INF1060:
Introduction to Operating Systems and Data Communication

Operating Systems:

Processes & CPU Scheduling

Thursday, October 3, 2013

Overview

- Processes
  - primitives for creation and termination
  - states
  - context switches
  - (processes vs. threads)

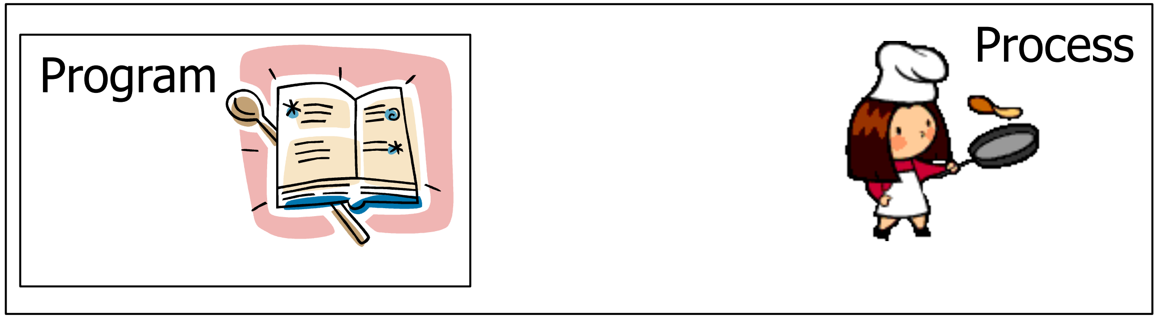
- CPU scheduling
  - classification
  - timeslices
  - algorithms

Processes

Processes

- What is a process?

  The "execution" of a program is often called a process



- Process table entry (process control block, PCB):

Process management	Memory management	File management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Process Creation

- A process can create another process using the `pid_t fork(void)` system call (see `man 2 fork`) :

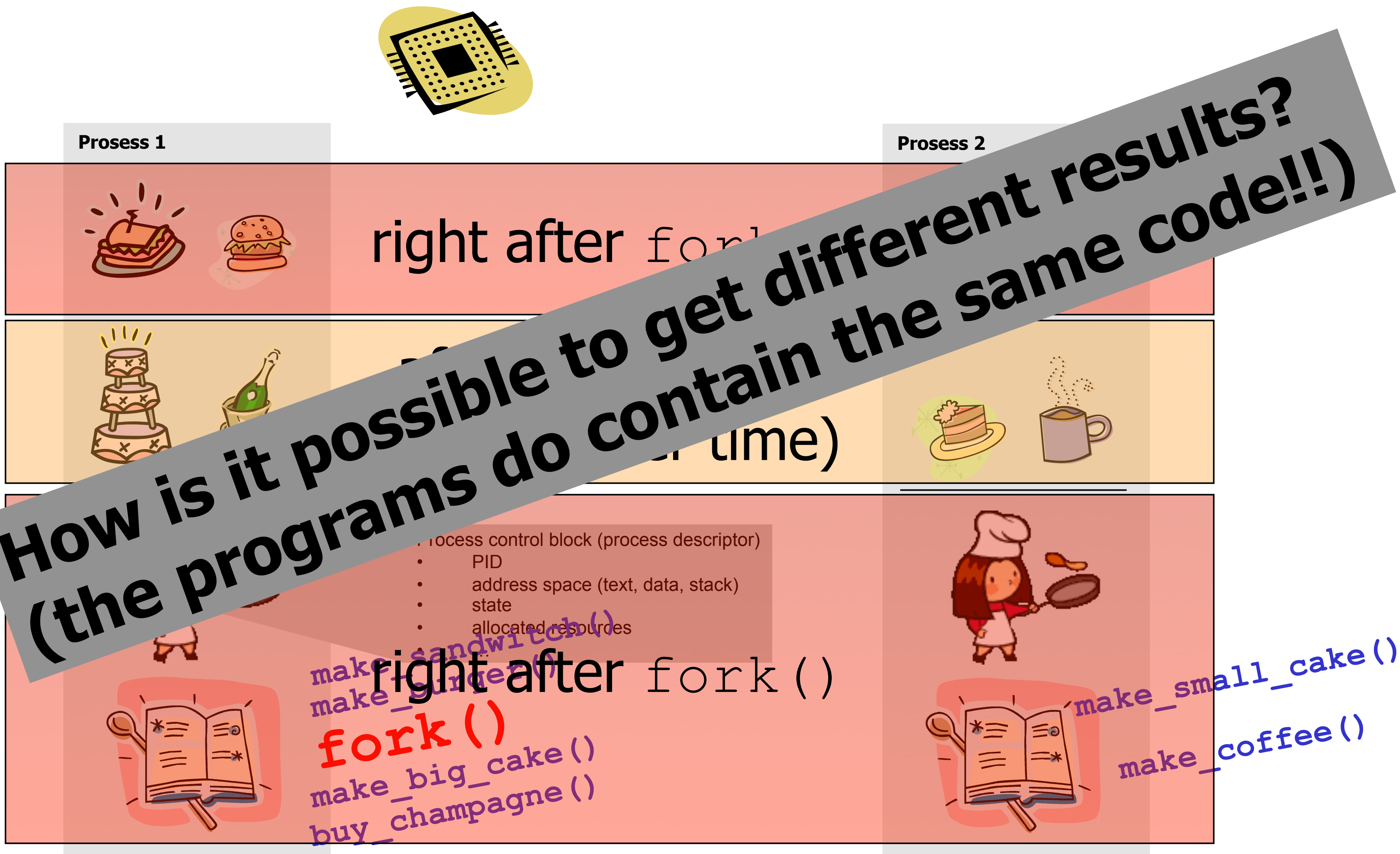
  - makes a **duplicate** of the calling process including a copy of virtual address space, open file descriptors, etc…
    (only PIDs are different – locks and signals are not inherited)

  - return value if …
    - …parent: child process' PID when successful, -1 otherwise
    - …child: 0 (if successful - if not, there will not be a child)

  - both processes continue in parallel

- Other possibilities include
  - `int clone(…)` – shares memory, descriptors, signals (see `man 2 clone`)
  - `pid_t vfork(void)` – suspends parent in `clone()` (see `man 2 vfork`)

Process Creation – fork()



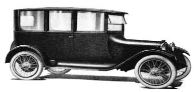
How is it possible to get different results?
(the programs do contain the same code!!)

