

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

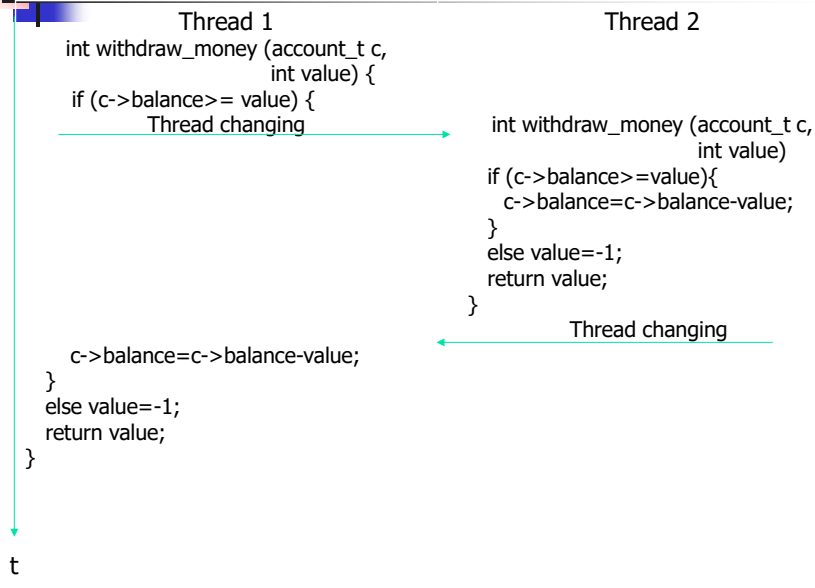
Concurrent Execution

```
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

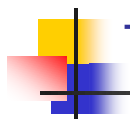
18



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;
```

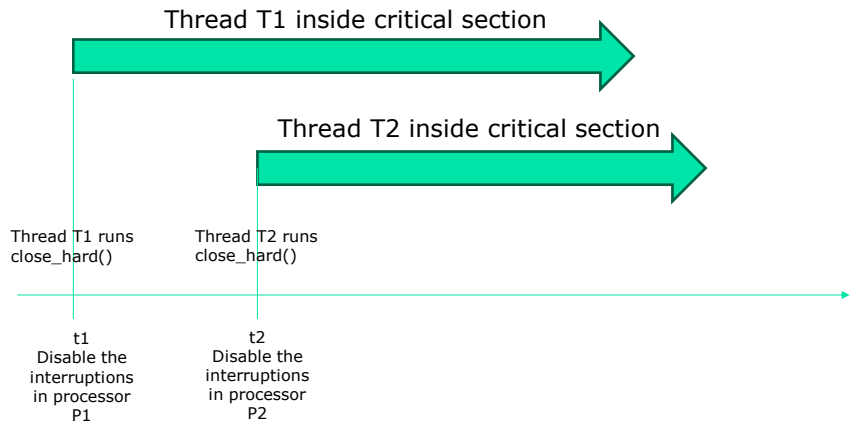
```
close_hard () {
   mask = mask_int ();
}
```

```
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

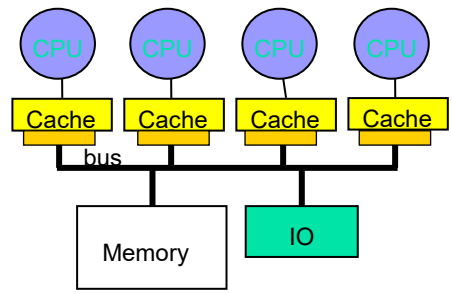
open_hard:
        MOV OPEN, AX
        MOV AX, latch
        RET
```

- 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

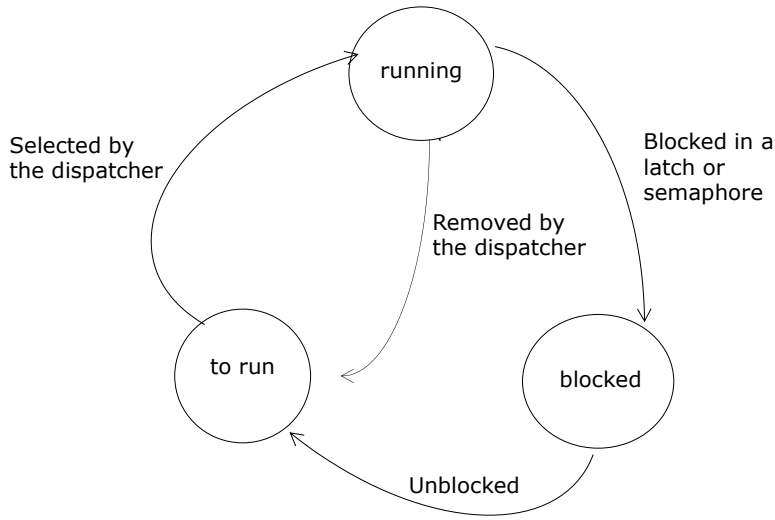


Solutions with Support from the OS

- 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

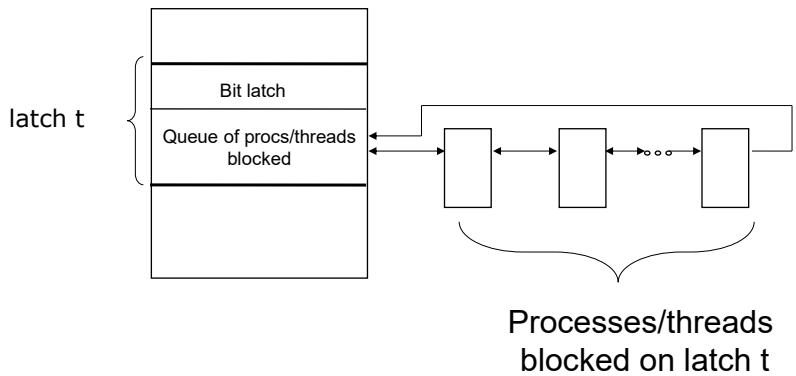
38

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.

