanosek/n:   17.32

Kjøring:4, ant kjerner:8, antTråder:8
Max para = a:9999216, paa:    60.11 msek. , nanosek/n:    6.01
Max sekv = a:9999216, paa:   185.84 msek. , nanosek/n:   18.58

Median seq time: 179.403, median para time:  64.950,
 Speedup:    2.76, n = 10 000 000
```

JIT-kompilering avslått :
> java –Xint
….
n= 10 mill

```
M:\INF2440Para\FinnMax>java FinnM 100000000 5
Kjoering:0, ant kjerner:8, antTraader:8
Max verdi parallell   i   a:99989305, paa: 41.913504 ms.
Max verdi sekvensiell i a:99989305, paa: 238.799921 ms.

Kjoering:1, ant kjerner:8, antTraader:8
Max verdi parallell   i   a:99989305, paa: 26.78024 ms.
Max verdi sekvensiell i a:99989305, paa: 235.431219 ms.

Kjoering:2, ant kjerner:8, antTraader:8
Max verdi parallell   i   a:99989305, paa: 27.791271 ms.
Max verdi sekvensiell i a:99989305, paa: 248.066478 ms.

Kjoering:3, ant kjerner:8, antTraader:8
Max verdi parallell   i   a:99989305, paa: 26.86283 ms.
Max verdi sekvensiell i a:99989305, paa: 236.013201 ms.

Kjoering:4, ant kjerner:8, antTraader:8
Max verdi parallell   i   a:99989305, paa: 27.755575 ms.
Max verdi sekvensiell i a:99989305, paa: 223.535073 ms.


Median sequential time:236.013201, median parallel time:27.755575,
 n=  100000000, Speedup:  8.59
```

n= 100 mill

JIT-kompilering +optimalisering

Søppel-tømming

Hva betyr dette for tidsmålingene

- Første gangen vi gjører er tiden vi måler en sum av:
  - Først litt interpretering av bytekode
  - Så oversetting(kompilering) av hyppig brukte metoder til maskinkode
  - kjøring av resten av programmet dels i maskinkode.
- Andre gang vi kjører, kan følgende skje:
  - JVM finner at noen av maskinkompilerte metodene våre må optimaliseres ytterligere
  - Kjøretiden synker ytterligere
- Tredje gang er som oftest optimaliseringa ferdig, men ytterligere optimalisering kan bli gjort
- Tidtakningen vår må endres !
- Vi kjører det sekvensielle og parallelle programmet f.eks 9 ganger i en løkke , noterer alle kjøretider i to arrayer som så sorteres og vi velger medianverdien = a[(a.length-1)/2]
- Du får aldri samme svaret to ganger – mye variasjon !!

FinnMax 3 ulike kjøringer (samme parametre , varierer antall tråder: 8, 16, 4)

Uke2>java FinnM 1000000 9
Kjøring:0**, ant kjerner:8, antTråder:8**
Max verdi parallell i a:999216, paa: 23.860968 ms.
Max verdi sekvensiell i a:999216, paa: 3.468803 ms.

Kjøring:1, ant kjerner:8, antTråder:8
Max verdi parallell i a:999216, paa: 0.311465 ms.
Max verdi sekvensiell i a:999216, paa: 0.549437 ms.
……..
Kjøring:8, ant kjerner:8, antTråder:8
Max verdi parallell i a:999216, paa: 0.422752 ms.
Max verdi sekvensiell i a:999216, paa: 0.532639 ms.

Median sequential time:0.52004,
median parallel time:0.429051,
Speedup: **1.26,** n = 1000000

Uke2>java FinnM 1000000 9
Kjøring:0, **ant kjerner:8, antTråder:4**
Max verdi parallell i a:999216, paa: 16.154151 ms.
Max verdi sekvensiell i a:999216, paa: 3.75507 ms.

Kjøring:1, ant kjerner:8, antTråder:4
Max verdi parallell i a:999216, paa: 1.280854 ms.
Max verdi sekvensiell i a:999216, paa: 0.520741 ms.

Kjøring:2, ant kjerner:8, antTråder:4
Max verdi parallell i a:999216, paa: 0.557136 ms.
Max verdi sekvensiell i a:999216, paa: 0.509191 ms.
……..
Kjøring:8, ant kjerner:8, antTråder:4
Max verdi parallell i a:999216, paa: 0.628527 ms.
Max verdi sekvensiell i a:999216, paa: 0.52354 ms.

Median sequential time:0.520741, median parallel time:0.628527,
Speedup: **0.88**, n = 1000000

Uke2>java FinnM 1000000 9
Kjøring:0, **ant kjerner:8, antTråder:16**
Max verdi parallell i a:999216, paa: 18.808946 ms.
Max verdi sekvensiell i a:999216, paa: 3.558043 ms.

Kjøring:1, ant kjerner:8, antTråder:16
Max verdi parallell i a:999216, paa: 1.847439 ms.
Max verdi sekvensiell i a:999216, paa: 0.453898 ms.
……..
Kjøring:8, ant kjerner:8, antTråder:16
Max verdi parallell i a:999216, paa: 0.502542 ms.
Max verdi sekvensiell i a:999216, paa: 0.471396 ms.

Median sequential time:0.509891,
median parallel time:0.646726,
Speedup: **0.90,** n = 1000000

«Aldri» samme resultatet to ganger

Uke2>java FinnM 1000000 9
ant kjerner:8, antTråder:8, n = 1mill

Med antall kjøringer for median = 9
1) Speedup: **0.68**, n = 1000000
2) Speedup: 0.96, n = 1000000
3) Speedup: 0.84, n = 1000000
4) Speedup: 0.71, n = 1000000
5) Speedup: 1.06, n = 1000000
6) Speedup: 1.26**,** n = 1000000

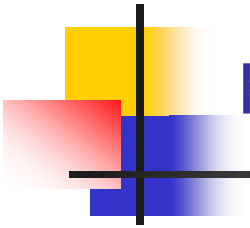
Med antall kjøringer for median = 21
7) Speedup: 1.00, n = 1000000
8) Speedup: 0.84, n = 1000000
9) Speedup: 0.88, n = 1000000
10) Speedup: **1.75**, n = 1000000
11) Speedup: 0.87, n = 1000000
12) Speedup: 1.11, n = 1000000
13) Speedup: 1.03, n = 1000000

Konklusjon på JIT-kompilering

- JIT-kompilering kan skrues av med >java –Xint MittProg ..
  - Brukes bare for debugging
- JIT kompilering kan gi 10 til 30 ganger så rask eksekvering for liten n (en god del mer for stor n)
- Første, andre (og tredje) kjøring er tidsmessig sterkt misvisende
- Vi må:
  - Kjøre programmet i en løkke f.eks 9 (eller 7 eller 11) ganger
  - Legge tidene i hver sin array (sekvensielt og parallell tid)
  - Sortere arrayene
  - Ta ut medianen (element (length-1)/2), som blir vår tidsmåling

```
import java.util.concurrent.*;
import java.util.*;
class Problem2 { int [] fellesData  ; // dette er felles, delte data for alle trådene
      double [] tidene ;
      int ant, svar;
      public static void main(String [] args) {
            ( new Problem()).utfoer(args);
      }
      void utfoer (String [] args) {
            ant = new Integer(args[0]);
            fellesData = new int [ant];
            tidene = new double[9];
            for (int m = 0; m <9; m++) {
                        long tid = System.nanoTime();
                        Thread t = new Thread(new Arbeider());
                         t.start();
                         try{t.join();}catch (Exception e) {return;}
                        tidene[m] = (System.nanoTime() -tid)/1000000.0;
                        System.out.println("Tid for "+m + ", tråd:"+tidene[m]+"millisec");
            }
            Arrays.sort(tidene);
            System.out.println("Median med svar:"+svar+", for trådene:"+tidene[(tidene.length-1)/2]+" millisec");
       } // end utfoer

       class Arbeider implements Runnable {
          int i,lokalData;    // dette er lokale data for hver tråd
                public void run() {     int sum =0;
                                        for (int i = 0; i < ant; i++) sum +=fellesData[i];
                                        svar =sum;
                }
    } // end indre klasse Arbeider
} // end class Problem
```



Hva med operativsystemet:

- Linux og Windows har om lag like rask implementasjon av Java og trådprogrammering,
- Dag Langmyhr testet to helt like maskiner med hhv. Linux og Windows, og resultatene tidsmessig (medianer) var nesten helt like, men
  - Ulike maskiner som Ifis store servere (diamant, safir,..) har en annen Linux og en noe langsommere ytelse for korte, trådbaserte programmer.

Hva med søppeltømming – garbage collection:

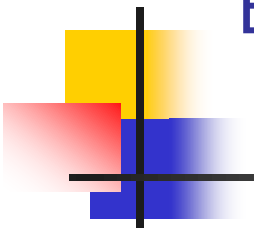
- Søppeltømming (=opprydding i lageret og fjerning av objekter vi ikke lenger kan bruke) kan slå til når som helst under kjøring:

```
Kjøring:2, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.35 msek. , nanosek/n:   35.07
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.36

Kjøring:3, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.57 msek. , nanosek/n:   56.87
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    0.66

Kjøring:4, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.43 msek. , nanosek/n:   43.47
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.33

Kjøring:5, ant kjerner:8, antTråder:8
Max para = a:9853, paa:     0.49 msek. , nanosek/n:   49.20
Max sekv = a:9853, paa:     0.01 msek. , nanosek/n:    1.36
```



End of lecture L05v26



IN3030 L06v26 – Prime Numbers Review and Intro to Synchronization

Eric Jul and Paulo Ferreira
Programming Technology Group
Department of Informatics
University of Oslo

1

1



Summary for L05v26

- Parallelization hints
- Amdahl's laws
- Prime numbers
- Timing
  - JIT compilation
  - Operating system
  - Garbage collection

2

2



Plan for L06v26

- Review of Eratosthenes sieve
- Review of factorization of numbers into primes
- Oblig 3
- Intro to synchronization

3

3



About prime numbers

- Prime Numbers and Factorization of Non-Prime Numbers
- A prime number is: An integer that can only be divided by 1 and itself.
  - 1 is not a prime number (many people thought so in the 18th century, and some still do)

4

4

Several ways to create prime numbers

- Want to find all prime numbers pi < N
- Divide all numbers < N by all numbers < N
  - Division method
  - Only odd numbers (2 special)
  - Only up to $\sqrt{N}$
- Make a table of all the prime numbers we need
  - Sieve of Eratosthenes

5

5

About prime numbers and factorization of integers

- Any integer N > 1 can be factorized as a product of prime numbers:
  - N = p1*p2*p3*……*pk
  - This factorization is unique (up to order)
  - is made unique if the numbers in the factorization are sorted
  - If there is only one number in this factorization, then N itself is a prime number
- Example: factorization of 532
  - 532 = 2 * 266
  - 532 = 266*2 = 2 * 2 * 133
  - 532 = 266*2 = 2 * 2 * 133 = 2 * 2 * 7 * 19

6

6

Prime numbers according to sieve of Eratosthenes

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75	77	79
81	83	85	87	89
91	93	95	97	99

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75	77	79
81	83	85	87	89
91	93	95	97	99

- Checking for 3, then 3*3, 3*5, 3*7, …

7

7

Checking for 5

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75	77	79
81	83	85	87	89
91	93	95	97	99

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45 45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75 75	77	79
81	83	85	87	89
91	93	95	97	99

- Checking for 5 starting with 25, then 5*5, 5*7,…

8

8

Check for 7

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45 45	47	49
51	53	55	57	59
61	63	65	67	69
71	73	75 75	77	79
81	83	85	87	89
91	93	95	97	99

1	3	5	7	9
11	13	15	17	19
21	23	25	27	29
31	33	35	37	39
41	43	45 45	47	49
51	53	55	57	59
61	63 63	65	67	69
71	73	75 75	77	79
81	83	85	87	89
91	93	95	97	99

- Check for 7 starts with 49, so 7*7, 7*9, 7*11…

Now, we finished because the next prime number we find is: 11, so 11*11=121 is outside the table

The primes are the black numbers and those shaded in orange.

9

9

Factoring a number into its prime factors

- We have created and stored using Erotosthanes' sieve all (except 2) prime numbers < N in a bit array over all odd numbers
  - 1 = prime, 0 = non-prime
  - We have crossed out those that are not prime
- How should we then use this to factor a number M < N*N ?
- Answer: Divide M by all prime numbers pi<SQUARE ROOT(M) (pi =2,3,5,..), and each time such a division M%pi ==0, then pi is one of the factors of M. We then continue by factoring a smaller number M'=M/pi.
  - The factorization of M = pi *..* pk is then the product of all the prime numbers that divide M without a remainder.
  - REMEMBER that pi can occur several times in the answer. ex: 20= 2*2*5, 81 = 3*3*3*3, etc.
  - If we find no factorization of M, i.e. no pi ≤ SQUARE ROOT (M) that divides M with remainder == 0, then M itself is a prime number

10

10



"Never" the same result twice

Week2>java FinnM 1000000 9number of cores:8,
number of threads:8, n = 1million

With number of runs for median = 9

1) Speedup: 0.68, n = 1000000
2) Speedup: 0.96, n = 1000000
3) Speedup: 0.84, n = 1000000
4) Speedup: 0.71, n = 1000000
5) Speedup: 1.06, n = 1000000
6) Speedup: 1.26, n = 1000000

it is the value at position (n+1)/2
always sort the values in ascending order first

With number of runs for median = 21

7) Speedup: 1.00, n = 1000000
8) Speedup: 0.84, n = 1000000
9) Speedup: 0.88, n = 1000000
10) Speedup: 1.75, n = 1000000
11) Speedup: 0.87, n = 1000000
12) Speedup: 1.11, n = 1000000
13) Speedup: 1.03, n = 1000000

it is the value at position (n+1)/2
always sort the values in ascending order first

11

11



Oblig 3

12

12

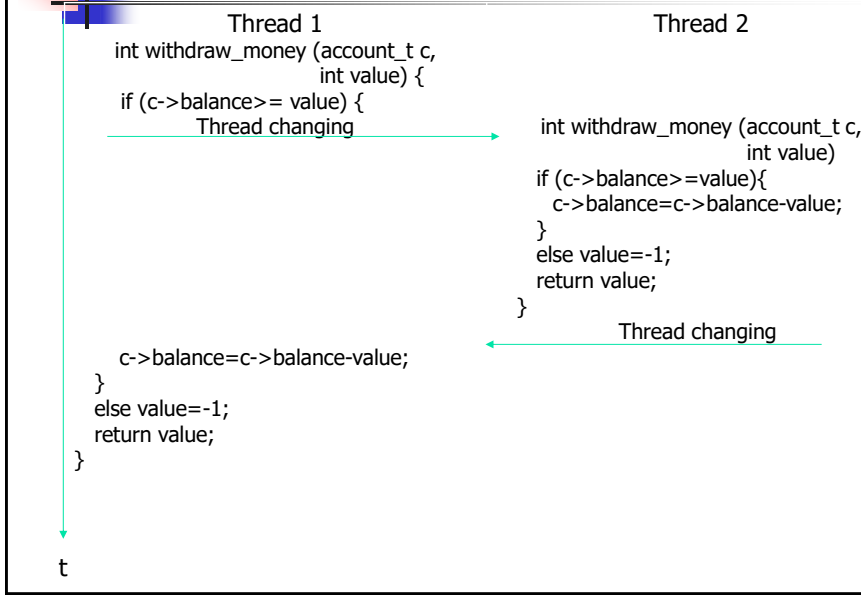
Concurrent Execution

```
struct {
   int balance;
   /* other variables, e.g., name, etc. */
} account_t;

int withdraw_money (account_t* c, int value) {
   if (c->balance >= value)
      c->balance = c->balance - value;
   else
      value = -1; /* -1 indicates an error occurred */
   return value;
}
```

13

Concurrent Execution



14



Programs in Assembly have more Operations

decrement VALUE to BALANCE, i.e. BALANCE= BALANCE - VALUE

```
; assuming that the variable c->balance is in memory position BALANCE
; assuming that the variable value is in memory position VALUE

mov AX, BALANCE; loads memory position BALANCE to
                 general registry AX
mov BX, VALUE  ; loads memory position VALUE to
                 general registry BX
sub AX, BX     ; subtracts AX = AX – BX
mov BALANCE, AX; write result from subtraction in
                 the memory position BALANCE
```

15



Critical Section

In concurrent programming every time you test or modify shared data you must do it in a critical section.

16



Critical Section

```
int withdraw_money (account_t *c, int value)
{
  close(); /* lock() */
  if (c->balance>= value) {
    c->balance= c->balance–value;
  } else value = -1;
  open(); /* unlock */
  return value;
}
```

Critical Section (executed in mutual exclusion)

17



Critical Section: Properties

- Mutual exclusion
- Progress (liveness):
  - deadlock
  - starvation
  - efficient

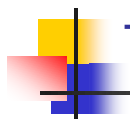
18



Critical Section: Possible Implementations

- Algorithmics
- Hardware
- Operating System

19



Try Solving #1

```
int latch = OPEN;

close() {
   while (latch == CLOSED)   {/* empty instruction*/};
   latch = CLOSED;
}

open () {
   latch = OPEN;
}
```

- When 1 executes the function close(), we can see that the test of variable latch (done in the while cycle) and the setting of latch = CLOSED; (done in the following instruction) is not done atomically
- It may happen that T1 losses the processor immediatly after having left the while cycle and before seting the latch = CLOSED;
- If T2 executes the function close(), this one can allow the entry in the Critical Section (as the variable latch is still with the value OPEN)

20

Try Solving #2

```
int latch_t1 = OPEN;
int latch_t2 = OPEN;
```

Thread T1

```
t1_close () {
  while (latch _t2 == CLOSED);
  latch _t1 = CLOSED;
}

t1_open() {latch _t1 = OPEN;}
```

Thread T2

```
t2_close ( ) {
  while (latch _t1 == CLOSED);
  latch _t2 = CLOSED;
}

t2_open() {latch _t2 = OPEN;}
```

- Considering t1, we can see that the test of latch _t2 (done in the while cycle) the setting of latch _t1 = CLOSED; is not done atomically
- If can happen that T1 losses the processor immediatly after leaving the cycle and before executing the setting of latch _t1 = CLOSED;
- If T2 execute the function t2_close ( ), this can allow the entry of the Critical Section (as the latch _t1 is still with the value OPEN)

21

Try Solving #3 (equal to #2 but with lines exchanged)

```
int latch _t1 = OPEN;
int latch _t2 = OPEN;
```

Thread T1

```
t1_close () {
  latch _t1 = CLOSED;
  while (latch _t2 == CLOSED);
}

t1_open () {latch _t1 = OPEN;}
```

Thread T2

```
t2_close ( ) {
  latch _t2 = CLOSED;
  while (latch _t1 == CLOSED);
 }

t2_open () {latch _t2 = OPEN;}
```

- It is enough that T1 runs the instruction latch _t1 = CLOSED; and losess the processor immediately after !
- When T2 runs, this thread will be in a cycle while (latch _t1 == CLOSED);
- The same will happen to thread T1 when it runs again, i.e., it will be in cycle in instruction while (latch _t2 == CLOSED);

22



Try Solving #4

int latch _turn = 1;

Thread T1

```
t1_close () {
   while (latch _turn == 2);
}

t1_open () {latch _turn = 2;}
```

Thread T2

```
t2_close () {
   while (latch _turn == 1);
}

t2_abrir () {latch _turn = 1;}
```

- The thread T2 (T1) can block outside the critical section, given the need for alternancy, prevents T1 (T2) from executing the Critical Section in the future
- In the ausence of contention, if thread T1 wants to get into the Critical Section, it can not do that in an efficient way, as will have to wait for T2 to execute the Critical Section
- So, this solution has the following problems:
  - progress - not fulfilled;
  - eficiency - not fulfilled.

23



ALGORITHMIC SOLUTIONS

24

Peterson Algorithm

```
int latch_t1 = OPEN;
int latch _t2 = OPEN;
int thread_prio = 1;

t1_close () {
   A1: latch _t1 = CLOSED;
   B1: tar_prio = 2;
   C1: while (latch _t2 == CLOSED && thread _prio == 2);
}

t1_open () {latch _t1 = OPEN;}

t2_close () {
   A1: latch _t2 = CLOSED;
   B1: tar_prio = 1;
   C1: while (latch _t1 == CLOSED && thread_prio == 1);
}

t2_ open () {latch _t2 = OPEN;}
```

T1 wants to access the Critical Section

T2 is prioritized

25

Lamport Algorithm (Bakery)

```
int ticket[N];    // Initialized to 0
int choice[N];    // Initialized to FALSE

close (int i) {
  int j;
  choice[i] = TRUE;
  ticket [i] = 1 + maxn(ticket);
  choice[i] = FALSE;

  for (j=0; j<N; j++) {
    if (j==i) continue;
    while (choice[j]) ;
    while (ticket [j] && (ticket [j] < ticket [i]) ||
           (ticket [i] == ticket [j] && j < i)));
  }
}

open (int i) {ticket [i] = 0;}
```

- tikect contains he number given to the thread
- choice indicates it the thread want to access the Critical Section

•Pi indicates that he is choosing a ticket
•Pi choses a ticket bigger than others
•Pi announces that it has already chosen a ticket

•Pi checks if it has the ticket less then other tickets of Pj

•In case Pj is choosing a ticket, wait for it

•Pj has already chosen a ticket or not:
•If it has already chosen, Pi will check if it has a ticket smaller than that of Pj
•If it has not chosen yet, Pj will see from Pi ticket and chose a bigger one

•If the ticket from Pi is less that the others, then Pi enters
•If the tickets have the same value, goes into the Critical Section the one with the least identifier

26



Algorithmic Solutions

- Conclusion:
  - They are complex → latency
  - They only contemplate active waiting
- Solution:
  - Introduce hardware instructions to make the solution easy

27



HARDWARE SOLUTIONS

28



Hardware Solutions

- Open( ) and Close( ) use special instructions offered by the processors:
  - Disable the interruptions
  - xchg on Intel
  - Test-and-set (cmpxchg on Intel)

29



Mutal Exclusion by Disabling the Interruptions