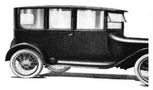
Program Execution

- To make a process execute a program, one might use the `int execve(char *filename, char *params[], char *envp[])` system call (see `man 2 execve`):
  - executes the program pointed to by `filename` (binary or script) using the parameters given in `params` and in the environment given by `envp`
  - return value
    - no return value on success, actually no process to return to
    - -1 is returned on failure (and `errno` set)
- Many other versions (frontends to **execve**) exist, e.g., `execl`, `execlp`, `execle`, `execv` and `execvp` (see `man 3 exec`)

fork(..)

execve(..)

process 1: 

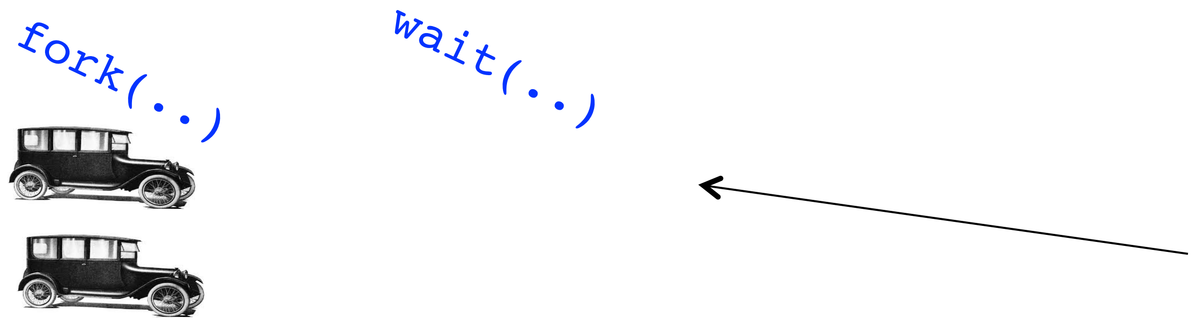
process 2:  

Process Waiting

- To make a process wait for another process, one can use the `pid_t wait(int *status)` system call (see `man 2 wait`):
  - waits until *any* of the child processes terminates (if there are running child processes)
  - returns
    - -1 if no child processes exist
    - PID of the terminated child process and puts the status of the process in `status`
  - see also `wait4, waitpid`

process 1:

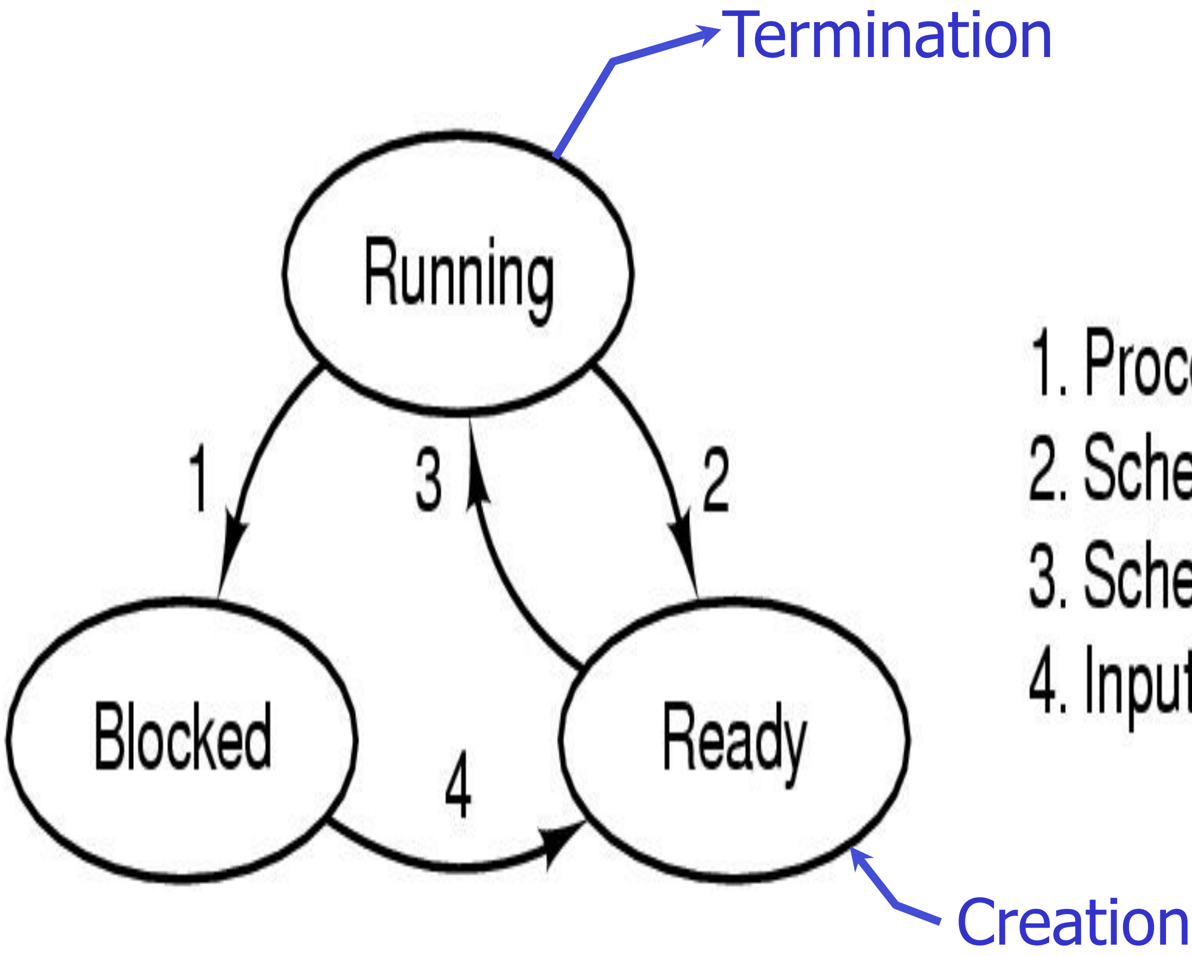
process 2:



Process Termination

- A process can terminate in several different ways:
  - no more instructions to execute in the program – unknown status value
  - a function in a program finishes with a **`return`** – parameter to return the status value
  - the system call `void exit(int status)` terminates a process and returns the status value (see `man 3 exit`)
  - the system call `int kill(pid_t pid, int sig)` sends a signal to a process to terminate it (see `man 2 kill, man 7 signal`)
- A status value of 0 indicates success, other values indicate errors

Process States



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Context Switches

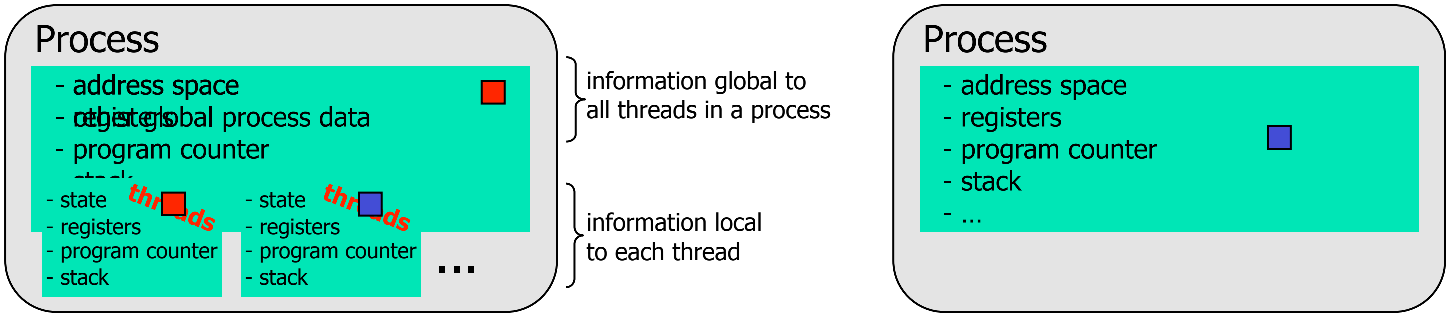
- Context switch: the process of switching one running process to another
  1. stop running *process 1*
  2. storing the state (like registers, instruction pointer) of *process 1* (usually on stack or PCB)
  3. restoring state of *process 2*
  4. resume operation on program counter for *process 2*

  – essential feature of multi-tasking systems
  – computationally intensive, important to optimize the use of context switches
  – some hardware support, but usually only for general purpose registers