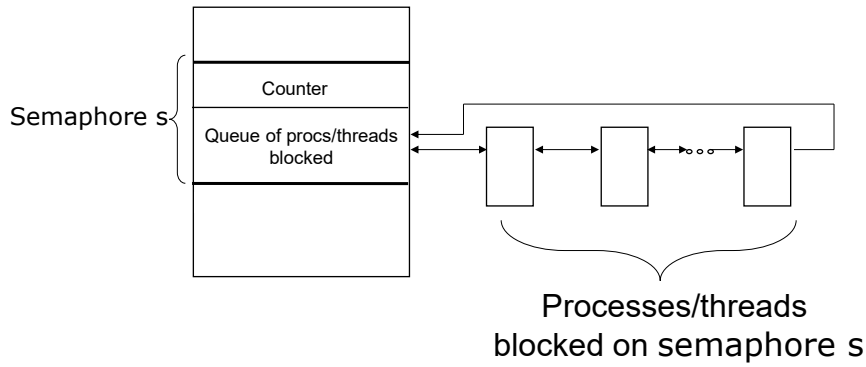
These operations are done atomically

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);

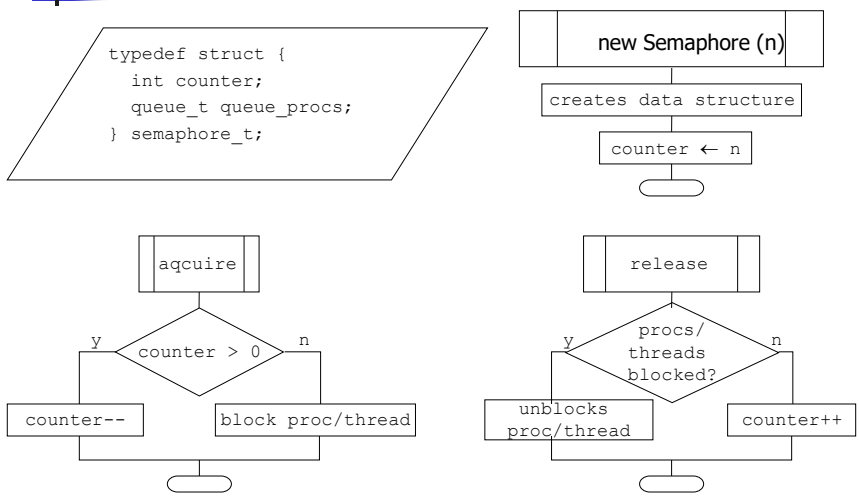
// Blocks the proc/thread if the counter is less or equal to zero; otherwise, it decrements the counter
sem.acquire();

// 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

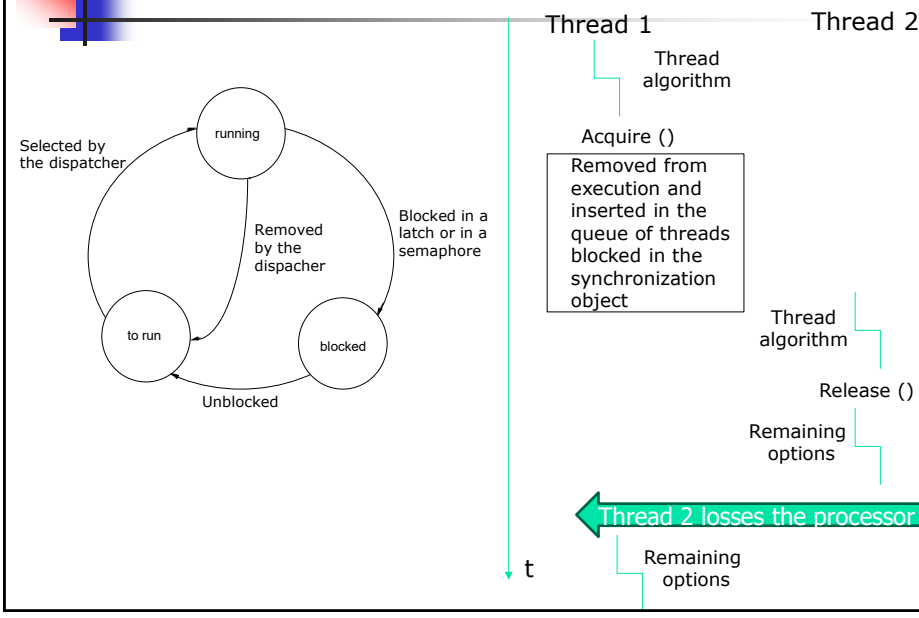
46

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