```
int mask;
```

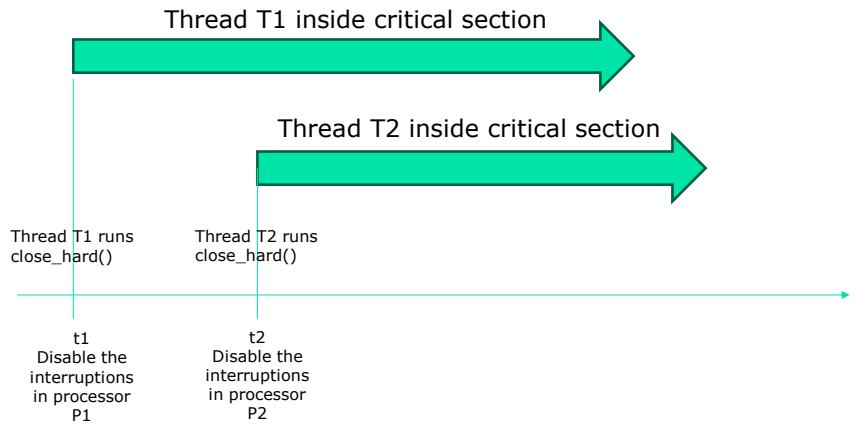
```
close_hard () {
   mask = mask_int ();
}
```

```
open_hard () {
   putback_int (mask);
}
```

- This should be used inside the OS in very short critical sections:
  - Disabling the interruptions prevents other services to run (I/O, etc.)
  - If the program forgets to call open_hard () the interruptions are disabled and the system stops
- It does not work in multiprocessors

30

Disable the Interruptions does not work in Multiprocessors



31

Test-and-Set

```
OPEN    EQU 0           ; OPEN is the value 0
CLOSED EQU 1            ; CLOSED is the value 1

MOV OPEN, AX            ; AX is initialized with the value 0 (OPEN)

close_hard:
L1:     BTS latch, AX   ; test if latch and AX are equal
        JC L1           ; if carry flag is 1, than latch was CLOSED
                        ; implies going to L1 and try again
                        ; if carry flag is 0, then latch was OPEN
                        ; and the latch is set to the value 1 (CLOSED)
        RET             ;

open_hard:
        MOV   OPEN, AX
        MOV   AX, latch
        RET
```

32

XCHG

```
OPEN    EQU     0       ; OPEN is with the valued 0
CLOSED EQU      1       ; CLOSED is with the valued 1

MOV AX, latch           ; latch initialized with the value OPEN

close_hard:
        XCHG latch, AX
L1:     CMP  AX, OPEN   ; compares the value of AX with 0
        JNZ  L1         ; if AX is different from zero it returns to L1
        RET

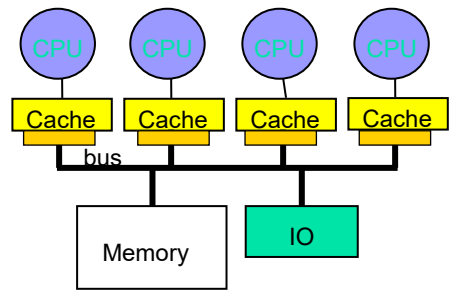
open_hard:
        MOV OPEN, AX
        MOV AX, latch
        RET
```

- The instruction XCHG makes sure that the lactch is closed:
  - it wrote the value 1 (that was in the registry AX) in memory position latch (bus lachted)
- Any other thread that runs the instruction CMP AX, OPEN is going to detect that the latch is with value 1 (is CLOSED)

33

XCHG in Multiprocessors

	P1	P2
Moment 1	P1 block the bus	
Moment 2	P1 blocks the critical section	P2 tries to do the exchange but blocks itself when obtaining the bus
Moment 3	P1 goes into the critical section	P2 checks that the latch is CLOSED and stays in active waiting



34



Solutions with Hardware Support

- Conclusion:
  - They offer the basic mechanims to implement mutual exclusion, but …
  - Some can not be used directly by programs in user mode:
    - e.g.., disable the interruptions
  - Others only contemplate active waiting:
    - e.g., exchange, test-and-set

35



SOLUTIONS WITH THE SUPPORT FROM OS

36



Solutions with the Support from OS

- Synchronization primitives:
  - Software trap (interruption SW)
  - Changing to nucleus mode
  - Data structures and synchronization code belongs to the nucleus
  - It uses the hardware support (exch. / test-and-set)
- We will see two synchronization primitives:
  - latches
  - semaphores

37

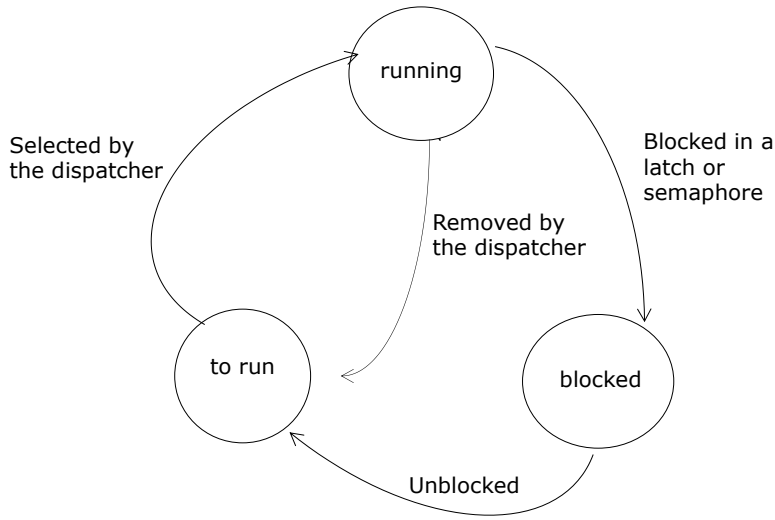


Solutions with Support from the OS

- Advantages (intuitively):
  - if the OS know which processes/threads are waiting in the critical section, then it does not give them processor time
  - prevents active waiting!
- Hardware solution can be used, but by code of the OS:
  - inside system calls

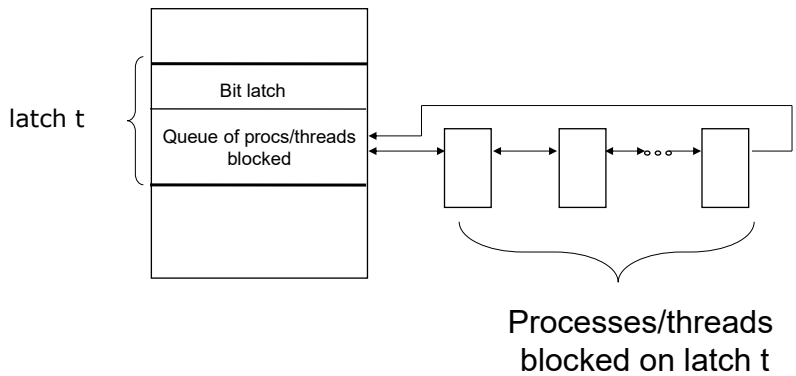
38

State Diagram of Processes/Threads



39

Data Structures Associated to Latches



40

Memory Allocator – example of concurrent programming

```
#define MAX_STACK 100
char*    stack[MAX_STACK];
int      top = MAX_STACK-1;

char *ask_for_mem() {
  ptr = stack[top];
  top--;
  return ptr;
}

void get_back_mem() {
  top++;
  stack[top] = ptr;
}
```

The program is wrong…

… because the data structures are not accessed atomically

41

Memory Allocator with Critical Section

```
#define MAX_STACK 100
char*    stack[MAX_STACK];
int      top = MAX_STACK-1;
latch_t  mutex = OPEN;

char *ask_for_mem() {
  close (mutex);
  ptr = stack[top];
  top--;
  open (mutex);
  return ptr;
}

void get_back_mem() {
  close (mutex);
  top++;
  stack[top] = ptr;
  open (mutex);
}
```

A latch is created always in the OPEN state

At the start of the critical section, the procs/threads must call close (mutex);

If the latch is CLOSED, the blocking proc/thread waits for the other to leave the critical section

If it is OPEN, pass the latch to CLOSED.

These operations are done atomically

At end of the critical section the procs/threads must call open (mutex)

The latch becomes OPEN and unblocks a proc/thread that is waiting to go into the critical section

42

Latches – Limitations

- Latches are not expressive enough to solve some synchronization problems
- E.g.: in the memory allocator blocks procs/threads if the stack is full:
  - Need a "resource counter" -> it blocks the calling proc/thread if the number of requests exceed some limit

43

Memory Allocator with Critical Section

```
#define MAX_STACK 100
char*    stack[MAX_STACK];
int      top = MAX_STACK-1;
latch_t  mutex = OPEN;

char *ask_for_mem() {
  close (mutex);
  ptr = stack[top];
  top--;
  open (mutex);
  return ptr;
}

void get_back_mem() {
  close (mutex);
  top++;
  stack[top] = ptr;
  open (mutex);
}
```

What if the stack is full ?

A latch is created always in the OPEN state

At the start of the critical section, the procs/threads must call close (mutex);

If the latch is CLOSED, the blocking proc/thread waits for the other to leave the critical section

If it is OPEN, pass the latch to CLOSED.

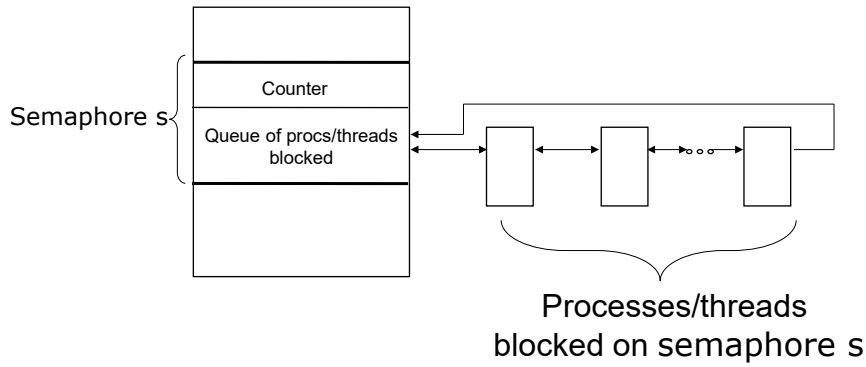
These operations are done atomically

At end of the critical section the procs/threads must call open (mutex)

The latch becomes OPEN and unblocks a proc/thread that is waiting to go into the critical section

44

Semaphores



- Never do the analogy with the traffic semaphore !!!

45

Semaphores: Primitives

```
// Creating a Semaphore object with a number of permits
Semaphore sem = new Semaphore(permits);

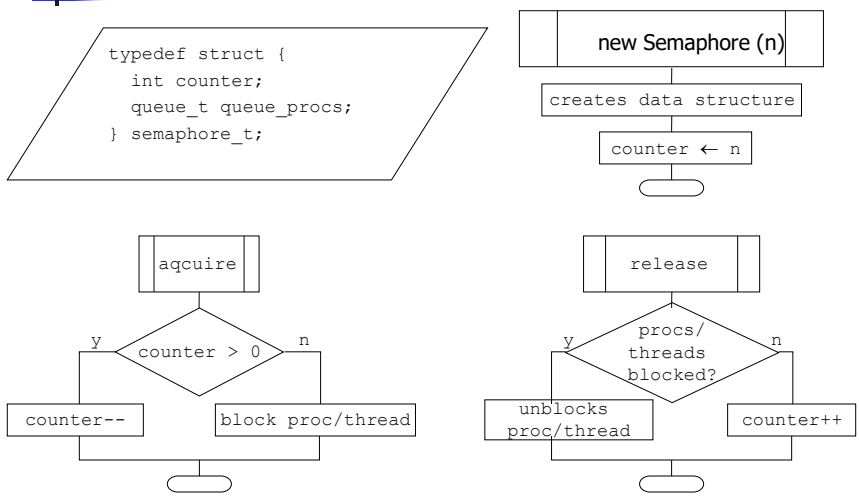
// Blocks the proc/thread if the counter is less or equal to zero; otherwise, it decrements the counter
sem.acquire();

// If there are procs/threads blocked, it releases one; // otherwise, it increments the counter
sem.release();
```

- All the primitives are executed atomically
- release() and acquire() can be called by different processess

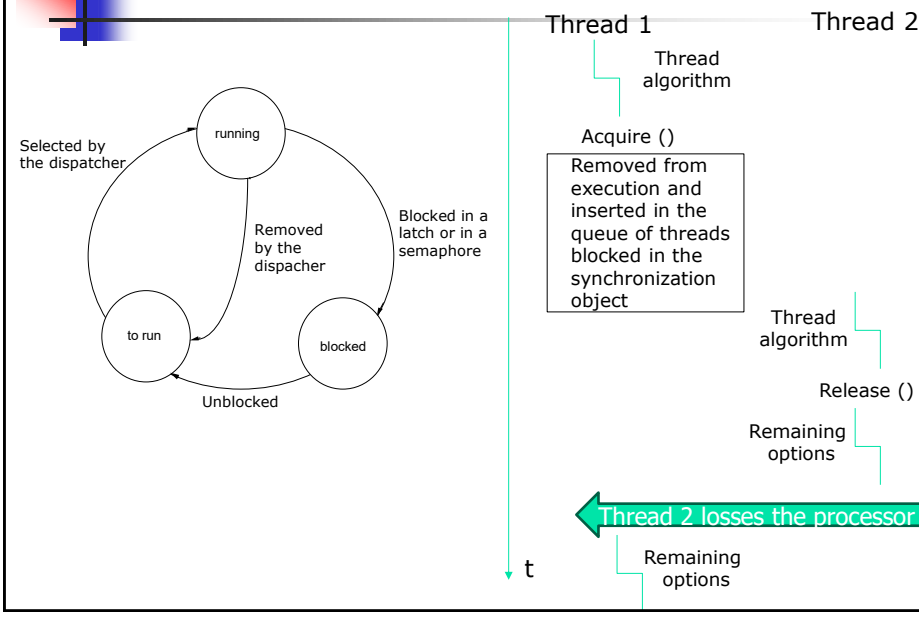
46

Semaphores: Primitives



47

Semaphores: Primitives



48

Semaphores (example)

```
#define MAX_STACK 100
char*    stack[MAX_STACK];
int      top = MAX_STACK-1;
Semaphore SemMutEx;
Semaphore SemMem;

char *ask_for_mem() {
  acquire(SemMem);
  acquire(SemMutEx);
  ptr = stack[top];
  top--;
  release(SemMutEx);
  return ptr;
}

void get_back_mem() {
  acquire(SemMutEx);
  top++;
  stack[top] = ptr;
  release(SemMutEx);
  release(SemMem);
}
```

Memory Allocator with semaphore SemMem that controls the free memory !

```
main(){
/*...*/
  semMutEx = new Semaphore(1);
  semMem = new Semaphore(MAX_STACK);
}
```

The semaphore is initialized with the resources available !

49

Semaphores (example)

```
#define MAX_STACK 100
char*    stack[MAX_STACK];
int      top = MAX_STACK-1;
Semaphore SemMutEx;
Semaphore SemMem;

char *ask_for_mem() {
  acquire(SemMem);
  acquire(SemMutEx);
  ptr = stack[top];
  top--;
  release(SemMutEx);
  return ptr;
}

void get_back_mem() {
  acquire(SemMutEx);
  top++;
  stack[top] = ptr;
  release(SemMutEx);
  release(SemMem);
}
```

Memory Allocator with semaphore SemMem that controls the free memory !

```
main(){
/*...*/
  semMutEx = new Semaphore(1);
  semMem = new Semaphore(MAX_STACK);
}
```

What happens if the order of instructions changed ?

50

Semaphores (example)

```
#define MAX_STACK 100
char*    stack[MAX_STACK];
int      top = MAX_STACK-1;
Semaphore SemMutEx;
Semaphore SemMem;

char *ask_for_mem() {
  acquire(SemMem);
  acquire(SemMutEx);
  ptr = stack[top];
  top--;
  release(SemMutEx);
  return ptr;
}

void get_back_mem() {
  acquire(SemMutEx);
  top++;
  stack[top] = ptr;
  release(SemMutEx);
  release(SemMem);
}
```

Memory Allocator with semaphore SemMem that controls the free memory !

```
main(){
/*...*/
  semExMut = new Semaphore(1);
  semMem = new Semaphore(MAX_STACK);
}
```

Deadlok
The change of instructions creates na error situation !

51

Semaphores: Variants

- Generic:
  - acquire() releases any proc/thread from the queue
- FIFO:
  - release() releases the proc/thread that has blocked the longest time
- Semaphores with priorities:
  - The proc/thread specifies when acquire() the priority,
  - acquire() releases the procs/threads by order of priorities
- Semaphore with units:
  - the primitives acquire() and release() allows the specification of the number of units to acquire or release

52



Interface POSIX

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       pthread_mutexattr_t *attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
                            const struct timespec *timeout);
```

Example:

```
pthread_mutex_t count_lock;

pthread_mutex_init(&count_lock, NULL);
pthread_mutex_lock(&count_lock);
count++;
pthread_mutex_unlock(&count_lock);
```

53



```
#include <stdio.h>
#include <pthread.h>
#define MAX_STACK 100
char*    stack[MAX_STACK];
int      top = MAX_STACK-1;
pthread_mutex_t semMutEx;
/* semaphore of mutal exclusion*/
```

```
char *ask_for_mem() {
  char *ptr;

  pthread_mutex_lock(&SemMutEx);
  if (top > 0) { /* check if there is memory available */
    ptr = stack[top];
    top--;
  }
  else {ptr = NULL; /* indicates error */}
  pthread_mutex_unlock(&SemMutEx);
  return ptr;
}

void get_back_mem() {
  pthread_mutex_lock(&SemMutEx);
  top++;
  stack[top] = ptr;
  pthread_mutex_unlock(&SemMutEx);
}

int main(int argc, char *argv[]) {
  pthread_mutex_init(&semExtMut, NULL); /* attr default */
  /* program */
  pthread_mutex_destroy(&semMutEx);
  return 0;
}
```

Memory Allocator with critical section programmed with mutex from POSIX library

54



Interface POSIX: Semaphores

```
int sem_init(sem_t *sem, int pshared, unsigned value);
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);

Example:

sem_t sharedsem;
sem_init(&sharedsem, 0, 1);
sem_wait(&sharedsem);
count++;
sem_post(&sharedsem);
```

55



End of lecture L06v25

56

56

Quantum Computations for Computer Systems Computer Scientists

Michael Kirkedal Thomsen
IN3030/IN4330, Apr 29, 2026

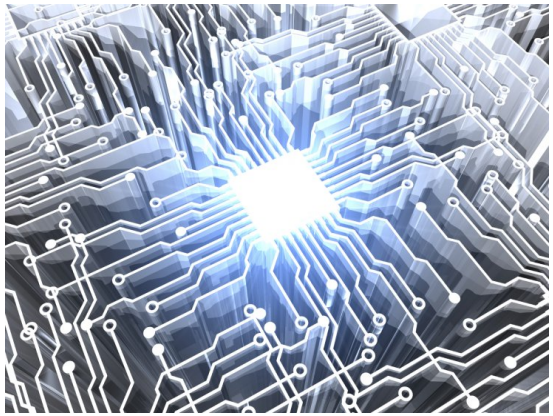
With material from:

Boris Düdder, DIKU, Univ. Cph
Anders Søndgaard Sørensen, NBI, Univ. Cph
Laura Mancinska, QMath, Univ. Cph
Michael Kastoryano, DIKU, Cph

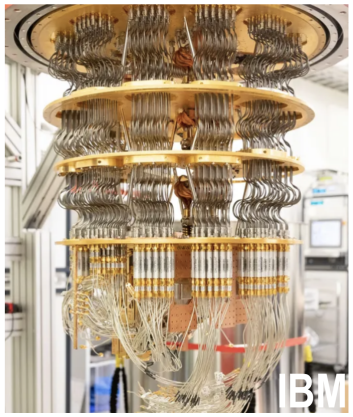
The promise of Quantum computing

Promise: Build and program an entirely new kind of computers based on quantum mechanics.

- more secure crypto (breaking)
- faster algorithms
- improved communication







Kvantemekanik er svært

Det kan da ikke passe



Albert Einstein

Jo det kan!



Niels Bohr

Fra filosofi til teknologi

- Bohr og Einstein: Mange diskussioner om betydning
  "Gud spiller ikke med terninger"

  Filosofi/religion

- I dag: kan lave eksperimenter med et atom Verden er bare mærkelig

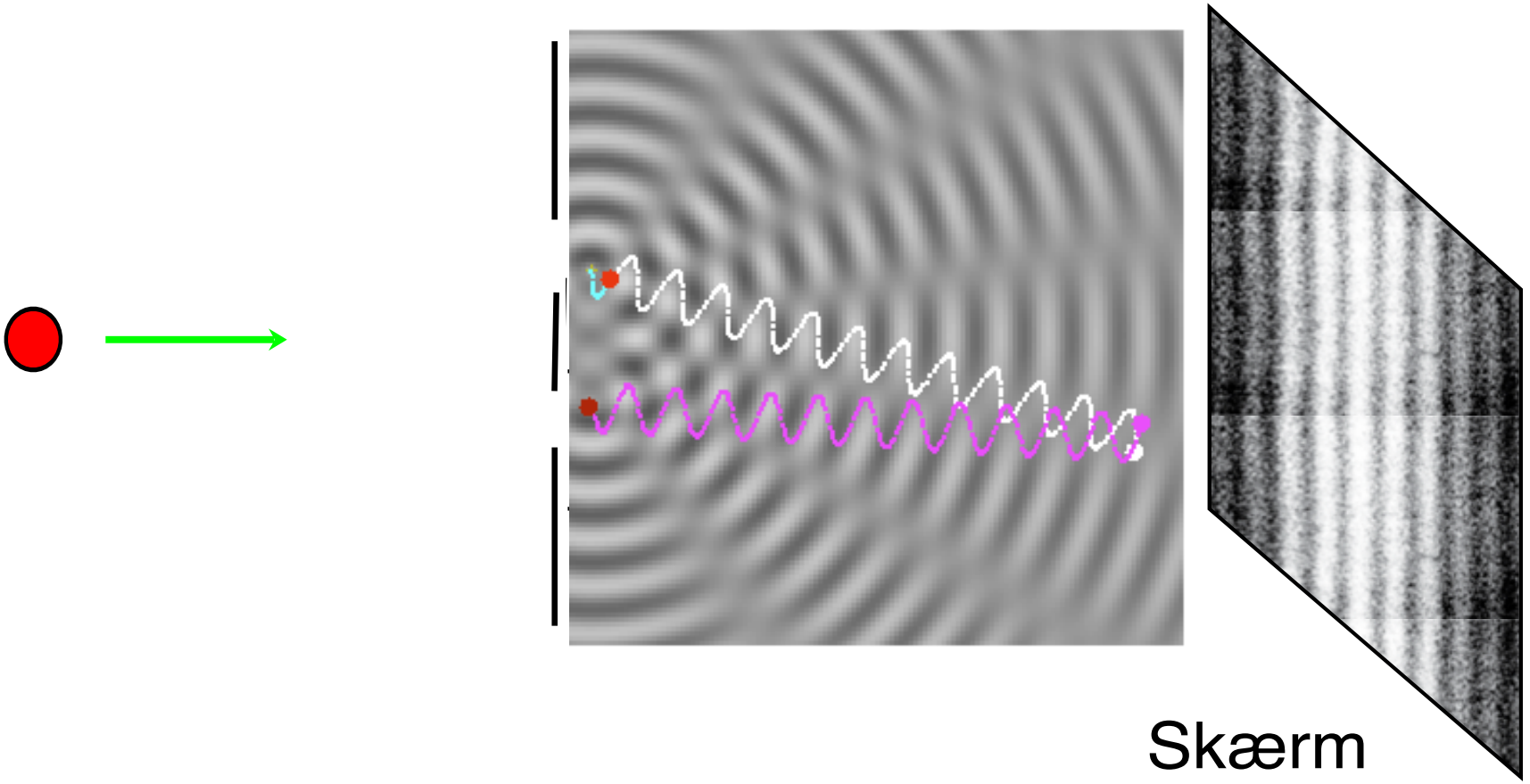
  Fysik

- Kan det mærkelig bruges til noget?