- Possible causes:
  – scheduler switches processes (and contexts) due to algorithm and time slices
  – interrupts
  – required transition between user-mode and kernel-mode

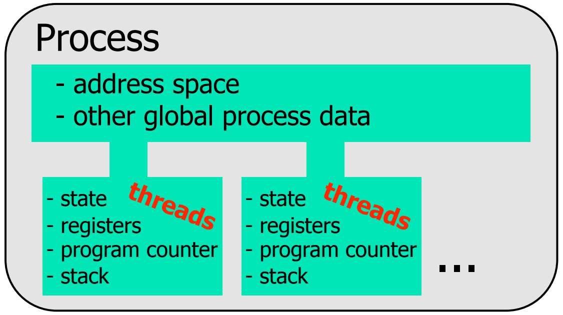
Processes vs. Threads

- Processes: resource grouping and execution
- Threads (light-weight processes)
  - enable more efficient cooperation among execution units
  - share many of the process resources (most notably address space)
  - have their own state, stack, processor registers and program counter



Processes vs. Threads

- Processes: resource grouping and execution
- Threads (light-weight processes)
  - enable more efficient cooperation among execution units
  - share many of the process resources (most notably address space)
  - have their own state, stack, processor registers and program counter



Example: time using `futex` to suspend and resume processes (incl. systemcall overhead):

Intel 5150:	~1900ns/process switch,	~1700ns/thread switch
Intel E5440:	~1300ns/process switch,	~1100ns/thread switch
Intel E5520:	~1400ns/process switch,	~1300ns/thread switch
Intel X5550:	~1300ns/process switch,	~1100ns/thread switch
Intel L5630:	~1600ns/process switch,	~1400ns/thread switch
Intel E5-2620:	~1600ns/process switch,	~1300ns/thread contex

http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html

  - no memory address switch
  - thread switching is much cheaper
  - parallel execution of concurrent tasks within a process

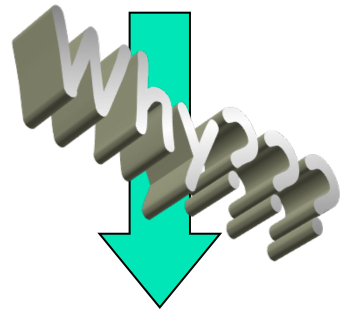
- No standard, several implementations (e.g., Win32 threads, Pthreads, C-threads) (see `man 3 pthreads`)

Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main(void){
   pid_t pid, n;
   int status = 0;

   if ((pid = fork()) == -1) {printf("Failure\n"); exit(1);}

   if (pid != 0) {   /* Parent */
      printf("parent PID=%d, child PID = %d\n",
                               (int) getpid(), (int) pid);

      printf("parent going to sleep (wait)...\n");

      n = wait(&status);

