CPU Scheduling

Operating Systems
Wenbo Shen

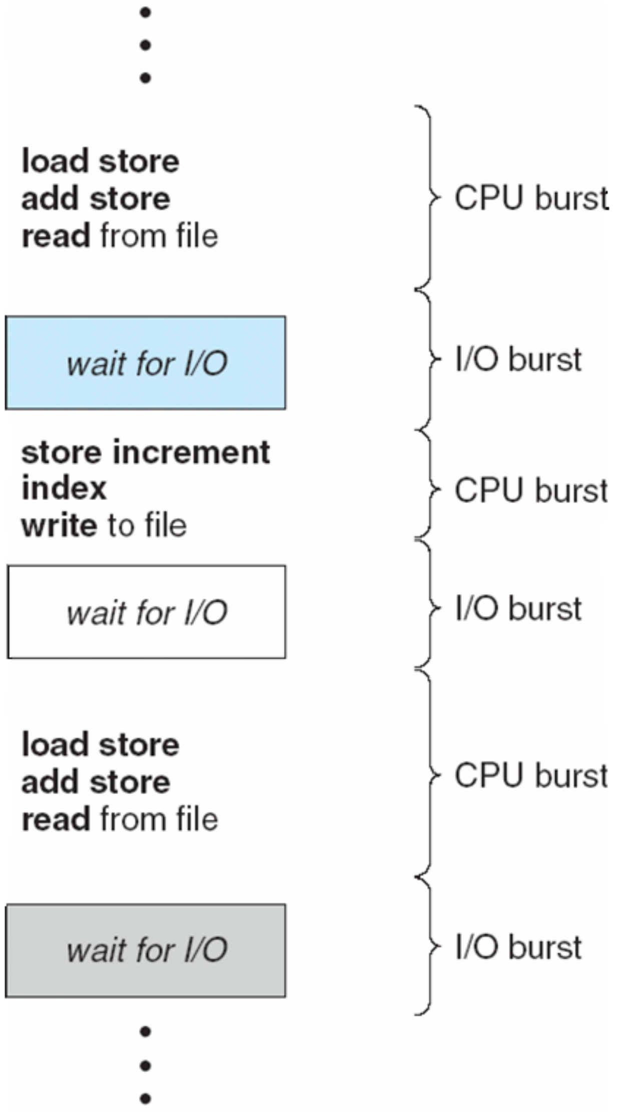
Some slides are from OSC and Henri Casanova@Hawaii

CPU Scheduling

- Definition
  - The decisions made by the OS to figure out which ready processes/threads should run and for how long
- Necessary in multi-programming environments
- CPU Scheduling is important for system performance and productivity
  - Maximizes CPU utilization so that it's never idle
- The policy is the scheduling strategy
- The mechanism is the dispatcher
  - A component of the OS that's used to switch between processes
    - That in turn uses the context switch mechanism
  - Must be lightning fast for time-sharing (dispatcher latency)