- Kvanteinformation: Gem en bit i et atom => nye muligheder

  Teknologi

Particle wave duality



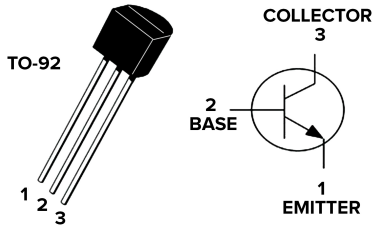
What is a Quantum Computing

Computer using quantum physical principles for computation

- Specialized algorithms for quantum computers
- Quantum Computer Programming languages and systems are different from traditional computer systems

Classical computing

Symbol	Truth Table		
 2-input NAND Gate	A	B	Q
	0	0	1
	0	1	1
	1	0	1
	1	1	0
Boolean Expression Q = A NAND B			



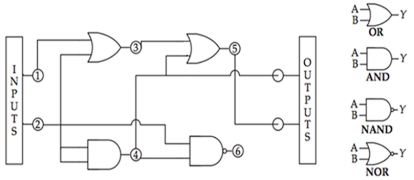
Formally: Input $\text{`}0\text{`} = \begin{pmatrix}1\\0\end{pmatrix}$ or $\text{`}1\text{`} = \begin{pmatrix}0\\1\end{pmatrix}$

Composition

$\text{`}01\text{`} = \text{`}0\text{`} \otimes \text{`}1\text{`}$

Operation NAND $\begin{pmatrix}1 & 1 & 1 & 0\\0 & 0 & 0 & 1\end{pmatrix}$

Larger computation

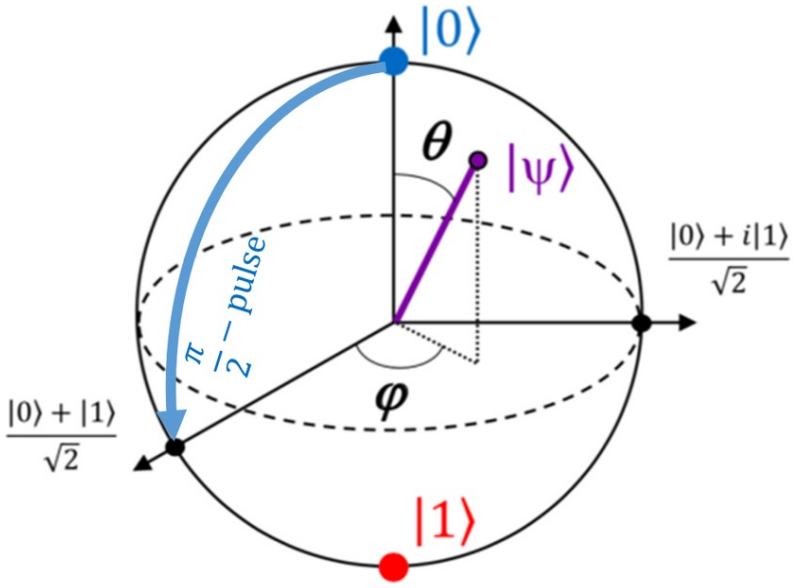


Deterministic input, deterministic output.

Operation errors $\mathcal{O}(10^{-15})$

Quantum computing

Input $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ $|\alpha|^2 + |\beta|^2 = 1$ $|0\rangle = \begin{pmatrix}1\\0\end{pmatrix}, \ |1\rangle = \begin{pmatrix}0\\1\end{pmatrix}$



$$|\psi\rangle = \cos\frac{\theta}{2}|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}|1\rangle$$

Quantum computing

Input $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ $|\alpha|^2 + |\beta|^2 = 1$ $|0\rangle = \begin{pmatrix}1\\0\end{pmatrix},\ |1\rangle = \begin{pmatrix}0\\1\end{pmatrix}$

Composition

$|01\rangle = |0\rangle \otimes |1\rangle$

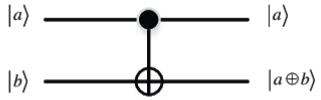
But also $|\phi\rangle = \begin{pmatrix}\phi_0\\\phi_1\\\phi_2\\\phi_3\end{pmatrix}$ $\sum_j |\phi_j|^2 = 1$

Legitimate input state

Like the square root of a probability vector.

Operation

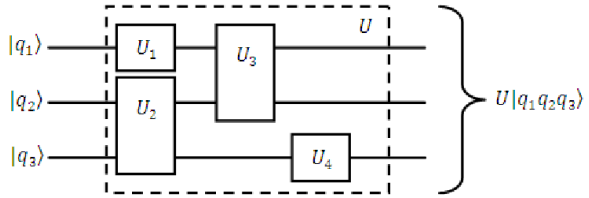
$$CNOT = \begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{bmatrix}$$



Deterministic input, deterministic output, but with quantum superposition and entanglement.

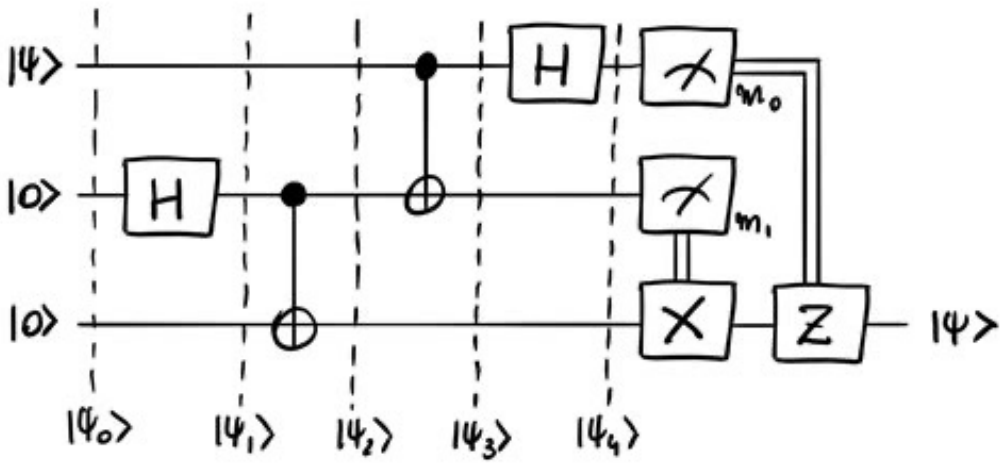
General: $U|\psi\rangle$ $UU^\dagger = \mathrm{id}$

Larger computation



Computations are large unitary matrices, decomposed as a sequence of smaller ones

Quantum circuit and running



Operator	Gate(s)	Matrix
Pauli-X (X)	X	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
Pauli-Y (Y)	Y	$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$
Pauli-Z (Z)	Z	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
Hadamard (H)	H	$\frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
Phase (S, P)	S	$\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$
$\pi/8$ (T)	T	$\begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$
Controlled Not (CNOT, CX)		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
Controlled Z (CZ)	Z	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$
SWAP		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Toffoli (CCNOT, CCX, TOFF)		$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$

What is a Quantum Computer

Computer using quantum physical principles for computation

- Specialized algorithms for quantum computers
- Quantum Computer Programming languages and systems are different from traditional computer systems

Principles:

- **Superposition** principle
- Quantum **entanglement**

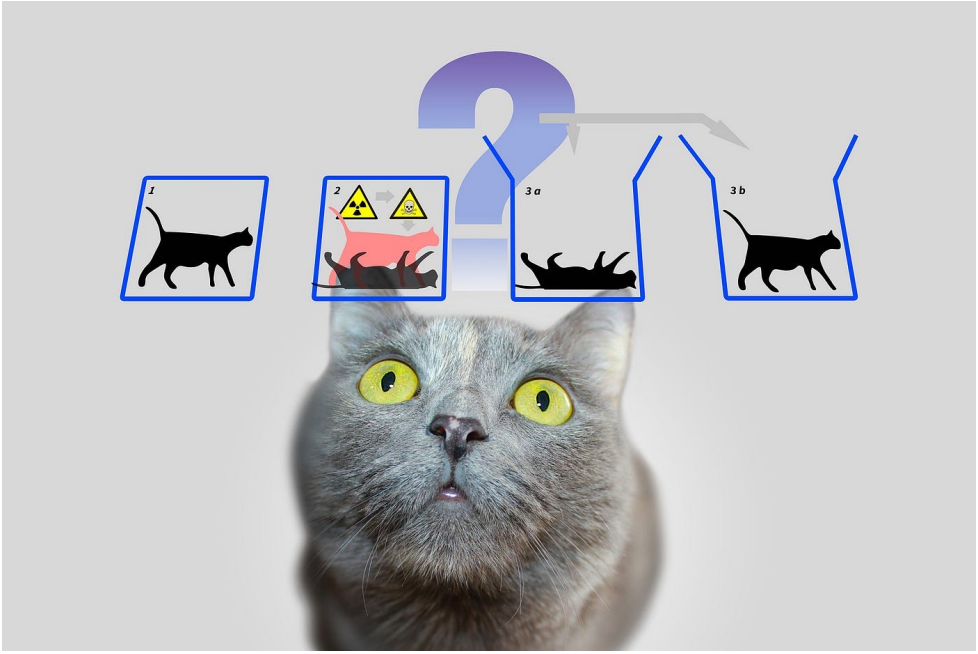
Superposition principle

Superposition principle: A quantum system consists of a superposition of equal physical quantities (these do **not** interfere with each other)

- Example: | 0> + | 1> = State is superposition of the two classic states "Bit is 0" and "Bit is 1". (Qubit = quantum bit)

Problem: **Decoherence**

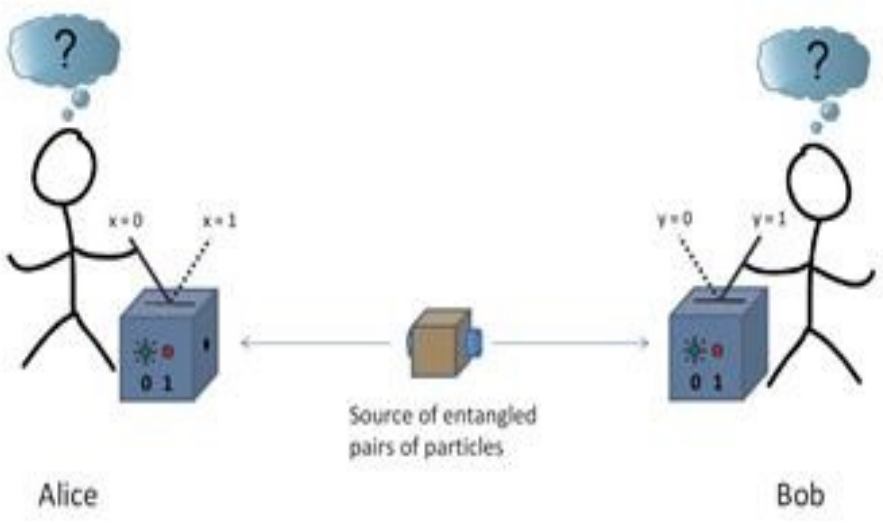
- Decoherence is agent of classicality: It describes the **loss** of quantum properties when a quantum system interacts with its surroundings (classical world).



Quantum entanglement

Quantum entanglement: The system state of two or more particles can not be described as a combination of independent one-particle states, but only by a **common** state

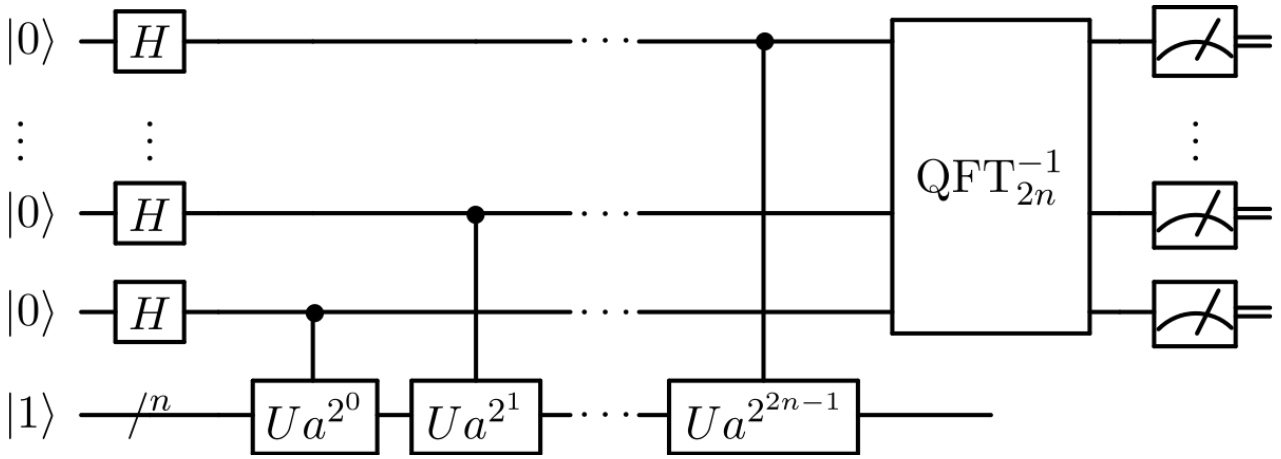
For example: a system of **two** entangled **spins** adding up to **zero** can be composed of a **superposition of pairs of spins** with opposite directions while it is absolutely undetermined in which direction the individual spin points.



Where does quantum computers get their power from?

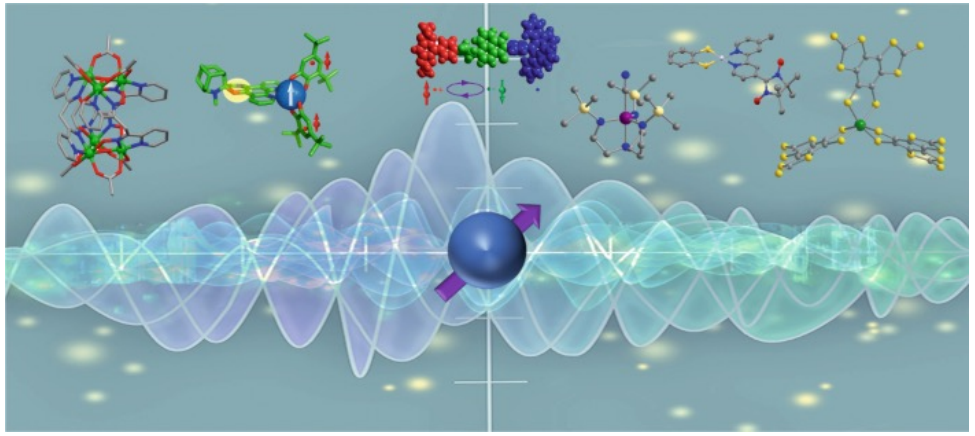
In a simple way, a combination of:

- Quantum entanglement
- Superposition

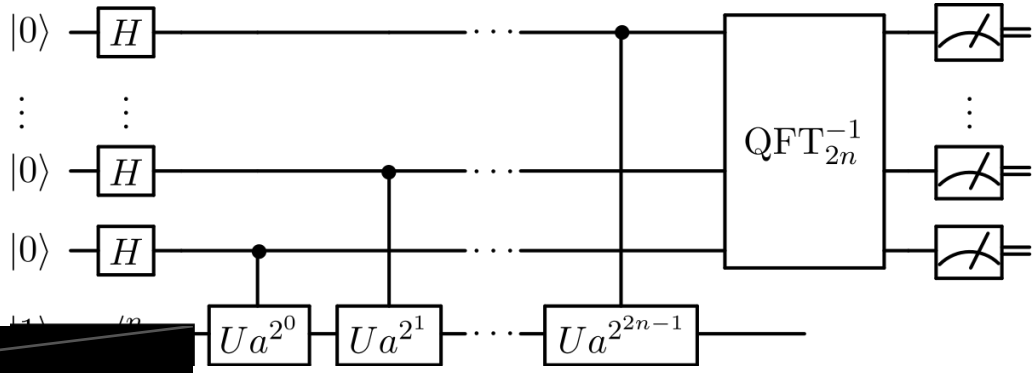


What can quantum computers do

- Quantum Fourier Transform
- Quantum simulations
  - Quantum dynamics
  - Quantum chemistry

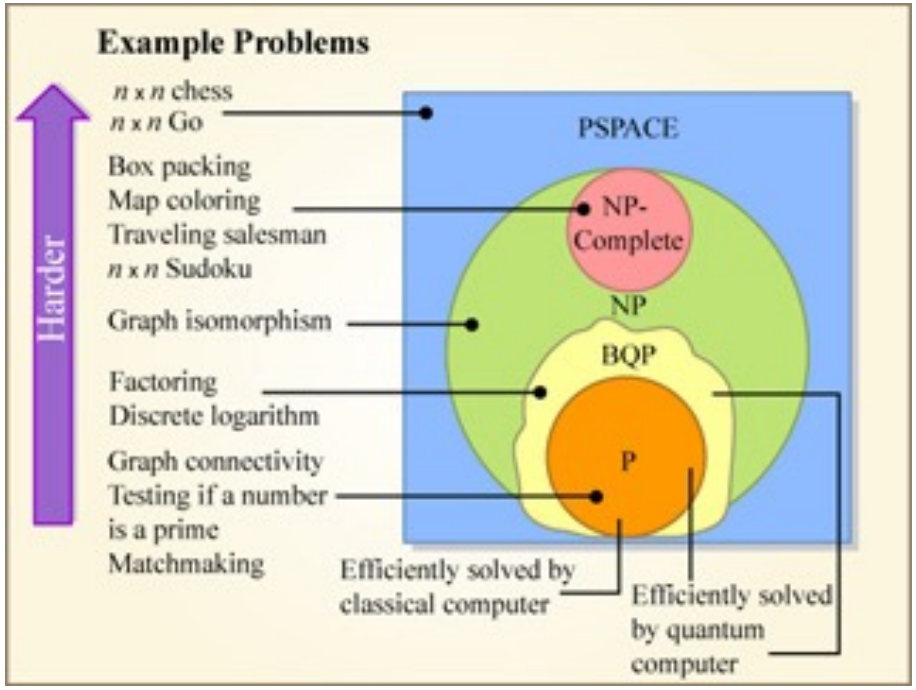


- Quantum algorithms
  - Shor's algorithm
  - Groover's search



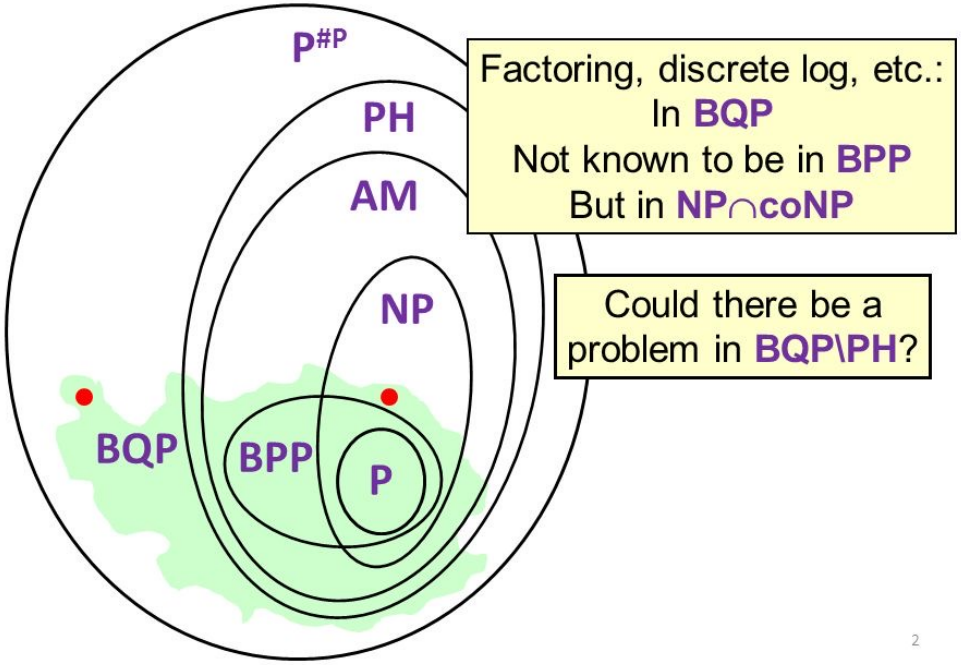
-

What can quantum computers do

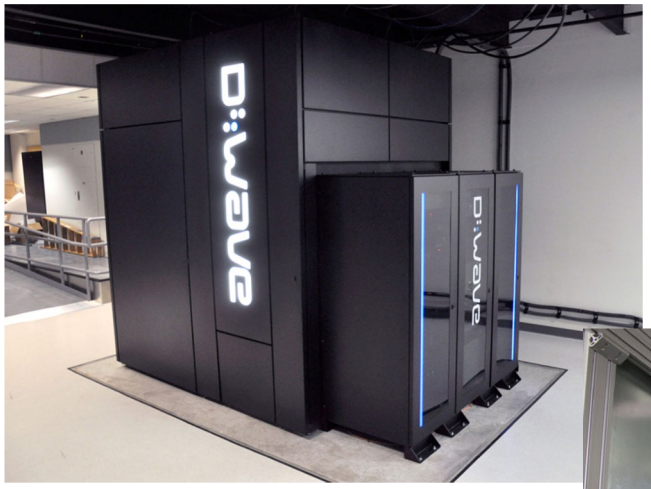


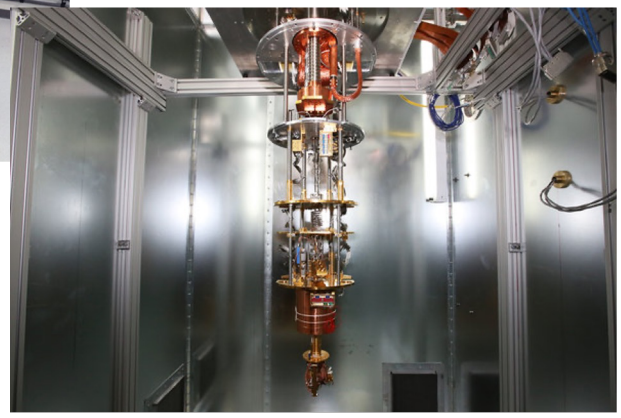
- Quantum machine learning?
- Quantum TSP?
- Quantum Finances

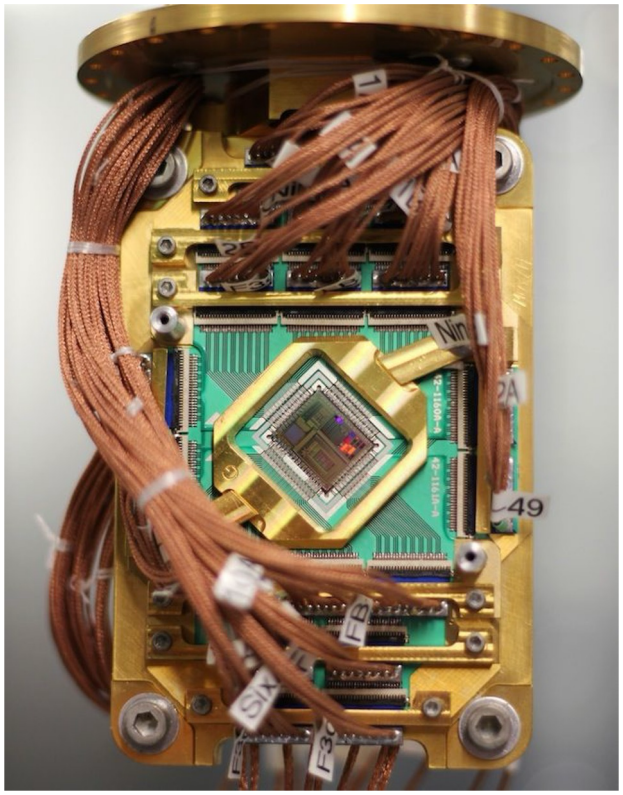
Quantum Computing: Where Does It Fit?