      printf("returned child PID=%d, status=0x%x\n",
                               (int)n, status);
      return 0;
   } else {          /* Child */
      printf("child PID=%d\n", (int)getpid());
      printf("executing /store/bin/whoami\n");
      execve("/store/bin/whoami", NULL, NULL);
      exit(0);       /* Will usually not be executed */
   }
}
```

[vizzini] > ./testfork
parent PID=2295, child PID=2296
parent going to sleep (wait)...
child PID=2296
executing /store/bin/whoami
paalh
returned child PID=2296, status=0x0

[vizzini] > ./testfork
child PID=2444
executing /store/bin/whoami
parent PID=2443, child PID=2444
parent going to sleep (wait)...
paalh
returned child PID=2444, status=0x0



Two concurrent processes running, **scheduled** differently

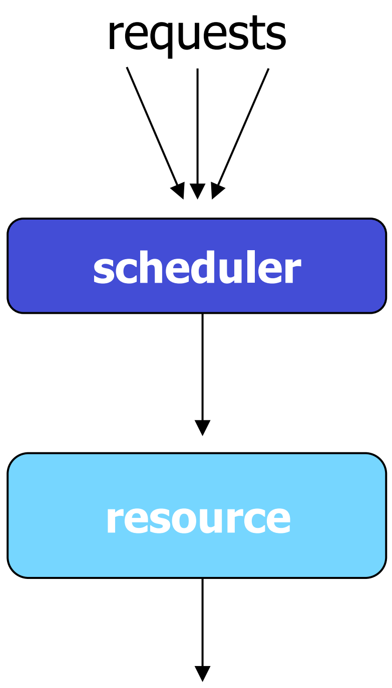
CPU Scheduling

Scheduling

- A task is a schedulable entity/something that can run
  (a process/thread executing a job, e.g.,
  a packet through the communication
  system or a disk request through the file system)

- In a multi-tasking system, several tasks may wish to use a resource simultaneously

- A scheduler decides which task that may use the resource, i.e., determines order by which requests are serviced, using a scheduling algorithm

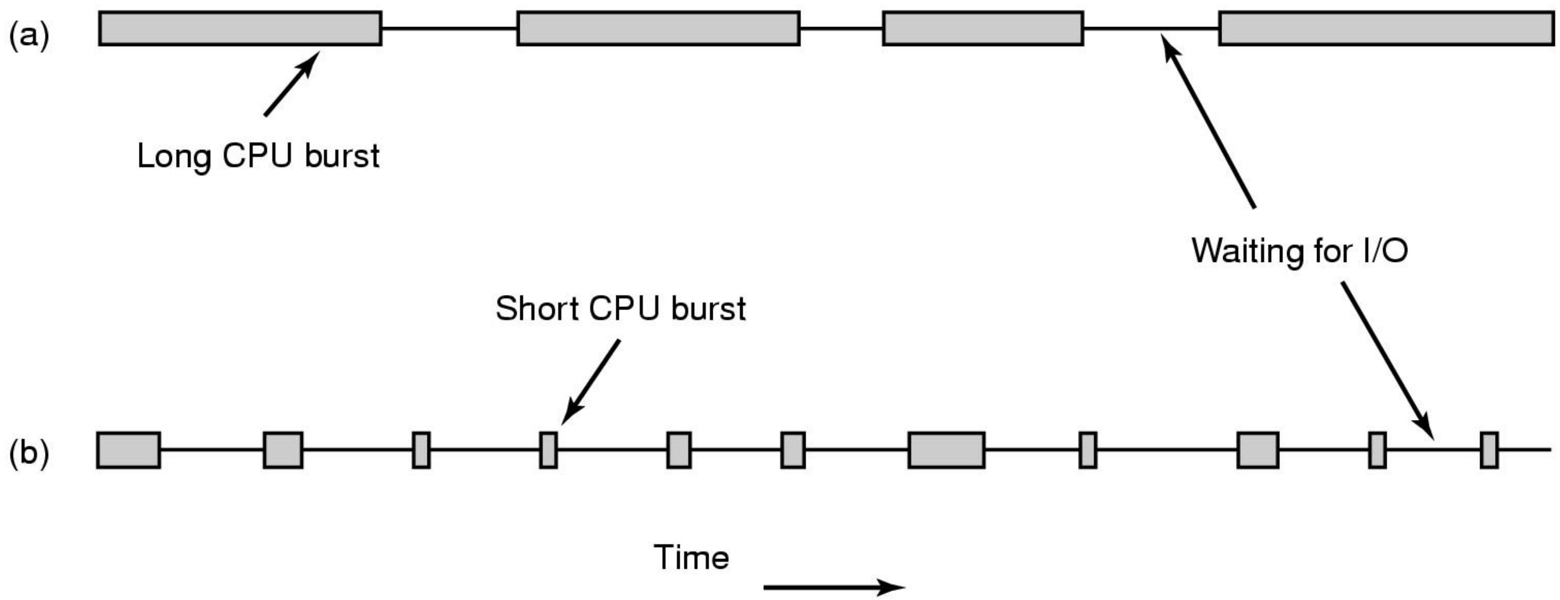


Why Spend Time on Scheduling?

- Scheduling is difficult and takes time – RT vs NRT example

Why Spend Time on Scheduling?

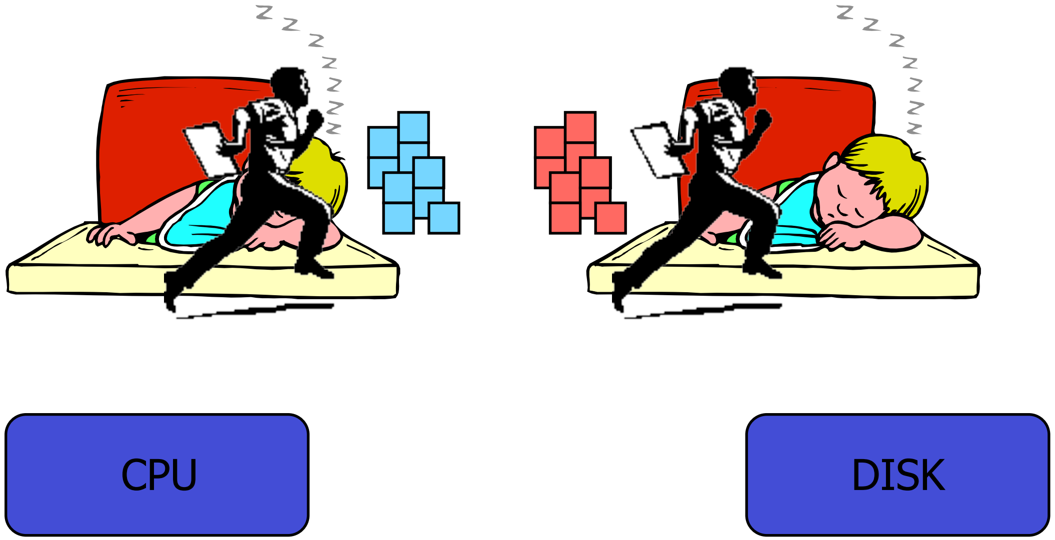
- Optimize the system to the given goals
  - e.g., CPU utilization, throughput, response time, waiting time, fairness, …
- Example: CPU-Bound vs. I/O-Bound Processes:



  - Bursts of CPU usage alternate with periods of I/O wait

Why Spend Time on Scheduling?

- Example: CPU-Bound vs. I/O-Bound Processes (cont.) – observations:
  - schedule all CPU-bound processes first, then I/O-bound



  - schedule all I/O-bound processes first, then CPU-bound?
  - possible solution:
    mix of CPU-bound and I/O-bound: overlap slow I/O devices with fast CPU

FIFO and Round Robin

FIFO:

- Run
  - to completion (old days)
  - until blocked, yield or exit
- Advantages
  - simple
- Disadvantage
  - when short jobs get behind long

Round-Robin (RR):

- FIFO queue
- Each process runs a timeslice
  - each process gets 1/n of the CPU in max t time units at a time
  - the preempted process is put back in the queue

FIFO and Round Robin

- Example: 10 jobs and each takes 100 seconds

- FIFO – the process runs until finished, no overhead (!!??)
  - start: job1: 0s, job2: 100s, ... , job10: 900s → **average** 450s
  - finished: job1: 100s, job2: 200s, ... , job10: 1000s → **average** 550s