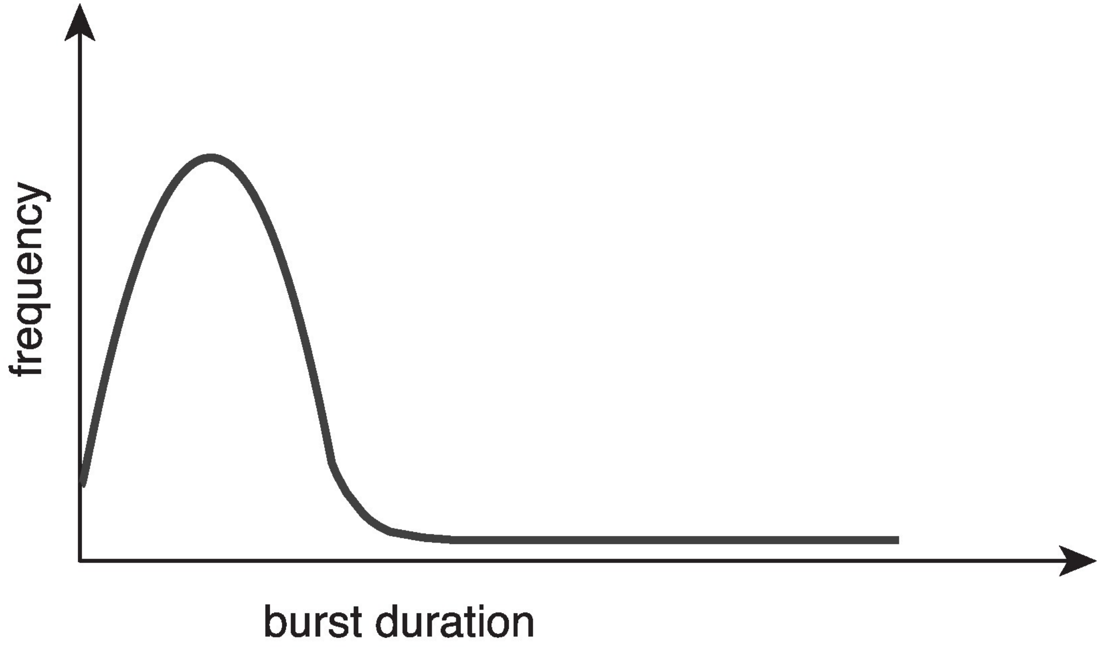
CPU-I/O Burst Cycle

- Most processes alternate between CPU and I/O activities
- One talks of a sequence of **bursts**
  - Starting and ending with a CPU burst
- I/O-bound process
  - Mostly waiting for I/O
  - Many short CPU bursts
  - e.g., /bin/cp
- CPU-bound process
  - Mostly using the CPU
  - Very short I/O bursts if any
  - e.g., enhancing an image
- The fact that processes are diverse makes CPU scheduling difficult



Histogram of CPU-burst Times

- Large number of short bursts
- Small number of longer bursts



The CPU Scheduler

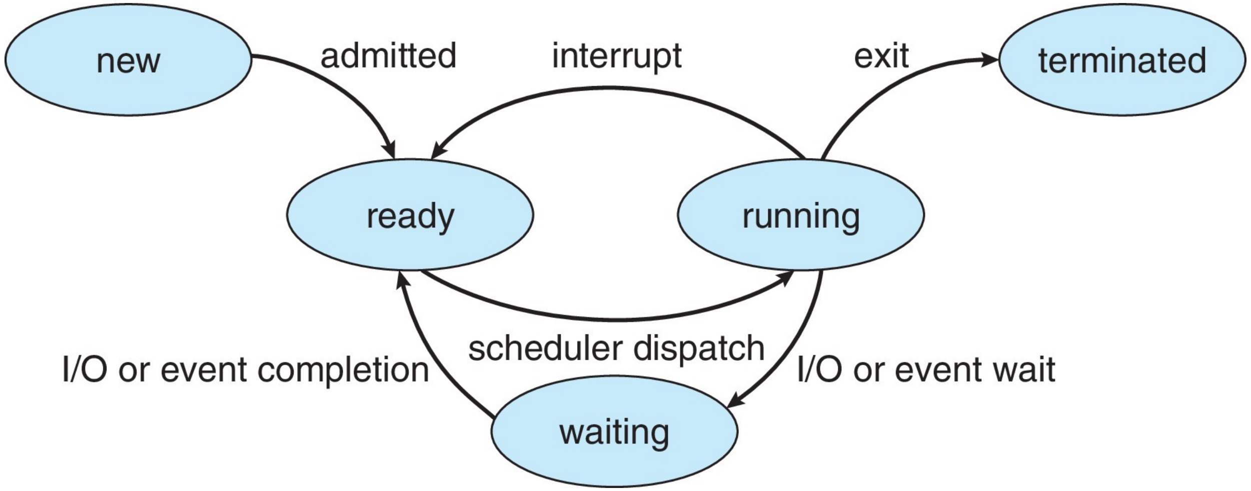
- Whenever the CPU becomes idle, a ready process must be selected for execution
  - The OS keeps track of process states
- Non-preemptive scheduling: a process holds the CPU until it is willing to give it up
  - Also called “cooperative” scheduling
- Preemptive scheduling: a process can be preempted even though it could have happily continued executing
  - e.g., after some “you’ve had enough” timer expires