Quantum Computer



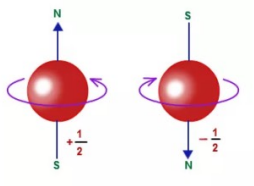


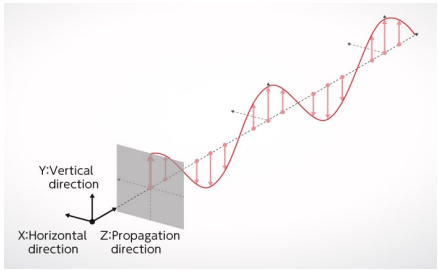


How do you build qubits?

Microscopic systems have mostly continuous degrees of freedom

Particle spin and light polarisation are discrete





Because quantum systems are very fragile, operations are very noisy

Qubit Type	Pros/Cons	Select Players
Superconducting	***Pros***: High gate speeds and fidelities. Can leverage standard lithographic processes. Among first qubit modalities so has a head start. ***Cons***: Requires cryogenic cooling; short coherence times; microwave interconnect frequencies still not well understood.	rigetti, Google, IBM Q, QuTech, OQC, IQM, qci Quantum Circuits, Inc, 本源量子 Origin Quantum
Trapped Ions	***Pros***: Extremely high gate fidelities and long coherence times. Extreme cryogenic cooling not required. Ions are perfect and consistent. ***Cons***: Slow gate times/ operations and low connectivity between qubits. Lasers hard to align and scale. Ultra-high vacuum required. Ion charges may restrict scalability.	IONQ, AQT, QUANTINUUM, oxford ionics, Universal Quantum
Photonics	***Pros***: Extremely fast gate speeds and promising fidelities. No cryogenics or vacuums required. Small overall footprint. Can leverage existing CMOS fabs. ***Cons***: Noise from photon loss; each program requires its own chip. Photons don't naturally interact so 2Q gate challenges.	PsiQuantum, XANADU, QUIX QUANTUM, ORCA Computing
Neutral Atoms	***Pros***: Long coherence times. Atoms are perfect and consistent. Strong connectivity, including more than 2Q. External cryogenics not required. ***Cons***: Requires ultra-high vacuums. Laser scaling challenging.	ColdQuanta, QuEra Computing Inc., atom computing, PASQAL
Silicon Spin/Quantum Dots	***Pros***: Leverages existing semiconductor technology. Strong gate fidelities and speeds. ***Cons***: Requires cryogenics. Only a few entangled gates to-date with low coherence times. Interference/cross-talk challenges.	intel, Silicon Quantum Computing, diraq, QUANTUM MOTION, QUANTUM BRILLIANCE

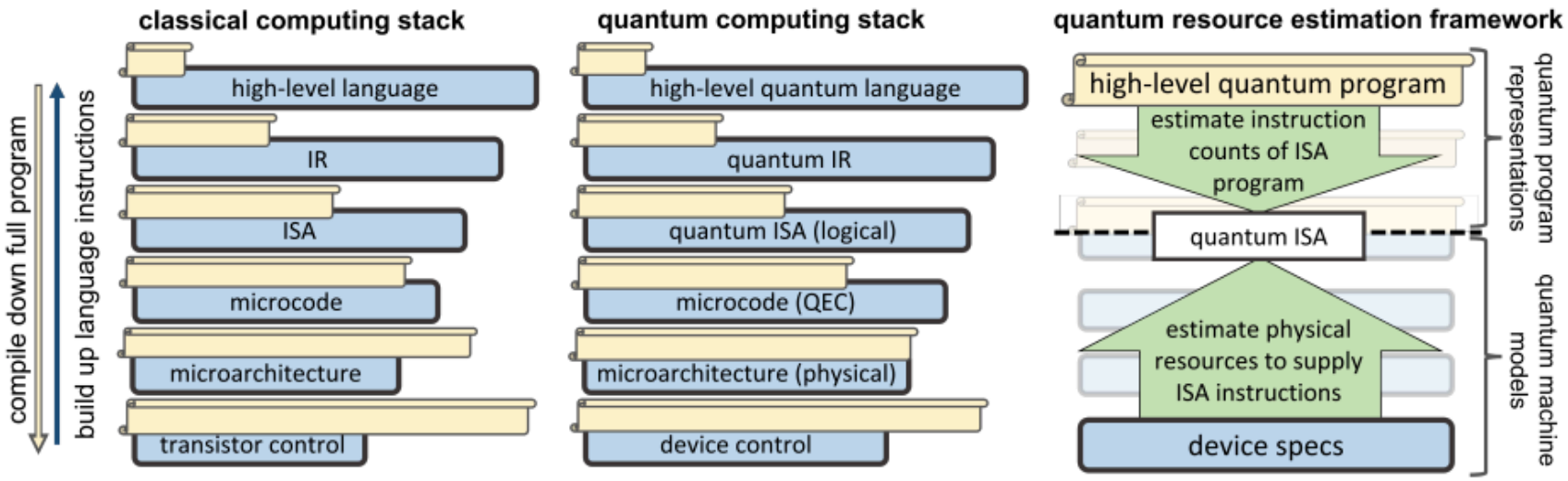
State of the art

Modality	Super-conducting	Trapped Ion	Photonic	Neutral Atom	Silicon Spin
# Qubits	127Q	32Q	20 Photons/ 216 Qumode	100Q	2Q
T2 Lifetime	Short 15μs-256μs	Long 0.2s-50s	Short 150μs	Long 0.2s-10s	Mixed 1μs-0.5s
2Q Gate Fidelity	High 99%-99.7%	High 98.5%-99.92%	Promising 98%	Promising 97.4%	Promising 90%-98%
Gate Speed	Fast 10ns-196ns	Mixed 1μs-3ms	Very Fast 1ns	Medium 1μs	Fast 0.8ns-80ns

Further considerations:

- How well can they be scaled up?
- How much energy do they consume?
- How large does the machine have to be?

Quantum Stack

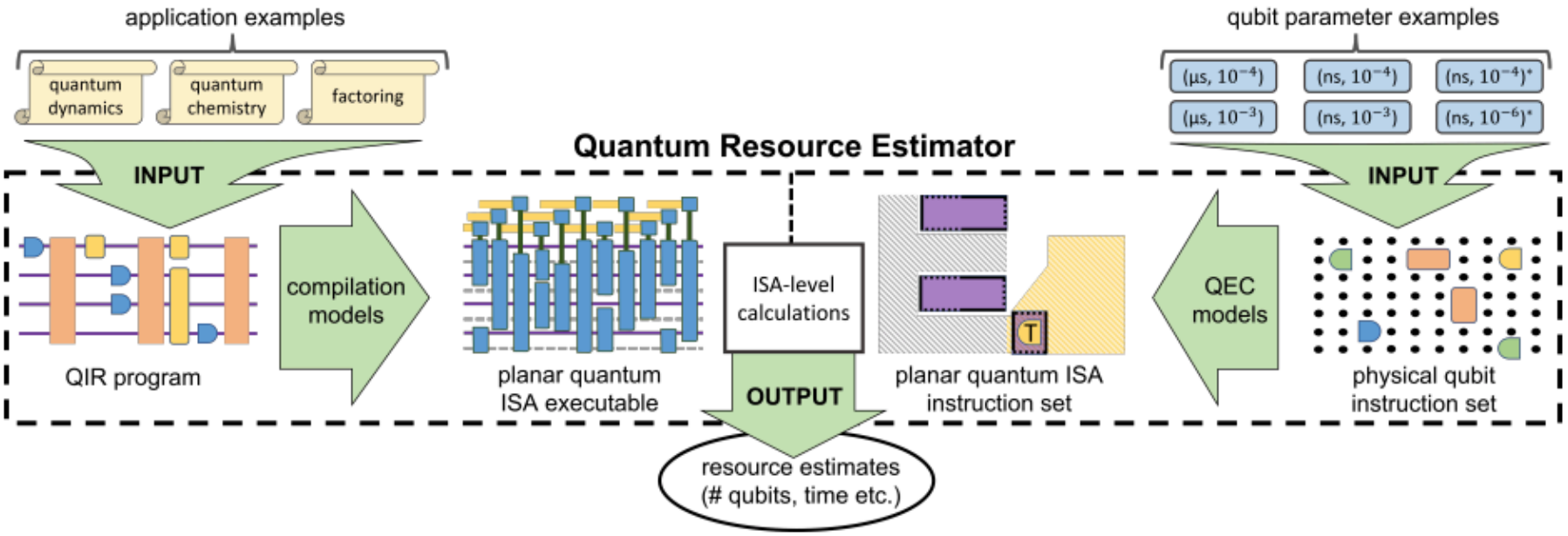


M. Beverland et. al, arXiv:2211.07629

Major differences:

- Multiple platforms, each with advantages and disadvantages
- Microarchitecture -> microcode very costly (because of quantum error correction)
- Applications are scarce, very specialised

Resource estimation



Resource estimation

Minimal applications

application	algorithm execution accuracy $1-\epsilon$	quantum executable parameters			quality requirements	
		Q	$C_{\min}$	M	max P	max P_T
quantum dynamics	0.999	230	$1.5 \cdot 10^5$	$2.4 \cdot 10^6$	$9.7 \cdot 10^{-12}$	$1.4 \cdot 10^{-10}$
quantum chemistry	0.99	2740	$4.1 \cdot 10^{11}$	$5.4 \cdot 10^{11}$	$3.0 \cdot 10^{-17}$	$6.1 \cdot 10^{-15}$
factoring	0.667	25481	$1.2 \cdot 10^{10}$	$1.5 \cdot 10^{10}$	$3.5 \cdot 10^{-16}$	$7.4 \cdot 10^{-12}$

 # qubits

 # sequential operations

 # required qubit quality

Cost of error correction

qubit parameter examples	QEC code selected	logical qubit parameters for $P(d) \leq 3.5 \cdot 10^{-16}$			T factory parameters for $P_T(\mathcal{D}) \leq 7.4 \cdot 10^{-12}$	
		distance d	# qubits $n(d)$	logical time step $\tau(d)$	# qubits $n(\mathcal{D})$	duration $\tau(\mathcal{D})$
(μs, 10^{-3}) qubit	surface	27	1458	16 ms	17640	163 ms
(μs, 10^{-4}) qubit	surface	13	338	7 ms	4840	85 ms
(ns, 10^{-3}) qubit	surface	27	1458	10 μs	33320	128 μs
(ns, 10^{-4}) qubit	surface	13	338	5 μs	5760	72 μs

Error correction trades off qubit quality for qubit number and slowdown (and a lot of infrastructure)

 # qubit overhead

 # clock speed (slowdown)

Quantum error correction

Classical error correction:

Example: repetition code

$00000 \rightarrow$ ‘0‘

$11111 \rightarrow$ ‘1‘

Decode errors by majority vote

$$\epsilon_L \sim e^{-\alpha(\epsilon)d}$$

$$d \propto N$$

`d` is the distance of the code.
Relation only holds below threshold

$\sim 50\%$

Quantum error correction:

Need to protect against `flip` and `phase` errors.

Flip $|0\rangle \leftrightarrow |1\rangle$

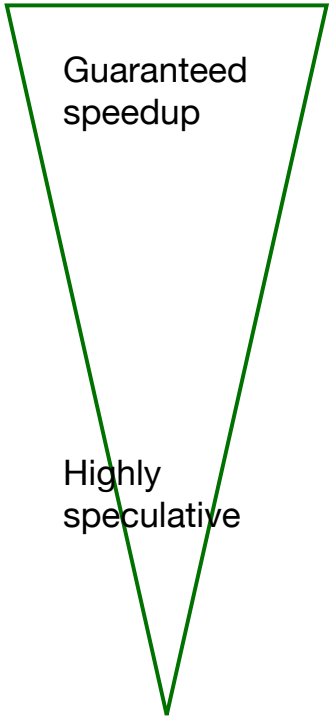
Phase $|1\rangle \leftrightarrow -|1\rangle$

Natural error rates are very high

Suppression of errors: $\epsilon_L \sim e^{-\alpha(\epsilon)d}$ $d \propto \sqrt{N}$

Threshold: $\sim 1\%$

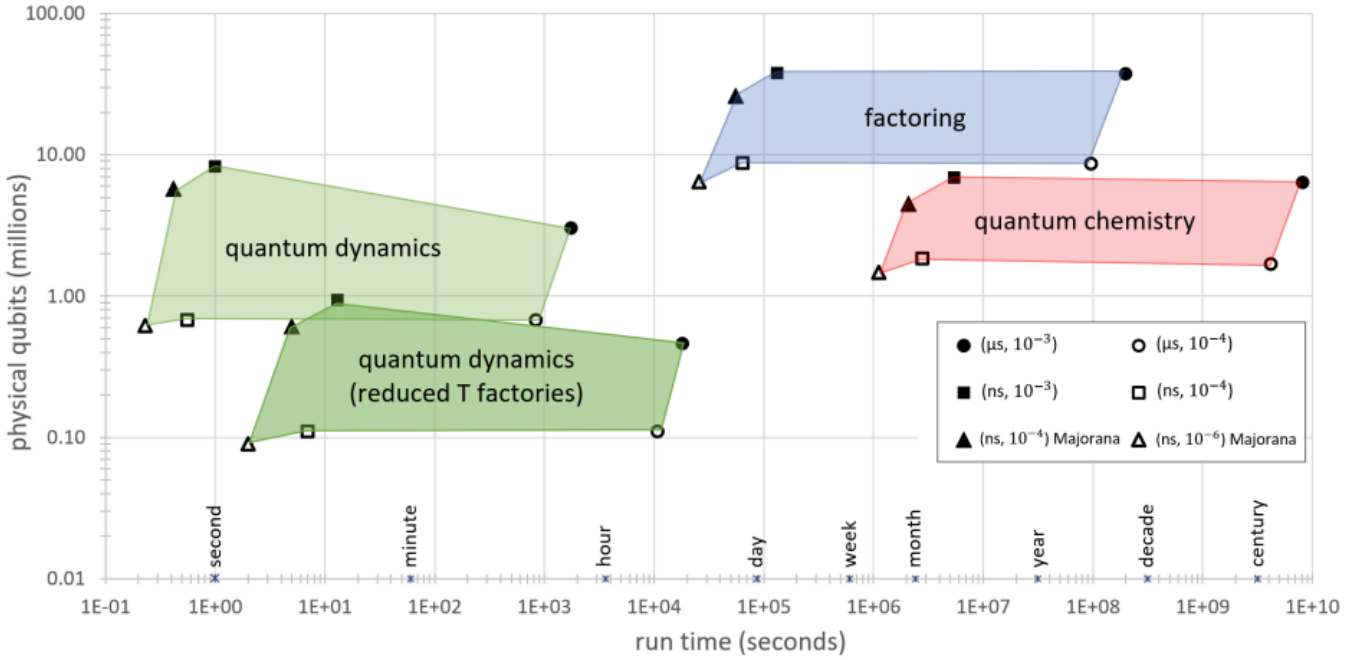
Applications



Factoring, discrete log	$\sim 10^7$qubits,	$\sim 10^{10}$gates
Materials simulation	$\sim 10^6$qubits,	$\sim 10^9$gates
Molecules simulations	$\sim 5 \times 10^6$qubits,	$\sim 10^{12}$gates
Everything else	$\sim 10^{4-}$ qubits,	$\sim 10^{13-}$ gates

Need to be extremely cautious with promises

Resource estimation

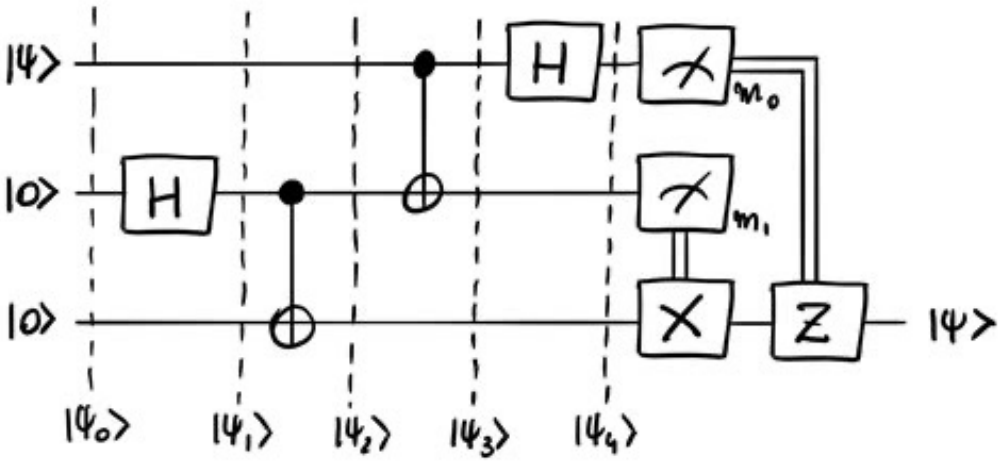


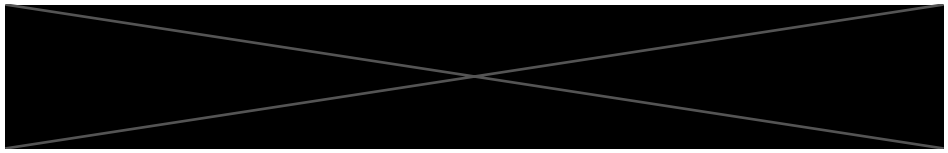
Three most promising applications of quantum computing

Boxes indicate minimal instances where we are confident that they are challenging for classical computers

Note, energy/cost is not represented here

Quantum circuit and running

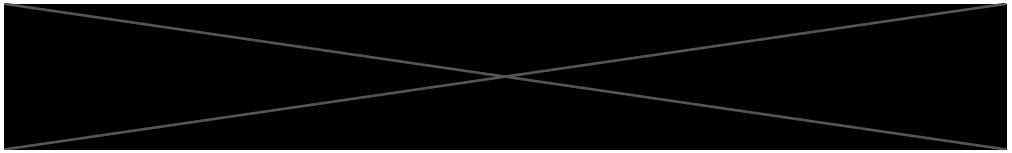




Operator	Gate(s)	Matrix
Pauli-X (X)	X	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
Pauli-Y (Y)	Y	$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$
Pauli-Z (Z)	Z	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
Hadamard (H)	H	$\frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
Phase (S, P)	S	$\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$
$\pi/8$ (T)	T	$\begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$
Controlled Not (CNOT, CX)		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
Controlled Z (CZ)	Z	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$
SWAP		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Toffoli (CCNOT, CCX, TOFF)		$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$

Quantum programming languages

- Python (Actual Quantum Programming Language)
- Qiskit (Open-source Programming Tool)
- Ocean™ (Quantum Computing Programming Suite)
- Q# (Quantum Computing Programming Algorithm)
- Cirq (Google AI Programming Language)
- Silq
- Qrisp
-



```
def solve[n:!N](bits:!𝔹^n){
  // prepare superposition between 0 and 1
  x:=H(0:𝔹);
  // prepare superposition between bits and 0
  qs := if x then bits else (0:int[n]) as 𝔹^n;
  // uncompute x
  forget(x=qs[0]); // valid because `bits[0]==1`
  return qs;
}

// EXAMPLE CALL

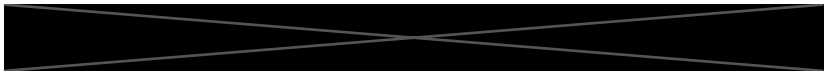
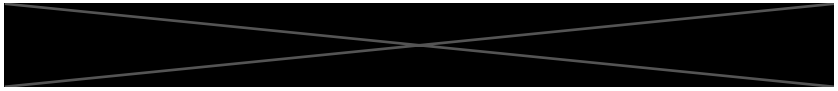
def main(){
  // example usage for bits=1, n=2
  x := 1:!int[2];
  y := x as !𝔹^2;
  return solve(y);
}
```

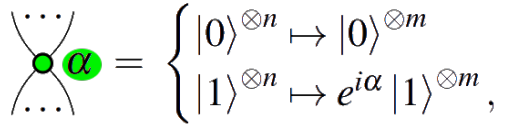
```
fun teleport (q1 : qubit<P>) : qubit<P> = (* pure type *)
  let q23 : (qubit & qubit)<P> = bell_pair () in
  let (q2 : qubit<M>, q3 : qubit<M>) = q23 in
  let (q1 : qubit<M>, q2 : qubit<M>) = CNOT (q1, q2) in
  let q1 : qubit<M> = H (q1) in
  let (q2 : qubit<M>, q3 : qubit<M>) = CNOT (q2, q3) in
  let (q1 : qubit<M>, q3 : qubit<M>) = CZ (q1, q3) in
  let q123 : ((qubit & qubit) & qubit)<M> = ((q1, q2), q3) in
  (* assert ((q1, q2), q3) is pure; check statically *)
  let q123 : ((qubit & qubit) & qubit)<P> = cast<P>(q123) in
  (* assert q3 is separable from (q1, q2); check dynamically *)
  let (q12 : (qubit & qubit)<P>, q3 : qubit<P>) = split<P>(q123) in
  let _ : bool * bool = measure (q12) in q3
```

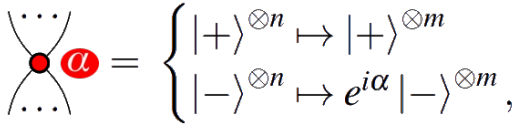
Quantum models and algebra!