  - unfair, but some are lucky

- RR - time slice of 1s, no overhead (!!??)
  - start: job1: 0s, job2: 1s, ... , job10: 9s → **average** 4.5s
  - finished: job1: 991s, job2: 992s, ... , job10: 1000s → **average** 995.5s
  - fair, but no one are lucky

- Comparisons
  - FIFO better for long CPU-intensive jobs (there **is** overhead in switching!!)
  - but RR much better for interactivity!

- **But, how to choose the right time slice??**

Case: Time Slice Size

- Resource utilization example
  - **A** and **B** run forever, and each uses 100% CPU
  - **C** loops forever (1 ms CPU and 10 ms disk)
  - assume no switching overhead

- Large or small time slices?
  - nearly 100% of CPU utilization regardless of size
  - Time slice 100 ms: nearly 5% of disk utilization with RR
    [A:100 + B:100 + C:1 → 201 ms CPU vs. 10 ms disk]
  - Time slice 1 ms: nearly 91% of disk utilization with RR
    [5x (A:1 + B:1) + C:1 → 11 ms CPU vs. 10 ms disk]

- What do we learn from this example?
  - The right time slice (in this case shorter) can improve overall utilization
  - CPU bound: benefits from having longer time slices (>100 ms)
  - I/O bound: benefits from having shorter time slices (≤10 ms)

Scheduling

- A variety of (contradicting) factors to consider
  - treat similar tasks in a similar way
  - no process should wait forever
  - short response times ($time_{request\ submitted}$ - $time_{response\ given}$)
  - maximize throughput
  - maximum resource utilization (100%, but 40-90% normal)
  - minimize overhead
  - predictable access
  - …

- Several ways to achieve these goals, …
  …but which criteria is most important?

Scheduling

- "Most reasonable" criteria depends on who you are
  - Kernel
    - Resource management
      - processor utilization, throughput, fairness
  - User
    - Interactivity
      - response time
        (*Example:* when playing a game, we will not accept waiting 10s each time we use the joystick)
    - Predictability
      - identical performance every time
        (*Example:* when using the editor, we will not accept waiting 5s one time and 5ms another time to get echo)
- "Most reasonable" criteria depends on environment
  - Server vs. end-system
  - Stationary vs. mobile
  - …

Scheduling

- "Most reasonable" criteria depends on target system
  - Most/All types of systems
    - fairness – giving each process a fair share
    - balance – keeping all parts of the system busy
  - Batch systems
    - turnaround time – minimize time between submission and termination
    - throughput – maximize number of jobs per hour
    - (CPU utilization – keep CPU busy all the time)