Scheduling Decision Points

- Scheduling decisions can occur when:
  - **#1**: A process goes from RUNNING to WAITING
    - e.g., waiting for I/O to complete
  - **#2**: A process goes from RUNNING to READY
    - e.g., when an interrupt occurs (such as a timer going off)
  - **#3**: A process goes from WAITING to READY
    - e.g., an I/O operation has completed
  - **#4**: A process goes from RUNNING to TERMINATED
  - **#5**: A process goes from NEW to READY
  - **#6**: A process goes from READY to WAITING

Process State

- As a process executes, it changes **state**
  - **New**: The process is being created
  - **Running**: Instructions are being executed
  - **Waiting**: The process is waiting for some events to occur
  - **Ready**: The process is waiting to be assigned to a processor
  - **Terminated**: The process has finished execution



Scheduling Decision Points

- Scheduling decisions can occur when:
  - **#1**: A process goes from RUNNING to WAITING
    - e.g., waiting for I/O to complete
  - **#2**: A process goes from RUNNING to READY
    - e.g., when an interrupt occurs (such as a timer going off)
  - **#3**: A process goes from WAITING to READY
    - e.g., an I/O operation has completed
  - **#4**: A process goes from RUNNING to TERMINATED
  - **#5**: A process goes from NEW to READY
- Non-preemptive scheduling: Except #2
  - Scheduling happens when a process gives up CPU voluntarily
  - From running to ”CANNOT RUN ANYMORE”
  - Windows 3.x, Mac OS 9
- Preemptive scheduling: #1, #2, #3, #4, #5
  - Windows 95 and later, Max OS X, Linux

Scheduling Decision Points

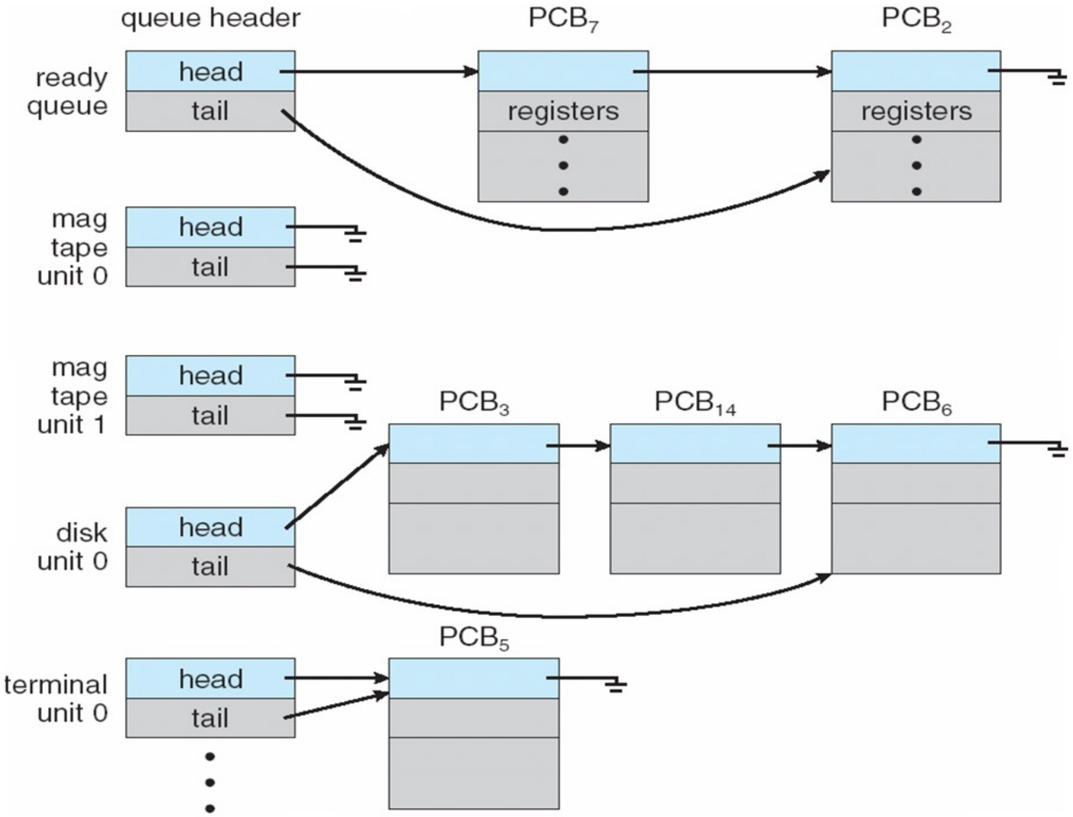
- Preemptive scheduling is good
  - No need to have processes willingly give up the CPU
  - The OS remains in control
- Preemptive scheduling is complex
  - Opens up many thorny issues having to do with process synchronization
    - Consider access to shared data
    - Consider preemption while in kernel mode
    - Consider interrupts occurring during crucial OS activities