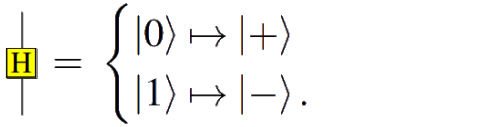
- Quantum Turing machines
  - Quantum Theory, the Church-Turing Principle and the Universal Quantum Computer*;* David Deutsch, 1985
- ZX calculus
  - Coecke, Duncan



- Programming the quantum future; Valiron, Ross, Selinger, Alexander, Smith
  - 
- Reversible quantum combinators: UΠ; Heunen, Kaarsgaard
  - 







How much hype is there in QC?

- Error correction (fault tolerance) is necessary
- Real quantum computers will still take a long time
- There are only few very promising applications at present
- Classical control hardware is also a big part of the game
- No algebraic model for QC

- There are many pitfalls, but the upsides seems to be real

- Related topics (not QC) so not to forget
  - Quantum key exchange
  - Post-Quantum encryption

Monitors: An Operating System Structuring Concept

C.A.R. Hoare
The Queen's University of Belfast

This paper develops Brinch-Hansen's concept of a monitor as a method of structuring an operating system. It introduces a form of synchronization, describes a possible method of implementation in terms of semaphores and gives a suitable proof rule. Illustrative examples include a single resource scheduler, a bounded buffer, an alarm clock, a buffer pool, a disk head optimizer, and a version of the problem of readers and writers.

Key Words and Phrases: monitors, operating systems, scheduling, mutual exclusion, synchronization, system implementation languages, structured multiprogramming
CR Categories: 4.31, 4.22

Copyright © 1974, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

This paper is based on an address delivered to IRIA, France, May 11, 1973. Author's address: Department of Computer Science, The Queen's University of Belfast, Belfast BT7 1NN, Northern Ireland.

1. Introduction

A primary aim of an operating system is to share a computer installation among many programs making unpredictable demands upon its resources. A primary task of its designer is therefore to construct resource allocation (or scheduling) algorithms for resources of various kinds (main store, drum store, magnetic tape handlers, consoles, etc.). In order to simplify his task, he should try to construct separate schedulers for each class of resource. Each scheduler will consist of a certain amount of local administrative data, together with some procedures and functions which are called by programs wishing to acquire and release resources. Such a collection of associated data and procedures is known as a *monitor*; and a suitable notation can be based on the *class* notation of SIMULA67 [6].

```
monitorname: monitor
  begin . . . declarations of data local to the monitor;
    procedure procname (. . . formal parameters . . .);
      begin . . . procedure body . . . end;
    . . . declarations of other procedures local to the monitor;
    . . . initialization of local data of the monitor . . .
  end;
```

Note that the procedure bodies may have local data, in the normal way.

In order to call a procedure of a monitor, it is necessary to give the name of the monitor as well as the name of the desired procedure, separating them by a dot:

```
monitorname.procname(. . . actual parameters . . .);
```

In an operating system it is sometimes desirable to declare several monitors with identical structure and behavior, for example to schedule two similar resources. In such cases, the declaration shown above will be preceded by the word **class**, and the separate monitors will be declared to belong to this class:

```
monitor 1, monitor 2: classname;
```

Thus the structure of a class of monitors is identical to that described for a data representation in [13], except for addition of the basic word *monitor*. Brinch-Hansen uses the word *shared* for the same purpose [3].

The procedures of a monitor are common to all running programs, in the sense that any program may at any time attempt to call such a procedure. However, it is essential that only one program at a time actually succeed in entering a monitor procedure, and any subsequent call must be held up until the previous call has been completed. Otherwise, if two procedure bodies were in simultaneous execution, the effects on the local variables of the monitor could be chaotic. The proce-

dures local to a monitor should not access any nonlocal variables other than those local to the same monitor, and these variables of the monitor should be inaccessible from outside the monitor. If these restrictions are imposed, it is possible to guarantee against certain of the more obscure forms of time-dependent coding error; and this guarantee could be underwritten by a visual scan of the text of the program, which could readily be automated in a compiler.

Any dynamic resource allocator will sometimes need to delay a program wishing to acquire a resource which is not currently available, and to resume that program after some other program has released the resource required. We therefore need: a "wait" operation, issued from inside a procedure of the monitor, which causes the calling program to be delayed; and a "signal" operation, also issued from inside a procedure of the same monitor, which causes exactly one of the waiting programs to be resumed immediately. If there are no waiting programs, the signal has no effect. In order to enable other programs to release resources during a wait, a wait operation must relinquish the exclusion which would otherwise prevent entry to the releasing procedure. However, we decree that a signal operation be followed immediately by resumption of a waiting program, without possibility of an intervening procedure call from yet a third program. It is only in this way that a waiting program has an absolute guarantee that it can acquire the resource just released by the signalling program without any danger that a third program will interpose a monitor entry and seize the resource instead.

In many cases, there may be more than one reason for waiting, and these need to be distinguished by both the waiting and the signalling operation. We therefore introduce a new type of "variable" known as a "condition"; and the writer of a monitor should declare a variable of type condition for each reason why a program might have to wait. Then the wait and signal operations should be preceded by the name of the relevant condition variable, separated from it by a dot:

```
condvariable.wait;
condvariable.signal;
```

Note that a condition "variable" is neither true nor false; indeed, it does not have any stored value accessible to the program. In practice, a condition variable will be represented by an (initially empty) queue of processes which are currently waiting on the condition; but this queue is invisible both to waiters and signallers. This design of the condition variable has been deliberately kept as primitive and rudimentary as possible, so that it may be implemented efficiently and used flexibly to achieve a wide variety of effects. There is a great temptation to introduce a more complex synchronization primitive, which may be easier to use for many purposes. We shall resist this temptation for a while.

As the simplest example of a monitor, we will design a scheduling algorithm for a single resource, which is dynamically acquired and released by an unknown number of customer processes by calls on procedures

```
procedure acquire;
procedure release;
```

A variable[1]

```
busy:Boolean
```

determines whether or not the resource is in use. If an attempt is made to acquire the resource when it is busy, the attempting program must be delayed by waiting on a variable

```
nonbusy:condition
```

which is signalled by the next subsequent release. The initial value of busy is false. These design decisions lead to the following code for the monitor:

```
single resource:monitor
begin busy:Boolean;
    nonbusy:condition;
  procedure acquire;
    begin if busy then nonbusy.wait;
          busy := true
    end;
  procedure release;
    begin busy := false;
        nonbusy.signal
    end;
  busy := false; comment initial value;
end single resource
```

Notes

1. In designing a monitor, it seems natural to design the procedure headings, the data, the conditions, and the procedure bodies, in that order. All subsequent examples will be designed in this way.
2. The acquire procedure does not have to retest that busy has gone false when it resumes after its wait, since the release procedure has guaranteed that this is so; and as mentioned before, no other program can intervene between the signal and the continuation of exactly one waiting program.
3. If more than one program is waiting on a condition, we postulate that the signal operation will reactivate the longest waiting program. This gives a simple neutral queuing discipline which ensures that every waiting program will eventually get its turn.
4. The single resource monitor simulates a Boolean semaphore [7] with *acquire* and *release* used for *P* and *V* respectively. This is a simple proof that the monitor/condition concepts are not in principle less powerful than semaphores, and that they can be used for all the same purposes.

[1] As in PASCAL [15], a variable declaration is of the form: ⟨*variable identifier*⟩:⟨*type*⟩;

2. Interpretation

Having proved that semaphores can be implemented by a monitor, the next task is to prove that monitors can be implemented by semaphores.

Obviously, we shall require for each monitor a Boolean semaphore "*mutex*" to ensure that the bodies of the local procedures exclude each other. The semaphore is initialized to 1; a *P*(*mutex*) must be executed on entry to each local procedure, and a *V*(*mutex*) must usually be executed on exit from it.

When a process signals a condition on which another process is waiting, the signalling process must wait until the resumed process permits it to proceed. We therefore introduce for each monitor a second semaphore "*urgent*" (initialized to 0), on which signalling processes suspend themselves by the operation *P*(*urgent*). Before releasing exclusion, each process must test whether any other process is waiting on *urgent*, and if so, must release it instead by a *V*(*urgent*) instruction. We therefore need to count the number of processes waiting on *urgent*, in an integer "*urgentcount*" (initially zero). Thus each exit from a procedure of a monitor should be coded:

if *urgentcount* > 0 **then** *V*(*urgent*) **else** *V*(*mutex*)

Finally, for each condition local to the monitor, we introduce a semaphore "*condsem*" (initialized to 0), on which a process desiring to wait suspends itself by a *P*(*condsem*) operation. Since a process signalling this condition needs to know whether anybody is waiting, we also need a count of the number of waiting processes held in an integer variable "*condcount*" (initially 0). The operation "*cond.wait*" may now be implemented as follows (recall that a waiting program must release exclusion before suspending itself):

condcount := *condcount* + 1;
if *urgentcount* > 0 **then** *V*(*urgent*) **else** *V*(*mutex*);
P(*condsem*);
comment This will always wait;
condcount := *condcount* − 1

The signal operation may be coded:

urgentcount := *urgentcount* + 1;
if *condcount* > 0 **then** {*V*(*condsem*); *P*(*urgent*)};
urgentcount := *urgentcount* − 1

In this implementation, possession of the monitor is regarded as a privilege which is explicitly passed from one process to another. Only when no one further wants the privilege is *mutex* finally released.

This solution is not intended to correspond to recommended "style" in the use of semaphores. The concept of a condition-variable is intended as a substitute for semaphores, and has its own style of usage, in the same way that while-loops or coroutines are intended as a substitute for jumps.

In many cases, the generality of this solution is unnecessary, and a significant improvement in efficiency is possible.

1. When a procedure body in a monitor contains no **wait** or **signal**, **exit** from the body can be coded by a simple *V*(*mutex*), since *urgentcount* cannot have changed during the execution of the body.
2. If a *cond.signal* is the last operation of a procedure body, it can be combined with monitor exit as follows:

if *condcount* > 0 **then** *V*(*condsem*)
else if *urgentcount* > 0 **then** *V*(*urgent*)
else *V*(*mutex*)

3. If there is no other wait or signal in the procedure body, the second line shown above can also be omitted.
4. If *every* signal occurs as the last operation of its procedure body, the variables *urgentcount* and *urgent* can be omitted, together with all operations upon them. This is such a simplification that O-J. Dahl suggests that signals should always be the last operation of a monitor procedure; in fact, this restriction is a very natural one, which has been unwittingly observed in all examples of this paper.

Significant improvements in efficiency may also be obtained by avoiding the use of semaphores, and by implementing conditions directly in hardware, or at the lowest and most uninterruptible level of software (e.g. supervisor mode). In this case, the following optimizations are possible.

1. *urgentcount* and *condcount* can be abolished, since the fact that someone is waiting can be established by examining the representation of the semaphore, which cannot change surreptitiously within noninterruptible mode.
2. Many monitors are very short and contain no calls to other monitors. Such monitors can be executed wholly in noninterruptible mode, using, as it were, the common exclusion mechanism provided by hardware. This will often involve *less* time in noninterruptible mode than the establishment of separate exclusion for each monitor.

I am grateful to J. Bezivin, J. Horning, and R.M. McKeag for assisting in the discovery of this algorithm.

3. Proof Rules

The analogy between a monitor and a data representation has been noted in the introduction. The mutual exclusion on the code of a monitor ensures that procedure calls follow each other in time, just as they do in sequential programming; and the same restrictions are placed on access to nonlocal data. These are the reasons why the same proof rules can be applied to monitors as to data representations.

As with a data representation, the programmer may associate an invariant $\mathcal{I}$ with the local data of a monitor, to describe some condition which will be true of this data before and after every procedure call. $\mathcal{I}$ must also be made true after initialization of the data, and before *every* wait instruction; otherwise the next following procedure call will not find the local data in a state which it expects.

With each condition variable b the programmer may associate an assertion B which describes the condition under which a program waiting on b wishes to be resumed. Since other programs may invoke a monitor procedure during a wait, a waiting program must ensure that the invariant $\mathcal{I}$ for the monitor is true beforehand. This gives the proof rule for waits:

$\mathcal{I}\ \{b.wait\}\ \mathcal{I}\&B$

Since a signal can cause immediate resumption of a waiting program, the conditions $\mathcal{I}\&B$ which are expected by that program must be made true before the signal; and since B may be made false again by the resumed program, only $\mathcal{I}$ may be assumed true afterwards. Thus the proof rule for a signal is:

$\mathcal{I}\&B\{b.signal\}\mathcal{I}$

This exhibits a pleasing symmetry with the rule for waiting.

The introduction of condition variables makes it possible to write monitors subject to the risk of deadly embrace [7]. It is the responsibility of the programmer to avoid this risk, together with other scheduling disasters (thrashing, indefinitely repeated overtaking, etc. [11]). Assertion-oriented proof methods cannot prove absence of such risks; perhaps it is better to use less formal methods for such proofs.

Finally, in many cases an operating system monitor constructs some "virtual" resource which is used in place of actual resources by its "customer" programs. This virtual resource is an abstraction from the set of local variables of the monitor. The program prover should therefore define this abstraction in terms of its concrete representation, and then express the intended effect of each of the procedure bodies in terms of the abstraction. This proof method is described in detail in [13].

4. Example: Bounded Buffer

A bounded buffer is a concrete representation of the abstract idea of a sequence of portions. The sequence is accessible to two programs running in parallel: the first of these (the producer) updates the sequence by appending a new portion x at the end; and the second (the consumer) updates it by removing the first portion. The initial value of the sequence is empty. We thus require two operations:

(1) *append*(x:*portion*);

which should be equivalent to the abstract operation

sequence := *sequence* ⌢ $\langle x \rangle$;

where $\langle x \rangle$ is the sequence whose only item is x and ⌢ denotes concatenation of two sequences.

(2) *remove*(**result** x:*portion*);

which should be equivalent to the abstract operations

x := *first*(*sequence*); *sequence* := *rest*(*sequence*);

where *first* selects the first item of a sequence and *rest* denotes the sequence with its first item removed. Obviously, if the sequence is empty, *first* is undefined; and in this case we want to ensure that the consumer waits until the producer has made the sequence nonempty.

We shall assume that the amount of time taken to produce a portion or consume it is large in comparison with the time taken to append or remove it from the sequence. We may therefore be justified in making a design in which producer and consumer can both update the sequence, but not simultaneously.

The sequence is represented by an array:

buffer :**array** 0..$N-1$ **of** *portion*;

and two variables:

(1) *lastpointer*:0..$N-1$;

which points to the buffer position into which the next append operation will put a new item, and

(2) *count*:0..N;

which always holds the length of the sequence (initially 0).

We define the function

$$seq\ (b,l,c) =_{df} \textbf{if } c = 0 \textbf{ then } empty \textbf{ else } seq(b,l\ominus 1,c-1)\ ⌢\langle b[l\ominus 1]\rangle$$

where the circled operations are taken modulo N. Note that if $c \neq 0$,

$$first(seq(b,l,c)) = b[l\ominus c]$$

and

$$rest(seq(b,l,c)) = seq(b,l,c-1)$$

The definition of the abstract sequence in terms of its concrete representation may now be given:

$$sequence =_{df} seq(buffer, lastpointer, count)$$

Less formally, this may be written

$$sequence =_{df} \langle buffer[lastpointer\ominus count], buffer[lastpointer\ominus count\oplus 1], \ldots, buffer[lastpointer\ominus 1]\rangle$$

Another way of conveying this information would be by an example and a picture, which would be even less formal.

The invariant for the monitor is:

$$0 \leq count \leq N\ \&\ 0 \leq lastpointer \leq N-1$$

There are two reasons for waiting, which must be represented by condition variables:

nonempty:*condition*;

means that the count is greater than 0, and

nonfull:*condition*;

means that the count is less than N.

With this constructive approach to the design [8], it is relatively easy to code the monitor without error.

```
bounded buffer:monitor
  begin buffer:array 0..N − 1 of portion;
        lastpointer:0..N − 1;
        count:0..N;
        nonempty,nonfull:condition;
    procedure append(x:portion);
      begin if count = N then nonfull.wait;
            note 0 ≤ count < N;
            buffer[lastpointer] := x;
            lastpointer := lastpointer ⊕ 1;
            count := count+1;
            nonempty.signal
      end append;
    procedure remove(result x:portion);
      begin if count = 0 then nonempty.wait;
            note 0 < count ≤ N;
            x := buffer[lastpointer⊖count];
            nonfull.signal
      end remove;
    count := 0; lastpointer := 0;
  end bounded buffer;
```

A formal proof of the correctness of this monitor with respect to the stated abstraction and invariant can be given if desired by techniques described in [13]. However, these techniques seem not capable of dealing with subsequent examples of this paper.

Single-buffered input and output may be regarded as a special case of the bounded buffer with $N = 1$. In this case, the array can be replaced by a single variable, the *lastpointer* is redundant, and we get:

```
iostream:monitor
begin buffer:portion;
      count:0..1;
      nonempty,nonfull:condition;
  procedure append(x:portion);
    begin if count = 1 then nonfull.wait;
      buffer := x;
      count := 1;
      nonempty.signal
    end append;
  procedure remove(result x:portion);
    begin if count = 0 then nonempty.wait;
      x := buffer;
      count := 0;
      nonfull.signal
    end remove;
  count := 0;
end iostream;
```

If physical output is carried out by a separate special purpose channel, then the interrupt from the channel should simulate a call of *iostream.remove*(x); and similarly for physical input, simulating a call of *iostream.append*(x).

5. Scheduled Waits

Up to this point, we have assumed that when more than one program is waiting for the same condition, a signal will cause the longest waiting program to be resumed. This is a good simple scheduling strategy, which precludes indefinite overtaking of a waiting process.

However, in the design of an operating system, there are many cases when such simple scheduling on the basis of first-come-first-served is not adequate. In order to give a closer control over scheduling strategy, we introduce a further feature of a conditional wait, which makes it possible to specify as a parameter of the wait some indication of the priority of the waiting program, e.g.:

```
busy . wait (p);
```

When the condition is signalled, it is the program that specified the lowest value of p that is resumed. In using this facility, the designer of a monitor must take care to avoid the risk of indefinite overtaking; and often it is advisable to make priority a nondecreasing function of the time at which the wait commences.

This introduction of a "scheduled wait" concedes to the temptation to make the condition concept more elaborate. The main justifications are:

1. It has no effect whatsoever on the *logic* of a program, or on the formal proof rules. Any program which works without a scheduled wait will work with it, but possibly with better timing characteristics.
2. The automatic ordering of the queue of waiting processes is a simple fast scheduling technique, except when the queue is exceptionally long—and when it is, central processor time is not the major bottleneck.
3. The maximum amount of storage required is one word per process. Without such a built-in scheduling method, each monitor may have to allocate storage proportional to the number of its customers; the alternative of dynamic storage allocation in small chunks is unattractive at the low level of an operating system where monitors are found.

I shall yield to one further temptation, to introduce a Boolean function of conditions:

```
condname.queue
```

which yields the value true if anyone is waiting on *condname* and false otherwise. This can obviously be easily implemented by a couple of instructions, and affords valuable information which could otherwise be obtained only at the expense of extra storage, time, and trouble.

A trivially simple example is an *alarmclock* monitor, which enables a calling program to delay itself for a stated number n of time-units, or "*ticks*". There are two entries:

```
procedure wakeme (n:integer);
procedure tick;
```

The second of these is invoked by hardware (e.g. an interrupt) at regular intervals, say ten times per second. Local variables are

```
now:integer;
```

which records the current time (initially zero) and

```
wakeup:condition;
```

on which sleeping programs wait. But the *alarmsetting* at which these programs will be aroused is known at the time when they start the wait; and this can be used to determine the correct sequence of waking up.

```
alarmclock:monitor
begin now:integer;
    wakeup:condition;
  procedure wakeme(n:integer);
    begin alarmsetting:integer;
      alarmsetting := now + n;
      while now < alarmsetting do wakeup.wait(alarmsetting);
      wakeup.signal;
      comment In case the next process is due to wake up at the
      same time;
    end;
  procedure tick;
    begin now := now + 1;
      wakeup.signal
    end;
  now := 0
end alarmclock
```

In the program given above, the next candidate for wakening is actually woken at every tick of the clock. This will not matter if the frequency of ticking is low enough, and the overhead of an accepted signal is not too high.

I am grateful to A. Ballard and J. Horning for posing this problem.