  - Interactive systems
    - response time – respond to requests quickly
    - proportionality – meet users' expectations
  - Real-time systems
    - meet deadlines – avoid loosing data
    - predictability – avoid quality degradation in multimedia systems

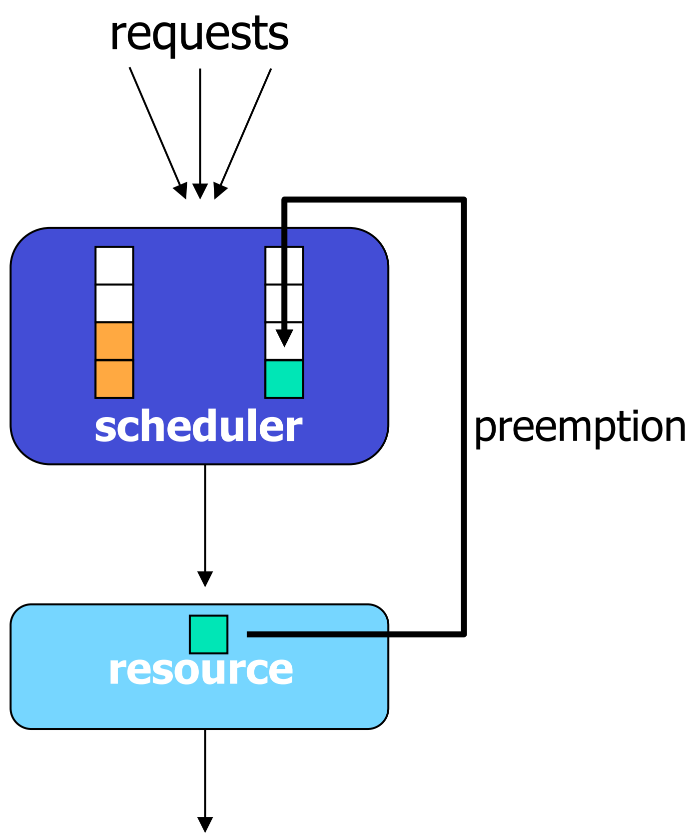
Scheduling

- Scheduling algorithm classification:
  - dynamic
    - make scheduling decisions at run-time
    - flexible to adapt
    - considers only the actual task requests and execution time parameters
    - large run-time overhead finding a schedule
  - static
    - make scheduling decisions at off-line (also called pre-run-time)
    - generates a dispatching table for run-time dispatcher at compile time
    - needs complete knowledge of the task before compiling
    - small run-time overhead

  - preemptive
    - currently executing task may be interrupted (preempted) by higher priority processes
    - preempted process continues later at the same state
    - overhead of contexts switching
  - non-preemptive
    - running tasks will be allowed to finish its time-slot (higher priority processes must wait)
    - reasonable for short tasks like sending a packet (used by disk and network cards)
    - less frequent switches

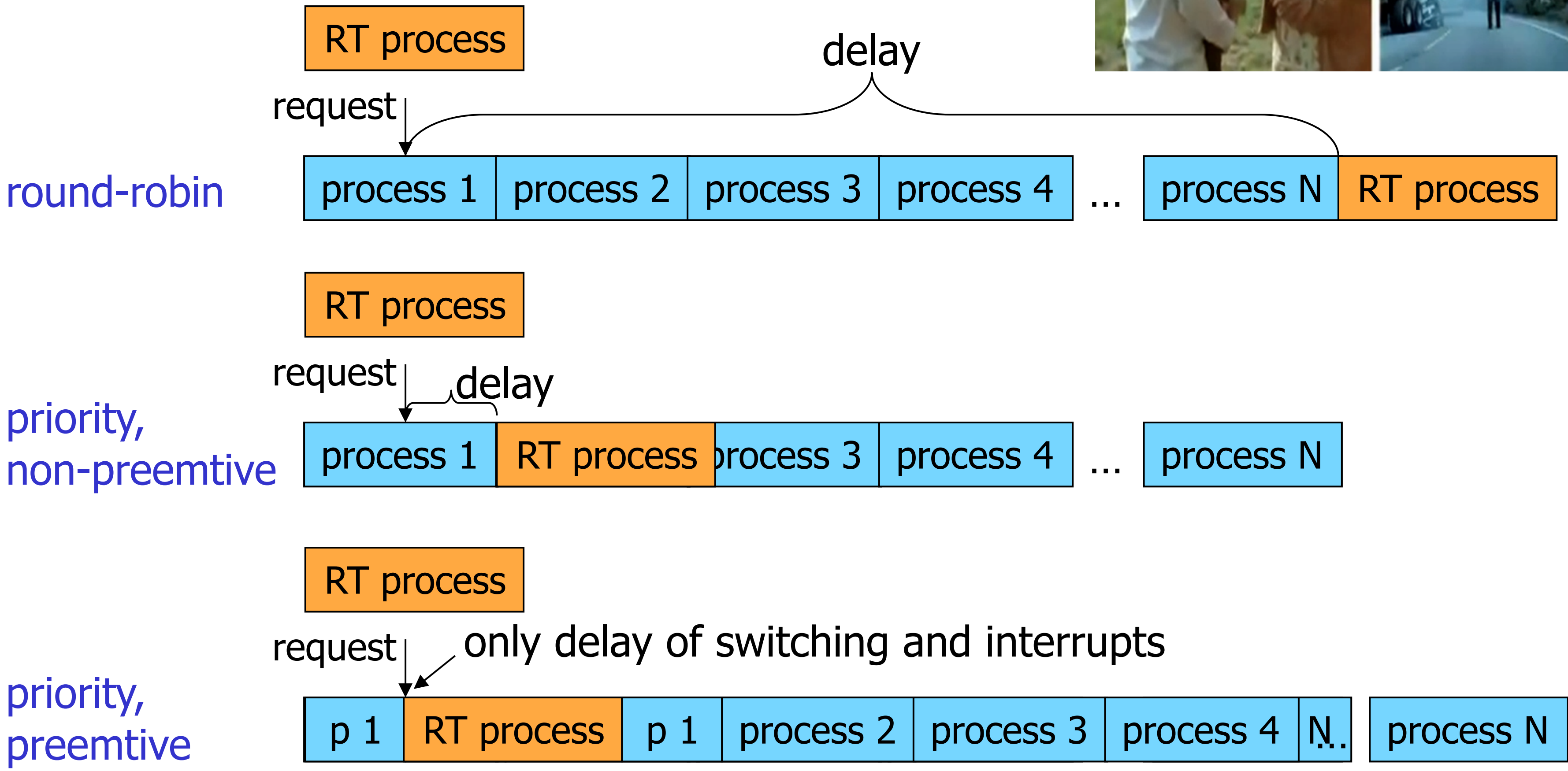
Preemption

- Tasks waits for processing
- Scheduler assigns priorities
- Task with highest priority will be scheduled first
- Preempt current execution if
  - a higher priority (more urgent) task arrives
  - timeslice is consumed
  - …
- Real-time and best effort priorities
  - real-time processes have higher priority (if exists, they will run)
- To kinds of preemption:
  - preemption points
    - predictable overhead
    - simplified scheduler accounting
  - immediate preemption
    - needed for hard real-time systems
    - needs special timers and fast interrupt and context switch handling



Preemption

- RT vs NRT example:



Many Algorithms Exist

- First In First Out (FIFO)
- Round-Robin (RR)
- Shortest Job First
- Shortest Time to Completion First
- Shortest Remaining Time to Completion First (a.k.a. Shortest Remaining Time First)
- Lottery
- Fair Queuing
- …

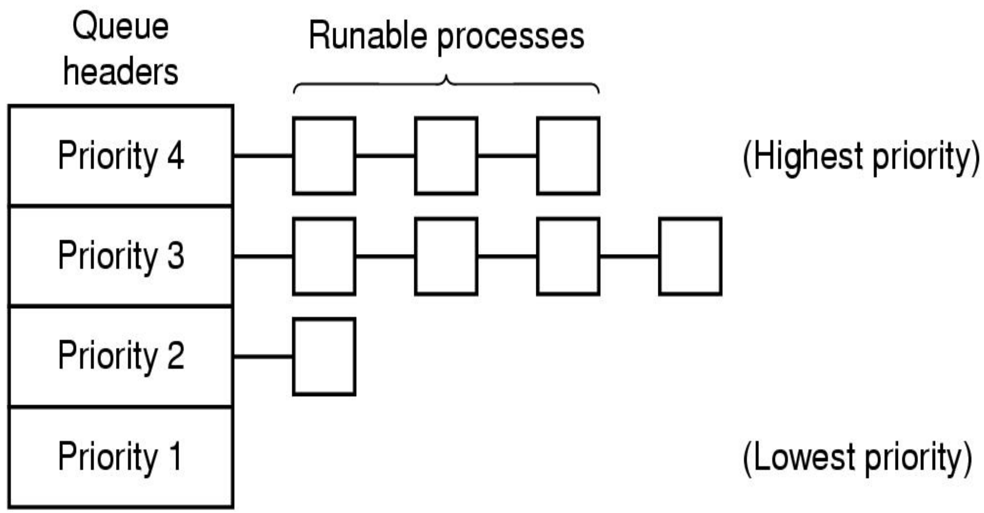
- Earliest Deadline First (EDF)
- Rate Monotonic (RM)
- …

- Most systems use some kind of *priority scheduling*

Priority Scheduling

- Assign each process a priority
- Run the process with highest priority in the ready queue first

- Multiple queues

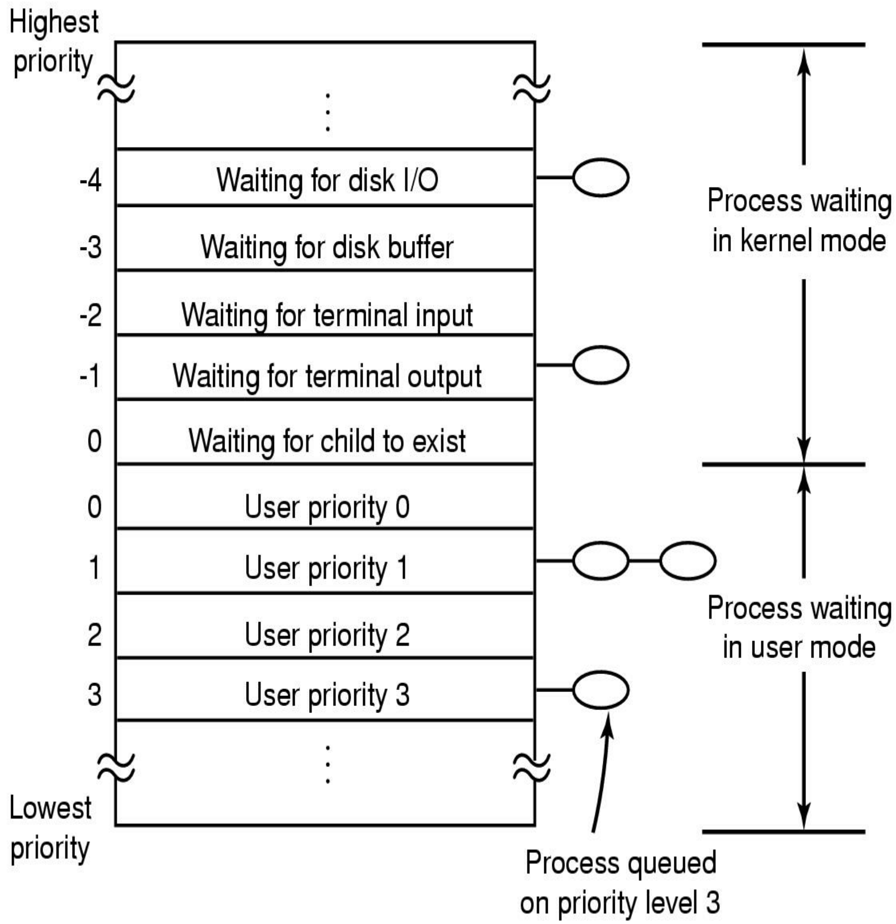


- Advantage
  - (Fairness)
  - Different priorities according to importance

- Disadvantage
  - Users can hit keyboard frequently