Scheduling Objectives

- Finding the right objective is an open question
- There are many conflicting goals that one could attempt to achieve
  - Maximize CPU Utilization
    - Fraction of the time the CPU isn’t idle
  - Maximize Throughput
    - Amount of “useful work” done per time unit
  - Minimize Turnaround Time
    - Time from process creation to process completion
  - Minimize Waiting Time
    - Amount of time a process spends in the READY state
  - Minimize Response Time
    - Time from process creation until the “first response” is received
- Question: should we optimize averages, maxima, variances?
  - Again, a lot of theory here...

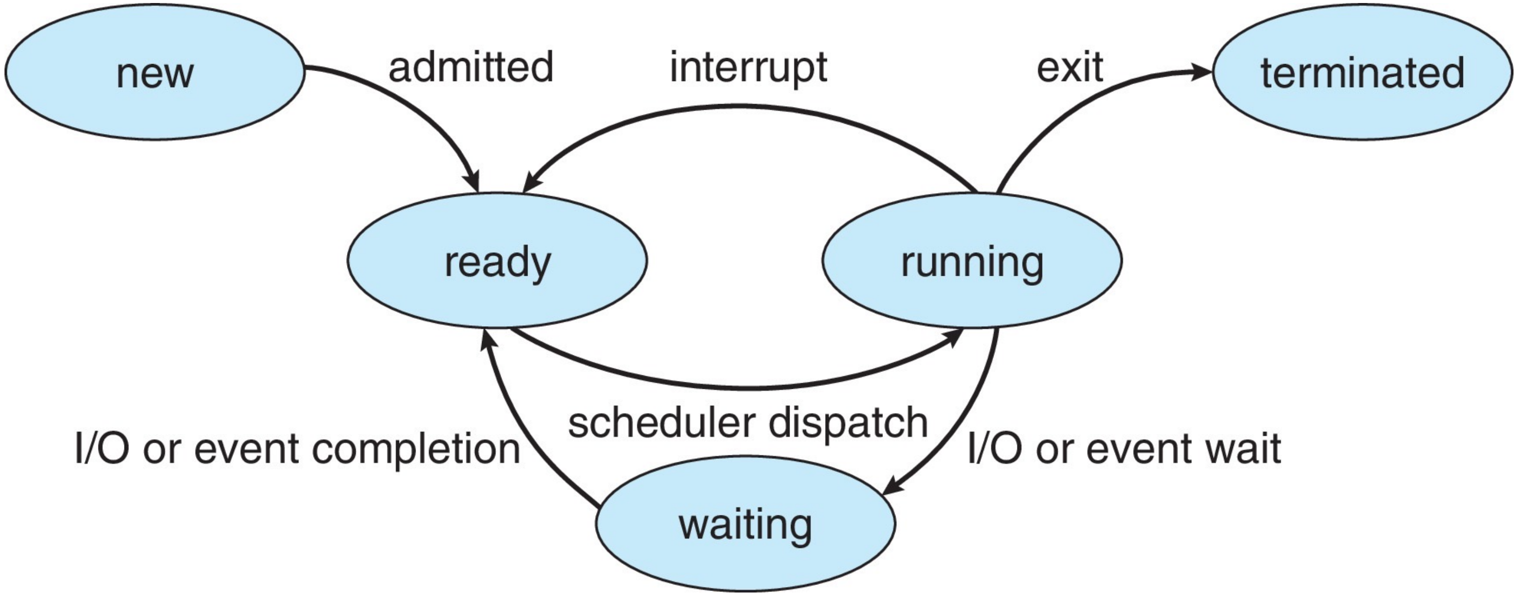
Scheduling Queues

- The Kernel maintains Queues in which processes are placed
  - Linked lists of pointers to PCB data structures
  - list_head struct in Linux
- The Ready Queue contains processes that are in the READY state
- Device Queues contain processes waiting for particular devices