6. Further Examples

In proposing a new feature for a high level language it is very difficult to make a convincing case that the feature will be both easy to use efficiently and easy to implement efficiently. Quality of implementation can be proved by a single good example, but ease and efficiency of use require a great number of realistic examples; otherwise it can appear that the new feature has been specially designed to suit the examples, or vice versa. This section contains a number of additional examples of solutions of familiar problems. Further examples may be found in [14].

6.1 Buffer Allocation

The bounded buffer described in Section 4 was designed to be suitable only for sequences with small portions, for example, message queues. If the buffers contain high volume information (for example, files for pseudo offline input and output), the bounded buffer may still be used to store the *addresses* of the buffers which are being used to hold the information. In this way, the producer can be filling one buffer while the consumer is emptying another buffer of the same sequence. But this requires an allocator for dynamic acquisition and relinquishment of *buffer addresses*. These may be declared as a type

```
type bufferaddress = 1..B;
```

where B is the number of buffers available for allocation.

The buffer allocator has two entries:

```
procedure acquire (result b:bufferaddress);
```

which delivers a free *buffer address* b; and

```
procedure release(b:bufferaddress);
```

which returns a *bufferaddress* when it is no longer required. In order to keep a record of free buffer addresses the monitor will need:

```
freepool:powerset bufferaddress;
```

which uses the *PASCAL* powerset facility to define a variable whose values range over all sets of *buffer addresses*, from the empty set to the set containing all *buffer addresses*. It should be implemented as a *bitmap* of B consecutive bits, where the ith bit is 1 if and only if i is in the set. There is only one condition variable needed:

```
nonempty:condition
```

which means that *freepool* $\neq$ *empty*. The code for the allocator is:

```
buffer allocator:monitor
begin freepool:powerset bufferaddress;
    nonempty:condition;
  procedure acquire (result b:bufferaddress);
    begin if freepool = empty then nonempty.wait;
      b := first(freepool);
      comment Any one would do;
      freepool := freepool - {b};
      comment Set subtraction;
    end acquire;
  procedure release(b:bufferaddress);
    begin freepool := freepool - {b};
      nonempty.signal
    end release;
  freepool := all buffer addresses
end buffer allocator
```

The action of a producer and consumer may be summarized:

```
producer: begin b:bufferaddress; . . .
            while not finished do
              begin bufferallocator.acquire(b);
                . . . fill buffer b . . .;
                bounded buffer.append(b)
              end; . . .
          end producer;
consumer: begin b:bufferaddress; . . .
            while not finished do
              begin bounded buffer.remove(b);
                . . . empty buffer b . . .;
                buffer allocator.release(b)
              end; . . .
          end consumer;
```

This buffer allocator would appear to be usable to share the buffers among several streams, each with its own producer and its own consumer, and its own instance of a bounded buffer monitor. Unfortunately,

when the streams operate at widely varying speeds, and when the freepool is empty, the scheduling algorithm can exhibit persistent undesirable behavior. If two producers are competing for each buffer as it becomes free, a first-come-first-served discipline of allocation will ensure (apparently fairly) that each gets alternate buffers; and they will consequently begin to produce at equal speeds. But if one consumer is a 1000 lines/min printer and the other is a 10 lines/min teletype, the faster consumer will be eventually reduced to the speed of the slower, since it cannot forever go faster than its producer. At this stage nearly all buffers will belong to the slower stream, so the situation could take a long time to clear.

A solution to this is to use a scheduled wait, to ensure that in heavy load conditions the available buffers will be shared reasonably fairly between the streams that are competing for them. Of course, inactive streams need not be considered, and streams for which the consumer is currently faster than the producer will never ask for more than two buffers anyway. In order to achieve fairness in allocation, it is sufficient to allocate a newly freed buffer to that one among the competing producers whose stream currently owns fewest buffers. Thus the system will seek a point as far away from the undesirable extreme as possible.

For this reason, the entries to the allocator should indicate for what stream the buffer is to be (or has been) used, and the allocator must keep a count of the current allocation to each stream in an array:

```
count:array stream of integer;
```

The new version of the allocator is:

```
bufferallocator:monitor
  begin freepool:powerset bufferaddress;
      nonempty:condition
      count:array stream of integer;
    procedure acquire(result b:bufferaddress; s:stream);
      begin if freepool = empty then nonempty.wait(count[s]);
        count[s] := count[s] + 1;
        b := first(freepool);
        freepool := freepool - {b}
      end acquire;
    procedure release(b:bufferaddress; s:stream)
      begin count[s] := count[s] - 1;
        freepool := freepool - {b};
        nonempty·signal
      end;
    freepool := all buffer addresses;
    for s:stream do count[s] := 0
  end bufferallocator
```

Of course, if a consumer stops altogether, perhaps owing to mechanical failure, the producer must also be halted before it has acquired too many buffers, even if no one else currently wants them. This can perhaps be most easily accomplished by appropriate fixing of the size of the bounded buffer for that stream and/or by ensuring that at least two buffers are reserved for each stream, even when inactive. It is an interesting comment on dynamic resource allocation that, as soon as resources are heavily loaded, the system must be designed to fall back toward a more static regime.

I am grateful to E.W. Dijkstra for pointing out this problem and its solution [10].

6.2 Disk Head Scheduler

On a moving head disk, the time taken to move the heads increases monotonically with the distance traveled. If several programs wish to move the heads, the average waiting time can be reduced by selecting, first, the program which wishes to move them the shortest distance. But unfortunately this policy is subject to an instability, since a program wishing to access a cylinder at one edge of the disk can be indefinitely overtaken by programs operating at the other edge or the middle.

A solution to this is to minimize the frequency of change of direction of movement of the heads. At any time, the heads are kept moving in a given direction, and they service the program requesting the nearest cylinder in that direction. If there is no such request, the direction changes, and the heads make another sweep across the surface of the disk. This may be called the "elevator" algorithm, since it simulates the behavior of a lift in a multi-storey building.

There are two entries to a disk head scheduler:

(1) *request*(*dest*:*cylinder*);

where

```
type cylinder = 0..cylmax;
```

which is entered by a program just *before* issuing the instruction to move the heads to cylinder *dest*.

(2) *release*;

which is entered by a program when it has made all the transfers it needs on the current cylinder.

The local data of the monitor must include a record of the current headposition, *headpos*, the current direction of *sweep*, and whether the disk is *busy*:

```
headpos:cylinder;
direction:(up, down);
busy:Boolean
```

We need two conditions, one for requests waiting for an *upsweep* and the other for requests waiting for a *downsweep*:

```
upsweep, downsweep:condition
```

```
dischead:monitor
begin headpos:cylinder;
    direction:(up, down);
    busy:Boolean;
    upsweep,downsweep:condition;
  procedure request(dest:cylinder);
    begin if busy then
      {if headpos < dest ∨ headpos = dest & direction = up
        then upsweep·wait(dest)
        else downsweep.wait(cylmax-dest)};
      busy := true; headpos := dest
    end request;
```

```
  procedure release;
    begin busy := false;
      if direction = up then
        {if upsweep.queue then upsweep.signal
                          else {direction := down;
                                downsweep.signal}}
      else if downsweep.queue then downsweep.signal
                              else {direction := up;
                                    upsweep.signal}
    end release;
    headpos := 0; direction := up; busy := false
end dischead;
```

6.3 Readers and Writers

As a more significant example, we take a problem which arises in on-line real-time applications such as airspace control. Suppose that each aircraft is represented by a record, and that this record is kept up to date by a number of "writer" processes and accessed by a number of "reader" processes. Any number of "reader" processes may simultaneously access the same record, but obviously any process which is updating (writing) the individual components of the record must have exclusive access to it, or chaos will ensue. Thus we need a class of monitors; an instance of this class local to *each* individual aircraft record will enforce the required discipline for that record. If there are many aircraft, there is a strong motivation for minimizing local data of the monitor; and if each read or write operation is brief, we should also minimize the time taken by each monitor entry.

When many readers are interested in a single aircraft record, there is a danger that a writer will be indefinitely prevented from keeping that record up to date. We therefore decide that a new reader should not be permitted to start if there is a writer waiting. Similarly, to avoid the danger of indefinite exclusion of readers, all readers waiting at the end of a write should have priority over the next writer. Note that this is a very different scheduling rule from that propounded in [4], and does not seem to require such subtlety in implementation. Nevertheless, it may be more suited to this kind of application, where it is better to read stale information than to wait indefinitely!

The monitor obviously requires four local procedures:

startread	entered by reader who wishes to read.
endread	entered by reader who has finished reading.
startwrite	entered by writer who wishes to write.
endwrite	entered by writer who has finished writing.

We need to keep a count of the number of users who are reading, so that the last reader to finish will know this fact:

```
readercount:integer
```

We also need a *Boolean* to indicate that someone is actually writing:

```
busy:Boolean;
```

We introduce separate conditions for readers and writers to wait on:

```
OKtoread, OKtowrite:condition;
```

The following annotation is relevant:

```
OKtoread ≡ ¬ busy
OKtowrite ≡ ¬ busy & readercount = 0
invariant: busy ⟹ readercount = 0
```

```
class readers and writers:monitor
  begin readercount:integer;
      busy:Boolean;
      OKtoread, OKtowrite:condition;
    procedure startread;
      begin if busy ∨ OKtowrite.queue then OKtoread.wait;
        readercount := readercount + 1;
        OKtoread.signal;
        comment Once one reader can start, they all can;
      end startread;
    procedure endread;
      begin readercount := readercount − 1;
        if readercount = 0 then OKtowrite.signal
      end endread;
    procedure startwrite;
      begin
        if readercount ≠ 0 ∨ busy then OKtowrite.wait
        busy := true
      end startwrite;
    procedure endwrite;
      begin busy := false;
        if OKtoread.queue then OKtoread.signal
                          else OKtowrite.signal
      end endwrite;
    readercount := 0;
    busy := false;
  end readers and writers;
```

I am grateful to Dave Gorman for assisting in the discovery of this solution.

7. Conclusion

This paper suggests that an appropriate structure for a module of an operating system, which schedules resources for parallel user processes, is very similar to that of a data representation used by a sequential program. However, in the case of monitors, the bodies of the procedures must be protected against re-entrance by being implemented as critical regions. The textual grouping of critical regions together with the data which they update seems much superior to critical regions scattered through the user program, as described in [7, 12]. It also corresponds to the traditional practice of the writers of operating system supervisors. It can be recommended without reservation.

However, it is much more difficult to be confident about the condition concept as a synchronizing primitive. The synchronizing facility which is easiest to use is probably the conditional *wait* [2, 12]:

```
wait(B);
```

where B is a general Boolean expression (it causes the given process to wait until B becomes true); but this may be too inefficient for general use in operating systems,

because its implementation requires re-evaluation of the expression B after every exit from a procedure of the monitor. The condition variable gives the programmer better control over efficiency and over scheduling; it was designed to be very primitive, and to have a simple proof rule. But perhaps some other compromise between convenience and efficiency might be better. The question whether the signal should always be the last operation of a monitor procedure is still open. These problems will be studied in the design and implementation of a pilot project operating system, currently enjoying the support of the Science Research Council of Great Britain.

Another question which will be studied will be that of the disjointness of monitors: Is it possible to design a separate isolated monitor for each kind of resource, so that it will make sensible scheduling decisions for that resource, using only the minimal information about the utilization of that resource, and using no information about the utilization of any resource administered by other monitors? In principle, it would seem that, when more knowledge of the status of the entire system is available, it should be easier to take decisions nearer to optimality. Furthermore, in principle, independent scheduling of different kinds of resource can lead to deadly embrace. These considerations would lead to the design of a traditional "monolithic" monitor, maintaining large system tables, all of which can be accessed and updated by any of the procedures of the monitor.

There is no a priori reason why the attempt to split the functions of an operating system into a number of isolated disjoint monitors should succeed. It can be made to succeed only by discovering and implementing good scheduling algorithms in each monitor. In order to avoid undesirable interactions between the separate scheduling algorithms, it appears necessary to observe the following principles:

1. Never seek to make an optimal decision; merely seek to avoid persistently pessimal decisions.
2. Do not seek to present the user with a virtual machine which is better than the actual hardware; merely seek to pass on the speed, size, and flat unopiniated structure of a simple hardware design.
3. Use preemptive techniques in preference to non-preemptive ones where possible.
4. Use "grain of time" [9] methods to secure independence of scheduling strategies.
5. Keep a low variance (as well as a low mean) on waiting times.
6. Avoid fixed priorities; instead, try to ensure that every program in the system makes reasonably steady progress. In particular, avoid indefinite overtaking.
7. Ensure that when demand for resources outstrips the supply (i.e. in overload conditions), the behavior of the scheduler is satisfactory (i.e. thrashing is avoided).
8. Make rules for the correct and sensible use of monitor calls, and assume that user programs will obey them. Any checking which is necessary should be done not by a central shared monitor, but rather by an algorithm (called "user envelope") which is local to each process executing a user program. This algorithm should be implemented at least partially in the hardware (e.g. base and range registers, address translation mechanisms, capabilities, etc.).

It is the possibility of constructing separate monitors for different purposes, and of separating the scheduling decisions embodied in monitors from the checking embodied in user envelopes, that may justify a hope that monitors are an appropriate concept for the structuring of an operating system.

Acknowledgments. The development of the monitor concept is due to frequent discussions and communications with E.W. Dijkstra and P. Brinch-Hansen. A monitor corresponds to the "secretary" described in [9], and is also described in [1, 3].

Acknowledgment is also due to the support of IFIP WG.2.3., which provides a meeting place at which these and many other ideas have been germinated, fostered, and tested.

Received February 1973; revised April 1974

References

1. Brinch-Hansen, P. Structured multiprogramming. *Comm. ACM 15*, 7 (July 1972), 574–577.
2. Brinch-Hansen, P. "A comparison of two synchronizing concepts," *Acta Information 1* (1972), 190–199.
3. Brinch-Hansen, P. *Operating System Principles*. Prentice-Hall, Englewood Cliffs, N.J., 1973.
4. Courtois, P. J., Heymans, F., Parnas, D.L. Concurrent control with readers and writers. *Comm. ACM 14*, 10 (Oct. 1971), 667–668.
5. Courtois, P.J., Heymans, F., Parnas, D.L. Comments on [2]. *Acta Informatica 1* (1972), 375–376.
6. Dahl, O.J. Hierarchical program structures. In *Structured Programming*, Academic Press, New York, 1972.
7. Dijkstra, E.W. Cooperating Sequential Processes. In *Programming Languages* (Ed. F. Genuys), Academic Press, New York, 1968.
8. Dijkstra, E.W. A constructive approach to the problem of program correctness. *BIT 8* (1968), 174–186.
9. Dijkstra, E.W. Hierarchical ordering of sequential processes. In *Operating Systems Techniques*, Academic Press, New York, 1972.
10. Dijkstra, E.W. Information streams sharing a finite buffer. *Information Processing Letters 1*, 5 (Oct. 1972), 179–180.
11. Dijkstra, E.W. A class of allocation strategies inducing bounded delays only. Proc AFIPS 1972 SJCC, Vol. 40, AFIPS Press, Montvale, N.J., pp. 933–936.
12. Hoare, C.A.R. Towards a theory of parallel programming. In *Operating Systems Techniques*, Academic Press, New York, 1972.
13. Hoare, C.A.R. Proof of correctness of data representations. *Acta Informatica* 1 (1972), 271–281.
14. Hoare, C.A.R. A structured paging system. *Computer J. 16*, 3 (1973), 209–215.
15. Wirth, N. The programming language PASCAL. *Acta Informatica 1*, 1 (1971), 35–63.

Cramming more components onto integrated circuits

With unit cost falling as the number of components per circuit rises, by 1975 economics may dictate squeezing as many as 65,000 components on a single silicon chip

By Gordon E. Moore

Director, Research and Development Laboratories, Fairchild Semiconductor division of Fairchild Camera and Instrument Corp.

The future of integrated electronics is the future of electronics itself. The advantages of integration will bring about a proliferation of electronics, pushing this science into many new areas.

Integrated circuits will lead to such wonders as home computers—or at least terminals connected to a central computer—automatic controls for automobiles, and personal portable communications equipment. The electronic wristwatch needs only a display to be feasible today.

But the biggest potential lies in the production of large systems. In telephone communications, integrated circuits in digital filters will separate channels on multiplex equipment. Integrated circuits will also switch telephone circuits and perform data processing.

Computers will be more powerful, and will be organized in completely different ways. For example, memories built of integrated electronics may be distributed throughout the machine instead of being concentrated in a central unit. In addition, the improved reliability made possible by integrated circuits will allow the construction of larger processing units. Machines similar to those in existence today will be built at lower costs and with faster turn-around.

The author



Dr. Gordon E. Moore is one of the new breed of electronic engineers, schooled in the physical sciences rather than in electronics. He earned a B.S. degree in chemistry from the University of California and a Ph.D. degree in physical chemistry from the California Institute of Technology. He was one of the founders of Fairchild Semiconductor and has been director of the research and development laboratories since 1959.

Present and future

By integrated electronics, I mean all the various technologies which are referred to as microelectronics today as well as any additional ones that result in electronics functions supplied to the user as irreducible units. These technologies were first investigated in the late 1950's. The object was to miniaturize electronics equipment to include increasingly complex electronic functions in limited space with minimum weight. Several approaches evolved, including microassembly techniques for individual components, thin-film structures and semiconductor integrated circuits.

Each approach evolved rapidly and converged so that each borrowed techniques from another. Many researchers believe the way of the future to be a combination of the various approaches.

The advocates of semiconductor integrated circuitry are already using the improved characteristics of thin-film resistors by applying such films directly to an active semiconductor substrate. Those advocating a technology based upon films are developing sophisticated techniques for the attachment of active semiconductor devices to the passive film arrays.

Both approaches have worked well and are being used in equipment today.

The establishment

Integrated electronics is established today. Its techniques are almost mandatory for new military systems, since the reliability, size and weight required by some of them is achievable only with integration. Such programs as Apollo, for manned moon flight, have demonstrated the reliability of integrated electronics by showing that complete circuit functions are as free from failure as the best individual transistors.

Most companies in the commercial computer field have machines in design or in early production employing integrated electronics. These machines cost less and perform better than those which use "conventional" electronics.

Instruments of various sorts, especially the rapidly increasing numbers employing digital techniques, are starting to use integration because it cuts costs of both manufacture and design.

The use of linear integrated circuitry is still restricted primarily to the military. Such integrated functions are expensive and not available in the variety required to satisfy a major fraction of linear electronics. But the first applications are beginning to appear in commercial electronics, particularly in equipment which needs low-frequency amplifiers of small size.

Reliability counts

In almost every case, integrated electronics has demonstrated high reliability. Even at the present level of production—low compared to that of discrete components—it offers reduced systems cost, and in many systems improved performance has been realized.

Integrated electronics will make electronic techniques more generally available throughout all of society, performing many functions that presently are done inadequately by other techniques or not done at all. The principal advantages will be lower costs and greatly simplified design—payoffs from a ready supply of low-cost functional packages.

For most applications, semiconductor integrated circuits will predominate. Semiconductor devices are the only reasonable candidates presently in existence for the active elements of integrated circuits. Passive semiconductor elements look attractive too, because of their potential for low cost and high reliability, but they can be used only if precision is not a prime requisite.

Silicon is likely to remain the basic material, although others will be of use in specific applications. For example, gallium arsenide will be important in integrated microwave functions. But silicon will predominate at lower frequencies because of the technology which has already evolved around it and its oxide, and because it is an abundant and relatively inexpensive starting material.

Costs and curves

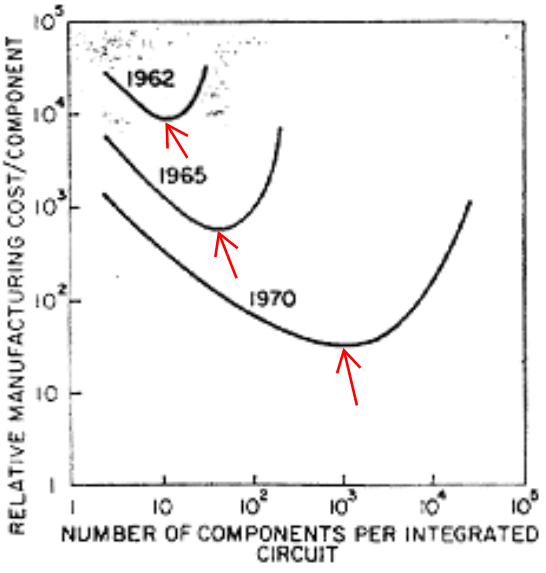
Reduced cost is one of the big attractions of integrated electronics, and the cost advantage continues to increase as the technology evolves toward the production of larger and larger circuit functions on a single semiconductor substrate. For simple circuits, the cost per component is nearly inversely proportional to the number of components, the result of the equivalent piece of semiconductor in the equivalent package containing more components. But as components are added, decreased yields more than compensate for the increased complexity, tending to raise the cost per component. Thus there is a minimum cost at any given time in the evolution of the technology. At present, it is reached when 50 components are used per circuit. But the minimum is rising rapidly while the entire cost curve is falling (see graph below). If we look ahead five years, a plot of costs suggests that the minimum cost per component might be expected in circuits with about 1,000 components per circuit (providing such circuit functions can be produced in moderate quantities.) In 1970, the manufacturing cost per component can be expected to be only a tenth of the present cost.

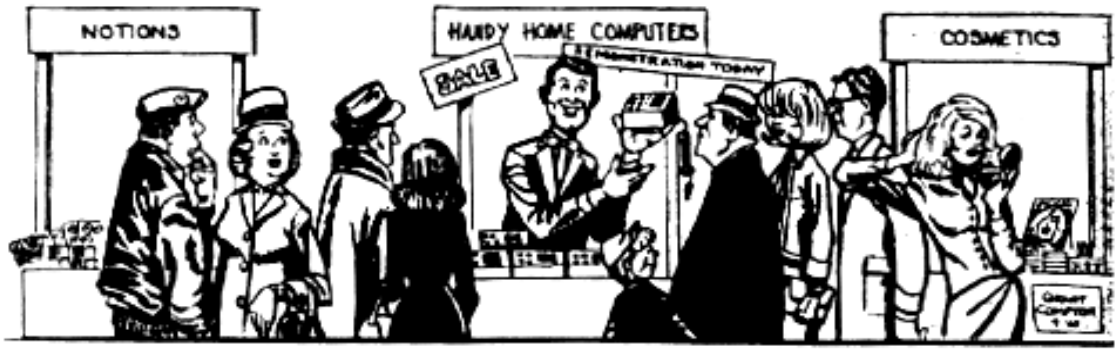
The complexity for minimum component costs has increased at a rate of roughly a factor of two per year (see graph on next page). Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000.