  - Starvation: so maybe use dynamic priorities?

Traditional scheduling in UNIX

- Many versions
- User processes have positive priorities, kernel negative
- Schedule lowest priority first
- If a process uses the whole time slice, it is put back at the end of the queue (RR)
- Each second the priorities are recalculated:
  priority =
  CPU_usage (average #ticks)
  + nice (± 20)
  + base (priority of last corresponding kernel process)



Scheduling in Windows 2000, XP, …

- Preemptive kernel
- Schedules threads individually

- Time slices given in quantums
  - 3 quantums = 1 clock interval (length of interval may vary)
  - defaults:
    - Win2000 server: 36 quantums
    - Win2000 workstation: 6 quantums (professional)
  - may manually be increased between threads (1x, 2x, 4x, 6x)
  - foreground quantum boost (add 0x, 1x, 2x):
    an active window can get longer time slices (assumed need for fast response)

Scheduling in Windows 2000, XP, …

- 32 priority levels:
  Round Robin (RR) within each level
- Interactive and throughput-oriented:
  - "Real time" – 16 system levels
    - fixed priority
    - may run forever
  - Variable – 15 user levels
    - priority may change:
      thread priority = process priority ± 2
    - uses much → drops
    - user interactions, I/O completions → increase
  - Idle/zero-page thread – 1 system level
    - runs whenever there are no other processes to run
    - clears memory pages for memory manager

Real Time (system thread)

31
30
...
17
16

Variable (user thread)

15
14
...
2
1

Idle (system thread)

0

Scheduling in Windows 8 (…server 2008, 7)

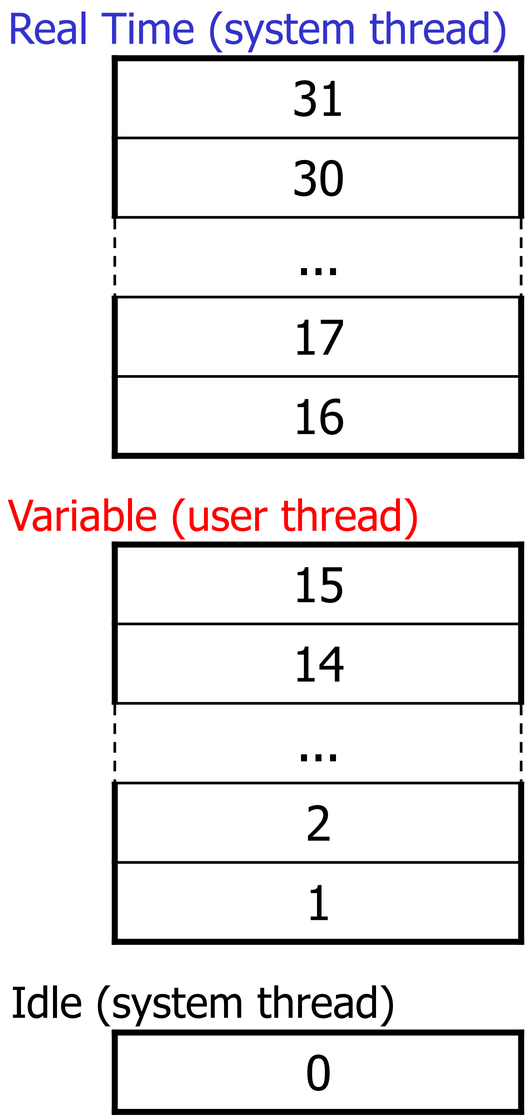
http://msdn.microsoft.com/en-us/library/windows/desktop/ms681917(v=vs.85).aspx

- Still 32 priority levels, with 6 classes - RR within each:
  - REALTIME_PRIORITY_CLASS
  - HIGH_PRIORITY_CLASS
  - ABOVE_NORMAL_PRIORITY_CLASS
  - ***NORMAL_PRIORITY_CLASS*** (default)
  - BELOW_NORMAL_PRIORITY_CLASS
  - IDLE_PRIORITY_CLASS

  ➥ each class has 7 thread priorities levels with different base priorities

- Dynamic priority (can be disabled):
  - \+ foreground
  - \+ window receives input (mouse, keyboard, timers, …)
  - \+ unblocks
  - – if increased, drop by one level every timeslice until back to default

- Support for user mode scheduling (UMS)
  - each application may schedule own threads
  - application must implement a scheduler component



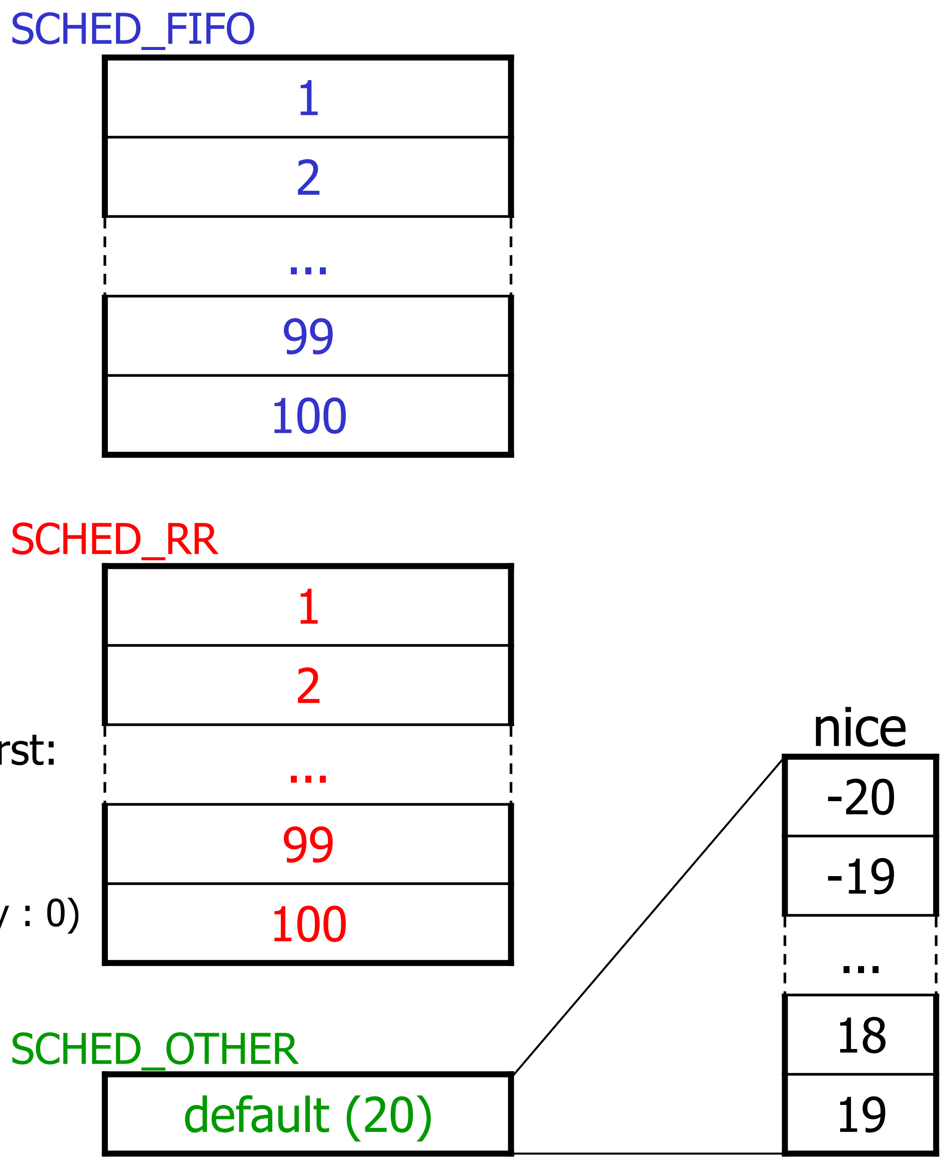
Scheduling in Linux

- Preemptive kernel
- Threads and processes used to be equal, but Linux uses (from 2.6) thread scheduling

- SCHED_FIFO
  - may run forever, no timeslices