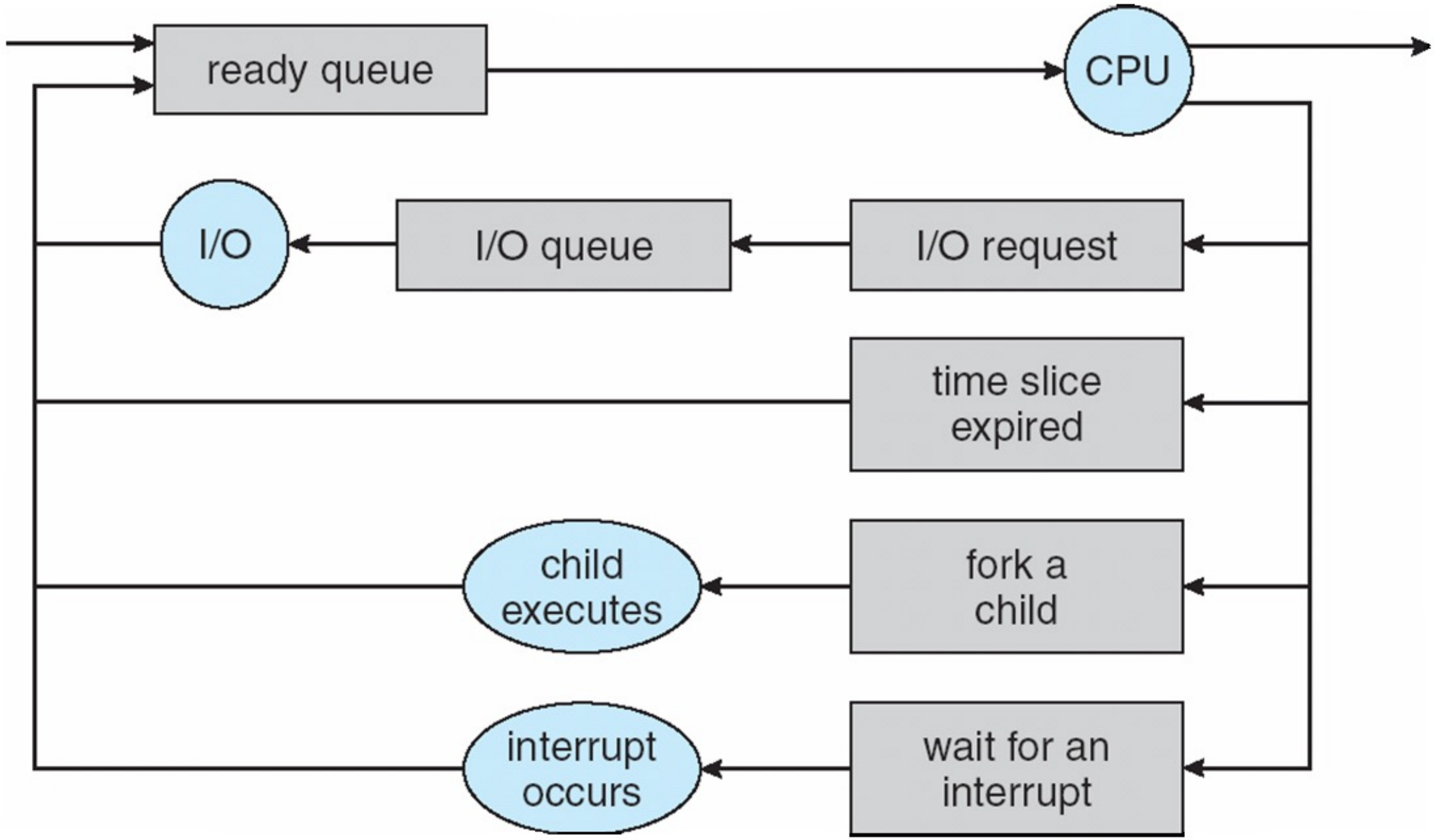


Process State

- As a process executes, it changes **state**
  - **New**: The process is being created
  - **Running**: Instructions are being executed
  - **Waiting**: The process is waiting for some event to occur
  - **Ready**: The process is waiting to be assigned to a processor
  - **Terminated**: The process has finished execution