I believe that such a large circuit can be built on a single wafer.

Two-mil squares

With the dimensional tolerances already being employed in integrated circuits, isolated high-performance transistors can be built on centers two thousandths of an inch apart. Such



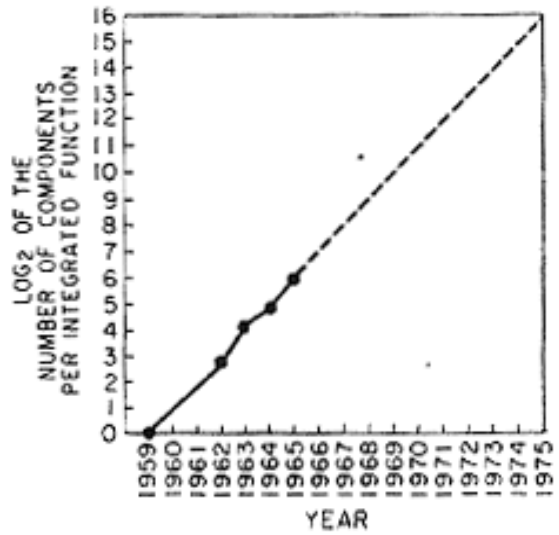


a two-mil square can also contain several kilohms of resistance or a few diodes. This allows at least 500 components per linear inch or a quarter million per square inch. Thus, 65,000 components need occupy only about one-fourth a square inch.

On the silicon wafer currently used, usually an inch or more in diameter, there is ample room for such a structure if the components can be closely packed with no space wasted for interconnection patterns. This is realistic, since efforts to achieve a level of complexity above the presently available integrated circuits are already underway using multilayer metalization patterns separated by dielectric films. Such a density of components can be achieved by present optical techniques and does not require the more exotic techniques, such as electron beam operations, which are being studied to make even smaller structures.

Increasing the yield

There is no fundamental obstacle to achieving device yields of 100%. At present, packaging costs so far exceed the cost of the semiconductor structure itself that there is no incentive to improve yields, but they can be raised as high as is economically justified. No barrier exists comparable to the thermodynamic equilibrium considerations that often limit yields in chemical reactions; it is not even necessary to do any fundamental research or to replace present processes. Only the engineering effort is needed.



In the early days of integrated circuitry, when yields were extremely low, there was such incentive. Today ordinary integrated circuits are made with yields comparable with those obtained for individual semiconductor devices. The same pattern will make larger arrays economical, if other considerations make such arrays desirable.

Heat problem

Will it be possible to remove the heat generated by tens of thousands of components in a single silicon chip?

If we could shrink the volume of a standard high-speed digital computer to that required for the components themselves, we would expect it to glow brightly with present power dissipation. But it won't happen with integrated circuits. Since integrated electronic structures are two-dimensional, they have a surface available for cooling close to each center of heat generation. In addition, power is needed primarily to drive the various lines and capacitances associated with the system. As long as a function is confined to a small area on a wafer, the amount of capacitance which must be driven is distinctly limited. In fact, shrinking dimensions on an integrated structure makes it possible to operate the structure at higher speed for the same power per unit area.

Day of reckoning

Clearly, we will be able to build such component-crammed equipment. Next, we ask under what circumstances we should do it. The total cost of making a particular system function must be minimized. To do so, we could amortize the engineering over several identical items, or evolve flexible techniques for the engineering of large functions so that no disproportionate expense need be borne by a particular array. Perhaps newly devised design automation procedures could translate from logic diagram to technological realization without any special engineering.

It may prove to be more economical to build large

systems out of smaller functions, which are separately packaged and interconnected. The availability of large functions, combined with functional design and construction, should allow the manufacturer of large systems to design and construct a considerable variety of equipment both rapidly and economically.

Linear circuitry

Integration will not change linear systems as radically as digital systems. Still, a considerable degree of integration will be achieved with linear circuits. The lack of large-value capacitors and inductors is the greatest fundamental limitations to integrated electronics in the linear area.

By their very nature, such elements require the storage of energy in a volume. For high Q it is necessary that the volume be large. The incompatibility of large volume and integrated electronics is obvious from the terms themselves. Certain resonance phenomena, such as those in piezoelectric crystals, can be expected to have some applications for tuning functions, but inductors and capacitors will be with us for some time.

The integrated r-f amplifier of the future might well consist of integrated stages of gain, giving high performance at minimum cost, interspersed with relatively large tuning elements.

Other linear functions will be changed considerably. The matching and tracking of similar components in integrated structures will allow the design of differential amplifiers of greatly improved performance. The use of thermal feedback effects to stabilize integrated structures to a small fraction of a degree will allow the construction of oscillators with crystal stability.

Even in the microwave area, structures included in the definition of integrated electronics will become increasingly important. The ability to make and assemble components small compared with the wavelengths involved will allow the use of lumped parameter design, at least at the lower frequencies. It is difficult to predict at the present time just how extensive the invasion of the microwave area by integrated electronics will be. The successful realization of such items as phased-array antennas, for example, using a multiplicity of integrated microwave power sources, could completely revolutionize radar.

Oblig 4 IN3030/IN4330, Spring 2026

Finding the Convex Hull of a Set of Points in a Plane: A Recursive Geometric Problem

Original version written by Arne Maus, this version edited by Eric Jul

Given a set of points P in a plane consisting of n (n >2) different points pi (i =0,1,2,…,n-1) with integer coordinates (*i.e.*, no decimals), where we can assume that *not all* points are on the same line, find the Convex Hull of the set of points. The convex is a polygon consisting of a subset of the points in P and with lines between some of these points such that all the other points are on the inside of this polygon and also so that all inner angles of this polygon are ≤ 180. In simpler terms convex means that there is no inwards dent on the set of lines running around the set of points (fig1).

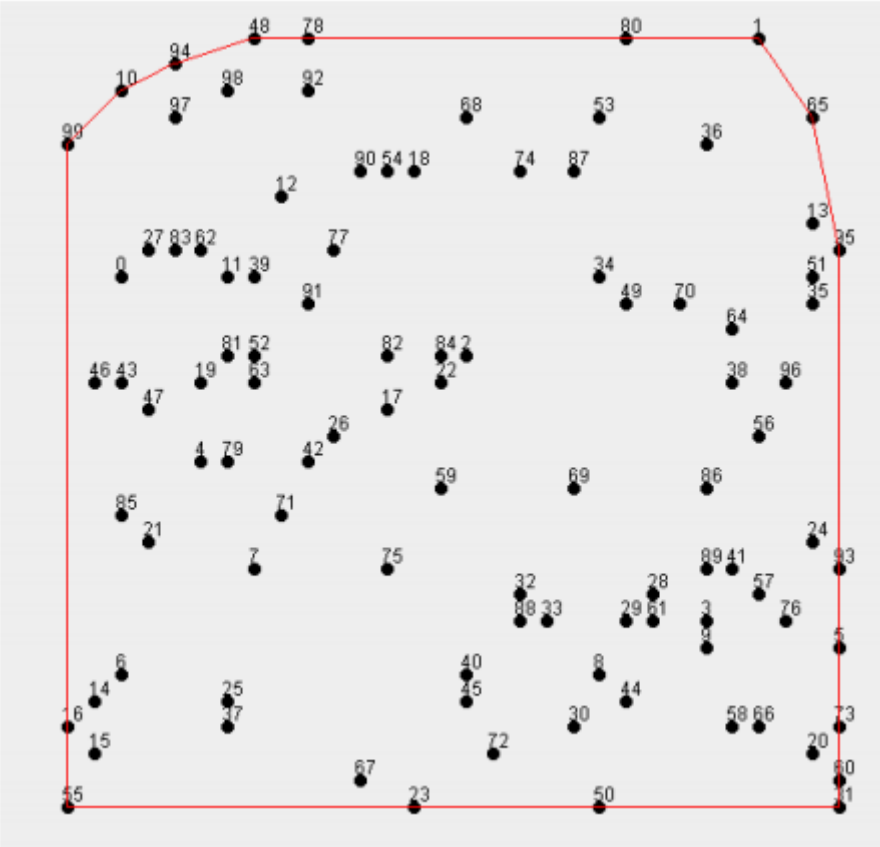


Fig1: The Convex Hull of 100 arbitrary points in a plane (the same as used in the assignment).

When finding such a convex hull, we start with an arbitrary point along the hull and jot down the following points that make up the hull *counter-clockwise*. In fig1, it could for instance be 16, 55, 23, 50, 31, 60, 73, 5, 93, 95, 65, 1, 80, 78, 48, 94, 10, 99. Notice that if there are more points on a line (for instance, 31, 60, 73, 5, 93, and 95), all of those points should be included in the hull. You are in other words not allowed to go straight from point 31 to 95, you have to specify all the lines 31-60, 60-73, 73-5, 5-93 and 93-95. We thus require that all points in P that are on one of the lines of the polygon, are part of the hull. This makes the subset of points included in the convex hull unique.
What point should you start with? Note that any point that has either the highest or lowest x or y value will always be on the hull, so any such point can be used as a starting point. For example points 31 or 55 in fig1.

How do we find the Convex Hull of n Points?

We can start with two points that we can see are obviously on the convex hull: the point that has the lowest value for x and the point that has the highest value for x. If there are more points with the same values, just arbitrarily choose one of those with the lowest and one of those with the highest values for x (in the example, for example 16 and 5). The rest of the algorithm is based on simple geometry, which is reviewed below.

Equation for a line

Each line from a point p1 (x1, y1) to p2 (x2, y2) can be written as:

$$\mathbf{a}x + \mathbf{b}y + \mathbf{c} = 0$$

Where: $\mathbf{a} = y1 - y2$, $\mathbf{b} = x2 - x1$ and $\mathbf{c} = y2 * x1 - y1 * x2$

Notice that this is a **straight line *from* p1 *to* p2.**

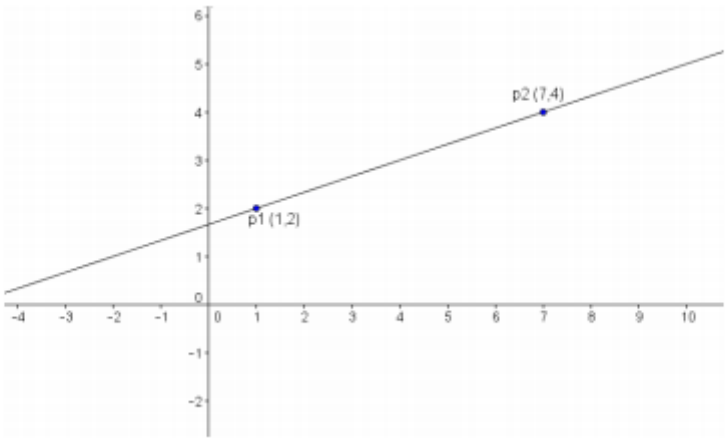


Fig2: A line from p1 (1,2) to p2 (7,4) has the line equation:
$(2 - 4)x + (7 - 1)y + (4 * 1 - 2 * 7) = 0$; i.e. $-2x + 6y - 10 = 0$

The line equation means that all points on this line satisfy the equation. We can also say that this line splits the plane in two: the points to the left side seen from the direction of the line - from p1 to p2 - and the points on the right side in the direction from p1 to p2.

The Distance from the Line to other Points

If we consider points that are not on the line p1-p2, we can see that all points, for example p3 (5,1) on the right side of the line gives a number <0 in the line equation ax + by + c, og all points to the left of the line, such as p4 (3,6) gives a number >0.

The further from the line the points are, the larger negatives or positive values they will have. Generally speaking we can say that the distance perpendicular down from a point p (x,y) to a line ax + by + c is:

$$d = \frac{ax + by + c}{\sqrt{a^2 + b^2}}$$

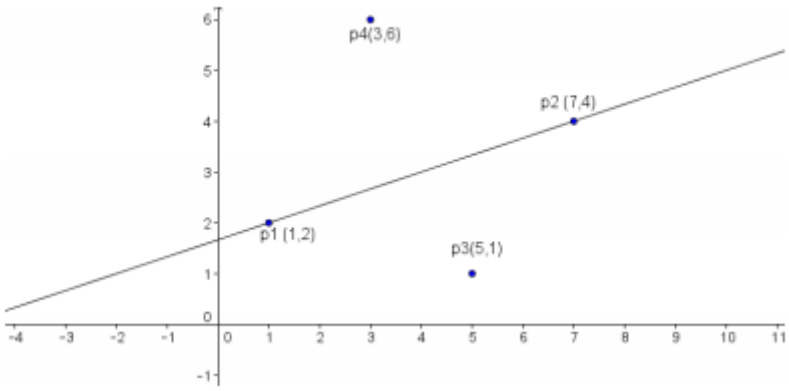


Fig3: The distance from a point to a line. The line divides the plane in two; those points with negative distance (to the right) and those with positive distance (to the left) from the line.

Consider two points in fig3: p3 (5,1) and p4 (3,6), we can calculate the distance from p4 to the line p1-p2 as:

$$\frac{-2*3+6*6-10}{\sqrt{(-2)^2+6^2}} = \frac{20}{\sqrt{40}} = 3{,}16..,$$

While the distance from p3 (5,1) to p1-p2 is:

$$\frac{-2*5+6*1-10}{\sqrt{40}} = \frac{-14}{6{,}32} = -2{,}21..$$

The Algorithm for the Convex Hull

After making another observation, we can formulate the algorithm for the convex hull. Observation: The point that has the longest (largest) negative distance from a line pi-pj is itself a point on the convex hull (see fig4, point P and line I - A).

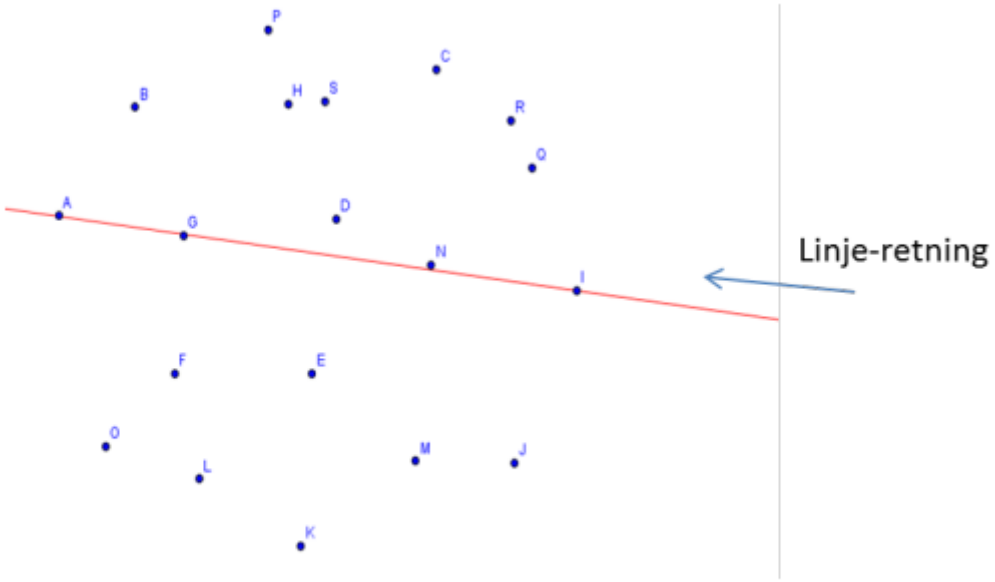


Fig.4. Starten på å finne innhyllinga fra maxx (I) til minx (A).

Fig4: The start to finding the convex hull from max_x (I) to min_x (A). **("linje-retning" translates to "direction of line"**

Convex Hull Algorithm:

1. Draw a line between the two points we know are on the hull from max_x to min_x (I to A in fig4)
2. Find the point with the largest negative distance (or 0) from the line (in fig4 it's P). If multiple points have the same distance, just choose one arbitrarily.
3. Draw a line from the two points on the line to this new point on the hull (in fig5: I-P and P-A)
4. Continue recursively from the two new lines and for each of them find a new point on the hull with the largest non-positive distance (less than or equal to 0)
5. Repeat step 3 and 4 until there are no more points on the outside of the lines
6. Repeat steps 2-4 for the line min_x-max_x, and find all points on the hull under it.

This process is illustrated in fig5. You can see the process of finding the points on the hull above the line I-A. Expect to find about 1.4 * sqrt(n) elements in the convex hull from n randomly chosen points on the plane, although all values from 3 to n are possible (we get n, if all points are on a circle or a triangle or a rectangle or any convex polygon).

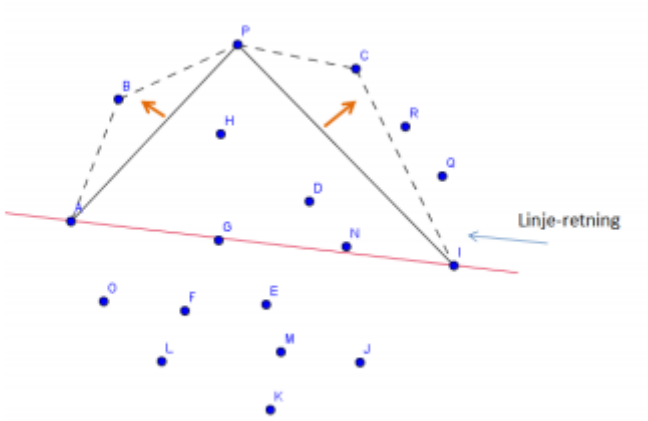


Fig5: We recursively find the point P on the hull from the line I-A (that with the largest negative distance from max_x-min_x), then C from P-I, R from I-C and lastly Q from I-R.

First, you are supposed to program this recursive algorithm sequentially, and then make a parallel version and measure speedups for n=100, 1000, 10 000, …, 10 million.

On Generating n Different Points

a) The data structure for your n points should be two int arrays x and y, each of them with length = n.

```
int []x = new int[n], y = new int[n];
```

In order to fill these arrays with x and y values that does not create equal points, you must use the pre-written class Npunkter, which you will find in the Oblig5 file.

b) For each value of n in your program, you should first create an object of the class NPunkter **(NPunkter translates to NPoints)**:

```
NPunkter p = new NPunkter(n);
```

and then fill `x[]` and `y[]` with a call to the method in the object p:

```
p.fyllArrayer(int [] x, int [] y);
```

(“fyllArrayer” translates to “fillArrays)

The reason why you must use this class, which you must not alter, is that all the deliveries for a given value of n should have the same points. Then it will be much easier for the correctors to correct the obligs (additionally, it is not easy to relatively quickly generate up to 10 million random points that you are certain are not equal, but that task is not a part of Oblig5).

c) The process of generating n points should be excluded from the reported execution times of the algorithm.

Efficiency

a) We are not really interested in the actual distance from a point to a line, it is just used as a relative measure to see which point "wins", *i.e.*, is furthest away from the line. The formula for the distance d can as such be simplified by ignoring the division.
b) It will be time-consuming if you have to run through the entire set of points P all the time to find the few points that are on the outside of a line. That is why you should let your program test on smaller and smaller sets of points
c) Remember to be critical regarding using threads instead of recursion (think "layers", like quicksort). Threads should only be used at the top of the recursive tree.
d) Is `ArrayList<Integer>` fast enough to hold large sets of points, or should you use the class `IntList` with the "same" methods you need, and where the integers you are supposed to store resides in a simple integer-array?

The Assignment

In Oblig 4 you must:

- Program a sequential version of the algorithm described above.
- Parallelize the sequential implementation preferably using one of the two methods that will be described in the course lectures: either method 1 or 2.
- Use the precode *NPunkter17.java* and *Oblig4Precode.java* published separately. You must use the `drawGraph()` to draw the graph.
- It must be possible to specify the seed for the precode on the command line when starting the program.
- The program should be able to print the resulting convex hull to a file using the `writeHullPoints()` in the precode for n < 10000.
- Write a report on your findings. Report recommendations are on the course web.
- **IN4330 students** must in addition: write a 1-2 page discussion of alternatives to their chosen method for paralelizing the problem. Either experiment with parameters for the chosen method, *e.g.*, vary the number of threads used, or implement one of the alternatives. In either case, compare and discuss the difference in performance.

Requirements

Oblig 4 Devilry deadline: See Oblig plan on course web page.

The delivery should consist of:

a) The code for both the sequential and parallel solution, wrapped in a .zip file.
b) An output of the convex hull for n = 1000.
c) A report with both a table and a curve showing speedup as a function of n (= 100, 1000, … 10 million) and your assessment as to why your runtimes are as low as they are, and why you get the speedups you get. NOTE: runtime for n = 100 milion should be less than 14s and n = 10 million should be less than 2s. Include specifics of the computer you ran your tests on (modell, clock frequency, number of cores) and the size of the main memory, and if possible the size of the caches and delay (run the

program latency.exe if you are on a Windows computer) and gladly a picture created by TegnUt when n=100.

Oblig 4 report recommendations have been published on the course page.

Tips

The following points are not requirements but describe areas where you might run into some trouble.

1) **The Sequential solution**
    a) **How to represent a point**
    Use the index in x[] and y[] - the contents of those will not change during runtime; they are only read.
    b) **Debugging**
    Very few, if any, manage to get their code right on the first run. We need to debug the code and that might be difficult on a graphical problem like this one. You can for instance use TegnUt to draw your lines and use a small number for n while debugging (n < 200). The call to TegnUt is made as such:

```
TegnUt tu = new TegnUt (this, koHyll);
```

    The first line is to draw the points and the convex hull that is expected to be added in to an `IntList coHull`.

    c) **Find points on the convex hull in the correct order**
    Tip: you are using two methods for the sequential solution (assuming here that you are using ArrayList, but this can trivially be changed by swapping it with a faster IntList):

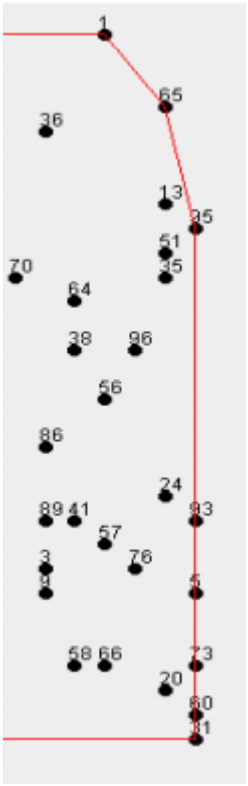
    seqMethod() which finds min_x, max_x, and starts the recursion. If you start on max_x-min_x, add max_x to the list of hull points (but not min_x yet), and start your recursion with two calls to the recursive method:

    seqRec(int p2, int p2, int p3, ArrayList m, ArrayList koHyll) which receives a set of points m (containing all points above or below the line p1-p2) and p3 that has already been found as the point with the largest negative or positive distance from p1-p2. koHyll is the set to which you should add the points from the convex hull in the correct order.