  - may use it's own scheduling algorithm
- SCHED_RR
  - each priority in RR
  - timeslices of 10 ms (quantums)
- SCHED_OTHER
  - ordinary user processes
  - uses "nice"-values: 1≤ priority≤40
  - timeslices of 10 ms (quantums)

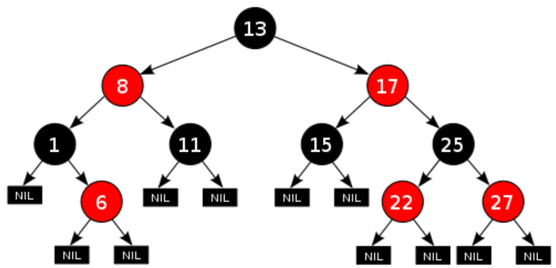
- Threads with highest *goodness* are selected first:
  - realtime (FIFO and RR):
    goodness = 1000 + priority
  - timesharing (OTHER):
    goodness = (quantum > 0 ? quantum + priority : 0)

- *Quantums* are reset when no *ready* process has quantums left (end of *epoch*):
  quantum = (quantum/2) + priority



Scheduling in Linux

- The current kernels (v.2.6.23+) use the ***Completely Fair Scheduler*** *(CFS)*
  - addresses unfairness in desktop and server workloads
  - uses ns granularity, does not rely on jiffies or HZ details
  - uses an extensible hierarchical scheduling classes
    - SCHED_NORMAL – the CFS desktop scheduler – replace SCHED_OTHER
    - SCHED_BATCH – similar to SCHED_OTHER, but assumes CPU intensive workloads
    - SCHED_RR and SCHED_FIFO (SCHED_RT)
      - use 100 priorities
  - no run-queues, a red-black tree-based timeline of future tasks based on virtual runtime



  - does not directly use priorities, but instead uses them as a decay factor for the time a task is permitted to execute

When to Invoke the Scheduler?

- Process creation 

- Process termination

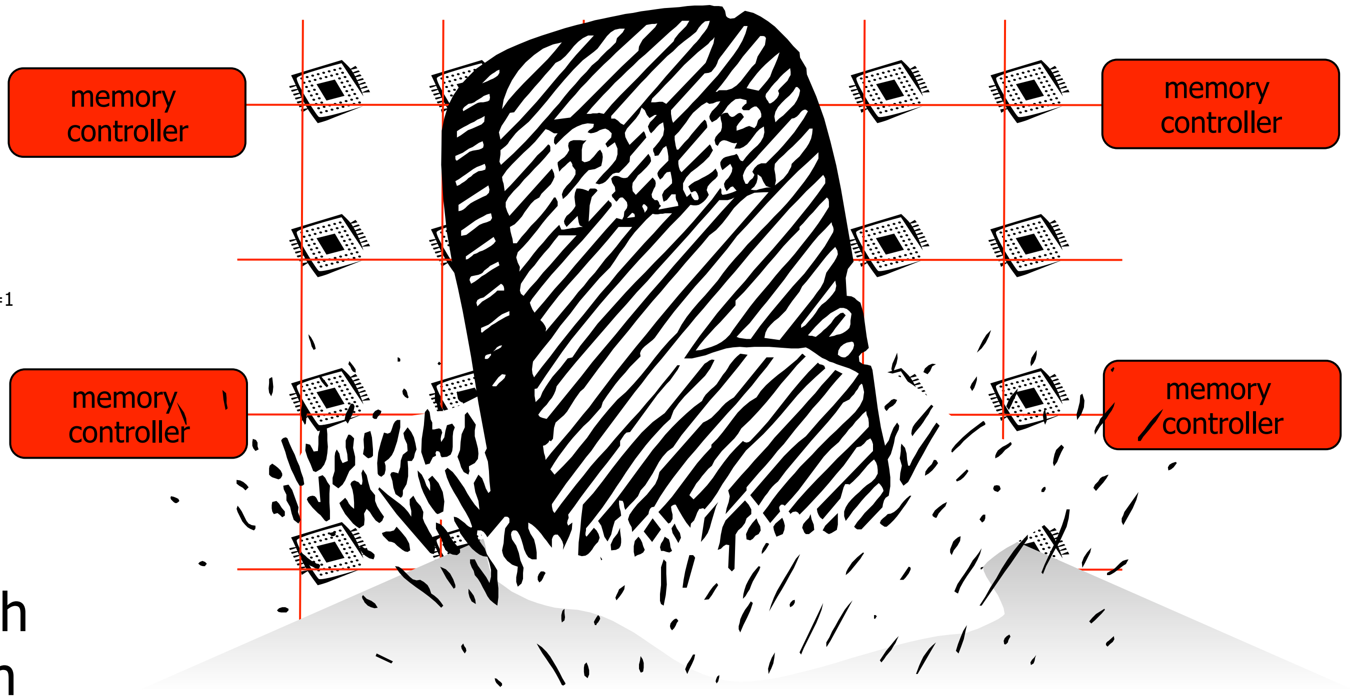
- Process blocks 

- Interrupts occur

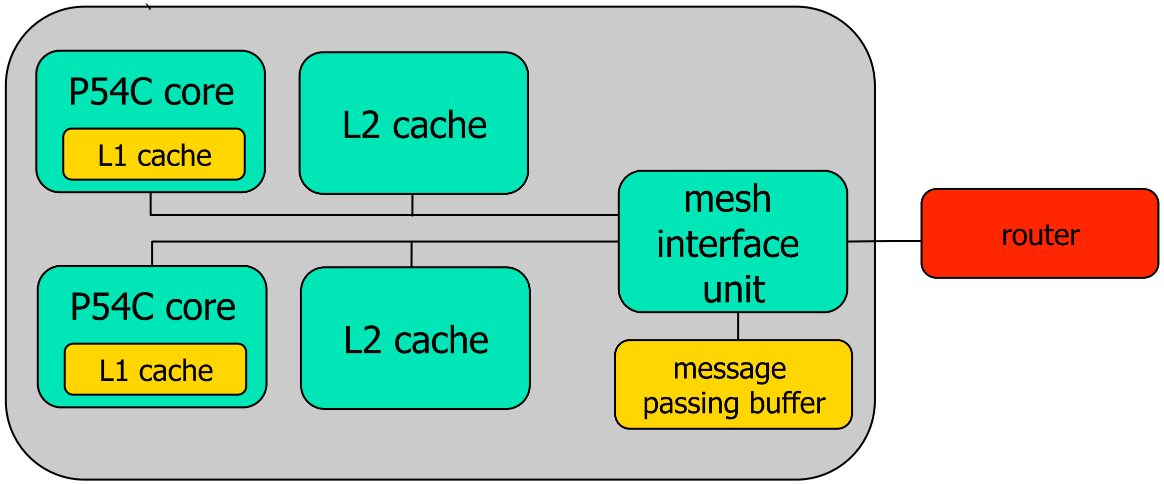
- Clock interrupts in the case of preemptive systems 

Future Chips: Something to think about!?

- Future Chips: Intel's Single-chip Cloud Computer (SCC)
  http://techresearch.intel.com/ProjectDetails.aspx?Id=1

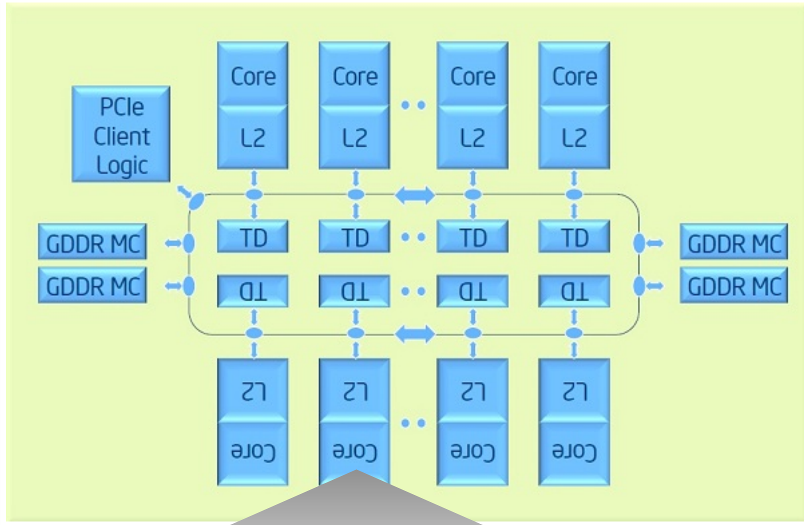


- What does introduction of such processors mean in terms of scheduling?
  - many cores
  - different memory access latencies
  - different connectivity
  - …

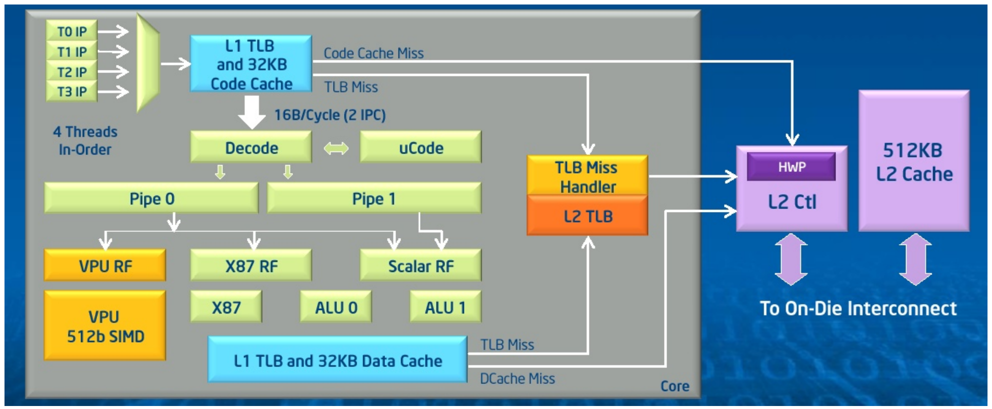


Future Chips: Something to think about!?

- Future Chips: Intel's Xeon Phi
  - up to 61 cores
  - 8 memory controllers
  - fully coherent L2 caches
  - High Performance On-Die Bidirectional Interconnect
  - …



- What does such processors mean in terms of scheduling?
  - many cores
  - different memory access latencies
  - different connectivity
  - …



Summary

- Processes are programs under execution
- Scheduling performance criteria and goals are dependent on environment
- The right timeslice can improve overall utilization
- There exists several different algorithms targeted for various systems
- Traditional OSes like Windows, UniX, Linux, ... usually use a priority-based algorithm