    Notice that you first “do a recursion” on the right side (line I-P in fig5) and then on the left side (line P-A) because we want the points to be added in a counter-clockwise order.

    You can let each call to seqRec add one point: p3 to the list of points on the hull, but where in your code should you do so? When do you add min_x to the list?

d) **Include all points on the hull in cases where multiple points are on the same line (distance = 0), such as on the right side in fig6, and in correct order.**
Remember that if you find that the largest negative distance == 0, there is no need to include p1 or p2 as possible new points (they have already been found).
Say that you have found p1=35 and p2=5 and you want to find all the points on the line between p1 and p2 (60 and 73) and you then have to test if a new point p3 has both x and y coordinates corresponding to those of p1 and p2. Then you can find one of the points with a call to seqRec above the line p1-p2 (31-5). Repeat recursively until there no longer are any points between the new p1 and p2.
The other points on the line (for instance, 5-95) will be found recursively similarly to 60 and 73.



2) The parallel solution

a) You could consider dividing the parallel case in two methods. The first parallel call from the main-thread (between the two lines where you time the parallel implementation). It will then find out how far down the call tree we should use threads. If we have k cores on our computer, it might be reasonable to move down to a layer in the call tree where there are approximately k nodes horizontally in the tree. Example: if we have 8 cores, a call the top of the tree level 1, then level 4 will have 8 nodes. Up until this level new threads will replace recursive calls, and from that level regular recursive calls can be used from each of the threads we started.

b) The first parallel method called from and being run from the main-method creates two threads. The following threads are born from these two threads and their offspring-threads. These are started in the parallel method, which is called from the run()-methods.

c) The other parallel method has the same parameters as the sequential one: p1, p2, p3, and the set of points to look within in addition to two more: the level of the tree and the set (IntList) of points on the hull that the thread has found.

d) The other parallel, recursive method decides whether to keep on creating threads or if we are going to start using recursive calls. Prior to that, we have found the point that has the largest negative distance and the set of points for the next call and we have also declared and generated to sets; one for each of those to recursions that are supposed to hold the convex hull that particular call finds.

e) This method strongly resembles the sequential method, and after we have reached the level where we seize to make new threads, we can actually just use the sequential recursive method.

f) Remember that when we start threads we have to first create both of them, and then wait (for instance by using join()) for them. If we do not, we do not get paralyzation.

Finding the points of the convex hull in the correct order (parallel solution)

Another problem you will face is adding points in the correct order. While it is pretty clear when to add points to the convex hull in the sequential solution, it is not as predictable in the parallel solution. A way to solve this is to use:

1. Threads started at the top of the recursion tree only traverse down to a certain level for all nodes of the tree (the level depends on the number of cores at our disposal)..
2. After we can no longer start new threads, each thread moves on to the sequential solution (on its side of data)
3. From each of these sequential executions we will get the points in the correct order (counter-clockwise)
4. The problem then is to join these sets of points that are individually in the correct order. Do this: add first the points the right-hand thread has produced, then the point found by the node itself, and then the points from the left-hand thread.
5. Point 4 is then executed (upon return after finding its points) upwards until we reach the top and is done.

Weekly assignment IN3030/IN4330 – v2026 (translated by Google)

Eric Jul based on a project by *Arne Maus*

During the group classes, the assistant teachers will help with parallel programming in Java. These weekly assignments are for practicing creating simple Java programs with threads. They are not mandatory, BUT it will be easier to do Obligatory 1, if you do it.

1. Hello world

The first thing everyone should do is create a program that generates as many threads k (in addition to the main thread) as there are cores on the machine you are sitting on. Each thread should have a local variable 'ind'. (ind = 0,1,..,antallTråder-1)

In the method 'run' you should first place this statement:

a) System.out.println("Thread no: " + ind+" says hello");

and then the following two statements in a try{..} catch block:

b) try { Thread.sleep(1000);
System.out.println("Thread no: " + ind+" says hello after waiting one second") ;
} catch (Exception e) { return;}

Run the program several times – both a) and b) – and check whether the statements come in the 'correct' order each time. Is the output mixed up?

2. Find the maximum value in an array

We will now parallelize the problem of finding the largest element in an integer array a[] of length n - and write: 'int findMax(int a[]) {..};

The array we will first create such an array with random content with the class Random in the package: java.util where we use the method nextInt(n) to pull the elements in [].

We will test this below for n = 100, 1000, ..., 10 million.

a) First create a sequential solution and take the time on this with System.nanoTime() then in nanoseconds and print the time in milliseconds. Divide the nanoseconds by 1000000.0 to get milliseconds as a double variable. Create a table of the results.

b) Parallel solution:
We will then look at two ways to parallelize this problem where we have k threads:

b.1 First, you divide the array so that thread 0 gets the first n/k elements, thread 1 has the next n/k elements, etc., and the last thread gets the rest of the array.

b.2 Then we divide the array a[] so that thread 0 tests element: 0, k, 2k, 3k,..etc. Thread 1 tests element no. 1, k+1, 2k+1, ... Thread 2 tests element 2, k+2, 2k+2,... Remember to test so that you do not go beyond the end of the array.

We see that in both cases the threads should update a common variable called e.g. globalMax. Here you can test out two strategies:

b.3 Use a synchronized method that all threads call for each element they have and that updates globalMax if the new element is greater than the old value of globalMax.

b.4 Let all threads each have their own variable localMax that they use to find the max in their part of the array. Finally, each thread then calls only once a synchronized method (same for setting globalMax in b.3) that then finds out which of the threads' local maxes was the largest.

Make a small report about the time consumption with the 4 possible variants of the algorithm + the time consumption of the sequential algorithm where you also calculate the speedup S. Also write whether it seems possible to get S >1 for any of these four alternatives. Comment on why some of the algorithms may be particularly fast or particularly slow.

3) If time: Solve the problem of creating a parallel version of the sum of the elements in an array

$$Sum = \sum_{i=0}^{a.length-1} a[i]$$

following the same template as exercise 2. Create a small report with the running times. Do you draw the same conclusions as with the finnMax problem?

February 11nd, 2026

Oblig 2 IN3030/IN4330 – v2026
Parallelization of Matrix Multiplication

This compulsory assignment (Oblig) is about Matrix Multiplication.

Given two matrices `A` and `B` of size `N` x `N`, you are to implement and run three variants of Matrix Multiplication `A x B`: one using the classic algorithm, one where `B` is transposed, and one where `A` Is transposed. For each version, you should implement and run both a sequential and a parallel version of Matrix Multiplication and measure and report on the speedup achieved.

You are to run with matrix sizes of `100x100`, `200x200`, `500x500`, and `1000x1000`.

Your program must be able to run regardless of the number of cores and should utilize them all by default; you can also optionally set a maximum number of cores as a parameter to the program.

In your solution include a check that you get the correct results, based on the sequential algorithm. Beware, that due to rounding errors, the results might differ very slightly—for floating point numbers `(a+b)+c` may not be exactly the same as `a+(b+c)`.

Show the speedup results both in table form and show a graphical representation of the speedup results.
You should achieve a speedup for at least some of the larger matrices.
Furthermore, show a graph comparing all six variant – use the variant that is the fastest (for large `N`) as a baseline, so that the others are depicted relative to the fastest.

You must use the published precode that we provide to fill the two arrays to be multiplied.

Be aware that the sequential code can be slow, especially for larger matrices.

You must write a short report explaining how you did the parallelization, how you synchronized and why (or why not) you achieved a speedup. Comment on why the speedup varies with different sizes of the matrices. Include the table and graphs in the report. It should also contain a short user's guide explaining how to execute your code. The submitted code must be executable by following the user's guide.
The report is to be delivered in PDF format. Your report, the Java code and any other file are to be uploaded as a single zip file or tar file. The uploaded code must be compilable and runnable. At least some of your speedups should be greater than one.

For IN4330 students: You must, in addition, run your program on two *different* machines, *e.g.*, your own laptop and one of Ifi's machines. Thus, the results and graphs are in two versions: one set for each machine. Comment on the difference.

Note: for the highly skilled parallel programmer – there are even better methods than this one, *e.g.*, *tiling* but we are requesting that you do the simpler transposition solution described above ;-)

Tiling will be discussed briefly in a later lecture.

Delivery: Deadline is February 24th , 2026 at 23:59:00 CET – that is 23:59:00 local time in Oslo.

Deliver in devilry.

Template for report:

1. Introduction
2. Sequential Matrix Multiplication – *short description*
3. Parallel Matrix Multiplication – *how you did the parallelization*
4. Measurements – *includes discussion, tables, graphs*
5. User guide – *how to run your program*
6. Conclusion – *just a short summary*

Appendix:

- Your Java code
- Any supporting files, *e.g.*, makefile, whatever …

Oblig3 IN3030/IN4330 v2025: Prime Numbers

Eric Jul and Paulo Ferreira, February 25th, 2026

This is mandatory assignment number 3 (Oblig3) for IN3030/IN4330, Spring 2026. The subject is calculation of prime numbers and the unique factorization of a given number into the product of only primes. You must program and run timings of a sequential and a parallel version of the Sieve of Eratosthenes and write a short report on your experience. You are to program a Java sequential version of the sieve of Eratosthenes and produce a Java parallel version that achieves speedups greater than one. The program must also sequentially calculate the factorization of large numbers based on the primes that your program generates. In the precode, there is a sequential version of the sieve, but it is hard to parallelize.

Your program must contain both the sequential and a parallel version of the sieve of Eratosthenes and of the factorization of large numbers. The program must take a parameter, N, greater than 16, and then first generate all the primes less than N, and thereafter calculate the factorization of the 100 largest numbers less than N * N. You must print the factorizations using the precode that we post on the web. Do not modify the precode. Simply place the file in the same folder as your own code, and use it as shown in the comments of the source file.

The program must also take a second parameter, k, the number of threads to be used, if zero then the program uses the number of cores on the machine. Note that your program must contain both the sequential and a parallel version of the sieve of Eratosthenes and both the sequential and a parallel version of factorizing a single number. You must test that your program (in both the sequential version and the parallel version) correctly calculates the primes less than 100 by printing the primes and manually checking them. You must run your program in the full version that finds all primes up to N and factorizes the largest 100 numbers less than N*N for the following four values of N: 2 million, 20 million, 200 million, 2 billion (note: 2 milliarder in Norwegian). For each run, you must print the execution times and speedups obtained. In the report, you must include the median times for a set of runs (at *least* 7 runs) and speedups achieved when generating primes and when factorizing the top 100 numbers. The 100 factorizations must be printed to a file using a class (our precode) that will be uploaded to the course web site. The order of the factors must be monotonically increasing. You MUST parallelize EACH factorization, i.e., use multiple threads even for a single factorization. You program should work for any number of cores.

Further requirements:

- Your program must *test* that the sequential and the parallel versions generate the SAME prime numbers.
- The report must be submitted in PDF and must include two graphs of the speedup, one for each algorithm, where the X-axis is the four values of N (perhaps log-scale), and the Y-axis is the speedup.
- The execution time for the sieve should not exceed 30 seconds – if it does, you are doing something wrong. Similarly, the time for factorization should not exceed 60 seconds.
- When printing timing values make sure to print the correct unit, *e.g.*, s, ms, ns, or whatever SI-unit you use.
- The program should print good error messages, if it is given incorrect/missing parameters.
- Use good programming style, *e.g.*, proper indentations, good variable names, *etc*.

The report must contain:

1. Introduction – *what this report is about.*
2. User guide – *how to run your program (short, but essential), include a very simple example.*
3. Parallel Sieve of Eratosthenes – *how you did the parallelization – consider including drawings.*
4. Parallel factorization of a large number – *how you did the parallelization – consider including drawings.*
5. Implementation – *a somewhat detailed description of how your Java program works & how tested.*
6. Measurements – *includes discussion, tables, graphs of speedups for the four values of N, number of cores used.*
7. Conclusion – *just a short summary of what you have achieved.*
8. Appendix – *the output of your program.*

Deadline

Deliver in devilry: Deadline is March 24th, 2025 at 23:59 CET – that is 23:59:00 local time in Oslo.

Recommendations for the report for Oblig5, IN3030/IN4330

Eric Jul, April 2026

Oblig 5: Synchronization

Further requirements

- The program should print good error messages, e.g., if it is given incorrect/missing parameters.
- Use good programming style, e.g., proper indentations, good variable names, etc.
- The report MUST be handed in as a PDF file.
- All submitted files should be contained in a single zip-file.

The report must contain:

1. Introduction – *what this report is about*
2. User guide – *how to run your program (short, but essential), include a very simple example. Do not forget to describe the parameter line arguments to the program. Also include any instructions necessary to BUILD the program*
3. Explain how you did the synchronization – *consider including drawings*
4. Implementation – *a somewhat detailed description of how your Java program works*
5. Testing – how did you test the program? Provide convincing test output
6. Conclusion – *just a short summary of what you have achieved*

April 22nd, 2026

Oblig 5: Synchronization

Mandatory Assignment 5 in IN3030 / IN4330

Eric Jul

Deadline: 2026-05-05, 23:59

This oblig is about implementation of one synchronization primitive using another.

You are to implement a variant of waitNext that was made two years ago in the exam.
You are welcome to base your solution on the published WaitNextC.java.

You are to write a program that implements a new synchronization primitive called waitAndSwap3 with the following semantics:

When a thread makes the first call of waitAndSwap3, it waits. When a second thread calls waitAndSwap3, it waits. When a third thread calls waitAndSwap3, it waits. When a fourth thread calls waitAndSwap3, it releases the third thread and then the second thread and then the first thread whereafter it waits. When a fifth thread calls waitAndSwap3, it waits. When a sixth thread calls waitAndSwap3, it waits. When a seventh thread calls waitAndSwap3, it releases the sixth, the fifth and the fourth thread in that order.
And so on for all the following calls to waitAndSwap3.
The effect is that the threads are released from waitAndSwap3 in the following call order:
3, 2, 1, 6, 5, 4, 9, 8, 7, 12, 11, 10, …
We can say that the threads are “swapped” as groups of three.

You are welcome to base your solution on the published WaitNextC.java.

You are to write a test program that demonstrates that your waitAndSwap3 works.
You are welcome to do this using the test output method debugPrintln and the variSpeed method from the WaitNextC.java program.

Hand in the test program including the implementation of waitAndSwap3 and suitable output showing your synchronization method works as intended. Explain in a report that follows the report guidelines.

NOTE: you are to use only semaphores for synchronization and you can use *only* the acquire and release methods except that you can use access methods such as availablePermits but *only inside test output* similar to the printSems method.

IMPORTANT: there must be substantial amounts of code and a good deal of the report done before you can qualify for a second try. This means that if either the code or the report is missing, you will automatically fail with no offer of a second try.

Sketch about synchronization

Eric Jul and Michael Kirkedal Thomsen

IN3030/IN4330, Spring 2026

Manuscript synchronization sketch IN3030 lecture 11/3-2026 02:15pm at University of Oslo, Small Auditorium, Kristen Nygaards Hus.

1. Michael enters the room several minutes before the start of the lecture – Michael should stand near the blackboard that is closest to the windows. Eric enters hurriedly at 14:15:05 and heads for the blackboard to the left as seen from the students.

2. As soon as Eric arrives at the blackboard, both starts writing on each their blackboards “IN3030 Today: Synchronization” and bids welcome to the lecture.

3. Discover each other:

> Eric: "Hi, what are YOU doing here?"
> Michael: "I am doing today’s lecture!”
> Eric: "No, I am doing today’s lecture!”

4. Both pull out a version of the lecture plan:

> Michael (pulls out a printed schedule): ”No, I am doing today’s lecture, see this schedule!”
> Eric (pulls out a printed schedule): ”No, I am doing today’s lecture, see THIS schedule!”
> Michael: ”Mine is the REVISED schedule!” (points to the word REVISED)
> Eric: ”Mine also says REVISED!” (points to the word REVISED)

5. EJ: "What do we do about this?”

6. Short pause, then: both *in unison* (triggered by Eric sweeping his hand across toward Michael): "After you!!!" (repeat 3 times, *i.e.*, a total of 4 times).

7. Eric: "This is not working out!! How are we to decide who should do the lecture?? I must be able to devise a method for us to decide who should lecture – after all, I AM a PROFESSOR!” (Eric scratches his head…) – ”… hey, I have a great idea! A FLAWLESS system that will ensure that there is only ONE lecturer!”

8. Eric: Sets up an opaque bowl and explains: “Here is how we ensure that there is no more than one lecturer: When entering, a lecturer goes over to the bowl and check whether or not the bowl is empty. If empty, the lecturer then finds a piece of chalk and puts it into the bowl, indicating that there now *is* a lecturer. If NOT empty then the lecturer knows that there ALREADY is another lecturer, and thus can leave. This FLAWLESS system thus *ensures* that there is ONLY one lecturer!”

9. Eric then walks over to the bowl, checks that it is empty then moves away gets a piece of chalk. He then starts bragging about his FLAWLESS system to ensure that only one will be lecturing while waving the piece of chalk in his hand. As Eric is walking away from the bowl, Michael resolutely and quickly moves over to the bowl, checks it (flip it upside-down), then swiftly finds a piece of chalk, dumps it into the bowl and then moves to the blackboard away from the door and starts writing the agenda on the board.

10. When Eric is done bragging about his FLAWLESS system, he goes to the bowl and drops the piece of chalk into the bowl and then goes back to the board. BOTH start lecturing – saying welcome and that today, we will talk about *synchronization* – BOTH stop when discovering that there are **TWO** lecturers.

11. Small discussion: Michael (sarcastic, shaking head): “FLAWLESS system indeed!”.

12. Eric (again scratching his head): “Let me think … Ah! Now I know: I need TWO bowls. And now we start by putting chalk into one’s OWN bowl and then checking the OTHER bowl. Eric: "This method ensures that there is AT MOST ONE LECTURER!”

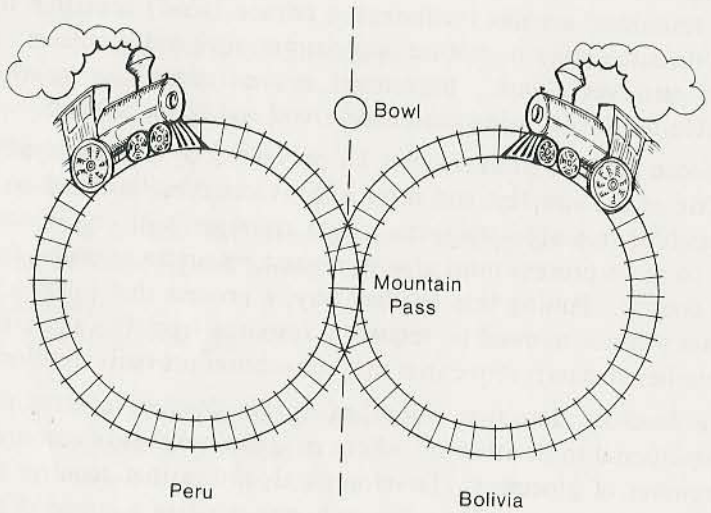
13. Both now go to the blackboard, find a piece of chalk, put it into one’s own bowl, then goes over to the other bowl to find chalk in it. Then both points to the other and says: “Ah, it is YOU that will lecture today!” and then both pack up and leave.

14. Eric returns and starts the actual lecturing.

Assignment: (Think carefully about your answer – and explain, *e.g.*, with examples):

- **What went wrong in point 4, 6, 9-10, and 12? Suggest a fix for the problem in 6.**
- **Is the statement at the end of point 8 correct?**
- **Is the statement at the end of point 12 correct? Why is it not sufficient?**

14. High in the Andes Mountains, there are two circular railroad lines. As shown in the diagram, one line is in Peru, the other in Bolivia. They share a section of track, where the lines cross a mountain pass that lies on the international border.



Unfortunately, the Peruvian and Bolivian trains occasionally collide when simultaneously entering the critical section of track (the mountain pass). The trouble is, alas, that the drivers of the two trains are blind and deaf, so they can neither see nor hear each other.

The two drivers agreed on the following method of preventing collisions. They set up a large bowl at the entrance to the pass. Before entering the pass, a driver must stop his train, walk over to the bowl, and reach into it to see if it contains a pebble. If the bowl is empty, the driver finds a pebble and drops it in the bowl, indicating that his train is entering the pass; once his train has cleared the pass, he must walk back to the bowl and remove his pebble, indicating that the pass is no longer being used. Finally he walks back to the train and continues down the line. If a driver arriving at the pass finds a pebble in the bowl, he leaves the pebble there; he repeatedly takes a siesta and re-checks the bowl until he finds it empty.

Then he drops a pebble in the bowl and drives his train into the pass. A smart aleck college graduate from the University at La Paz (Bolivia) claimed that subversive train schedules made up by Peruvian officials could block the Bolivian train forever. (Explain). The Bolivian driver just laughed and said that could not be true because it never happened. (Explain). Unfortunately, one day the two trains crashed. (Explain).

Following the crash, our college graduate was called in as a consultant to ensure that no more crashes would occur. He explained that the bowl was being used in the wrong way. The Bolivian driver must wait at the entry until the bowl is empty, drive through the pass and walk back to put a pebble in the bowl. The Peruvian driver must wait at the entry until the bowl contains a pebble, drive through the pass and walk back to remove the pebble from the bowl. Sure enough, his method prevented crashes. Prior to this arrangement, the Peruvian train ran twice a day and the Bolivian train ran once a day. The Peruvians were very unhappy with the new arrangement. (Why?)

Our college graduate was called in again and was told to prevent crashes while avoiding the problem of his previous method. He suggested that two bowls be used, one for each driver. When a driver reaches the entry, he first drops a pebble in his bowl, then checks the other bowl to see if it is empty. If so, he drives his train through the pass, stops it and walks back to remove his pebble. But if he finds a pebble in the other bowl he goes back to his bowl and removes his pebble. Then he takes a siesta, again drops a pebble in his bowl and re-checks the other bowl, and so on, until he finds the other bowl empty. This method worked fine until late in May, when the two trains were simultaneously blocked at the entry for many siestas. (Explain.)