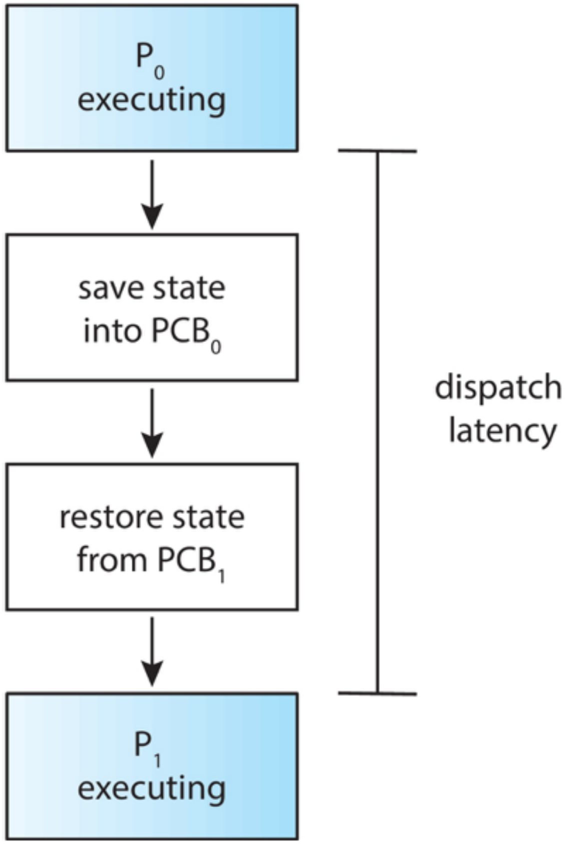


Scheduling and Queues



Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the scheduler; this involves:
  - switching to kernel mode
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another to run



Scheduling Algorithms

- Now that we understand the mechanisms (queues, dispatcher, context switching), the question is: what’s a good policy?
  - i.e., what (good) algorithms should be implemented to decide on which process runs?
- Defining “good” is very difficult, due to the wide range of conflicting goals
  - e.g., having many context switches is bad for throughput
    - No useful work is done during a context switch
  - e.g., having few context switches is bad for response time
    - Android sluggish problem - Linux kernel prefers throughput rather than responsiveness
- One thing is certain: the algorithms cannot be overly complicated so that they can be fast

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

Scheduling Algorithms

- First-Come, First-Served Scheduling
- Shortest-Job-First Scheduling
- Round-Robin Scheduling
- Priority Scheduling
- Multilevel Queue Scheduling
- Multilevel Feedback Queue Scheduling

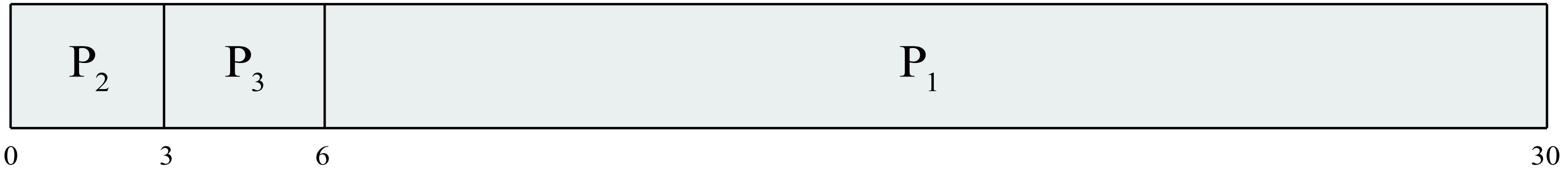
First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order at the same time:

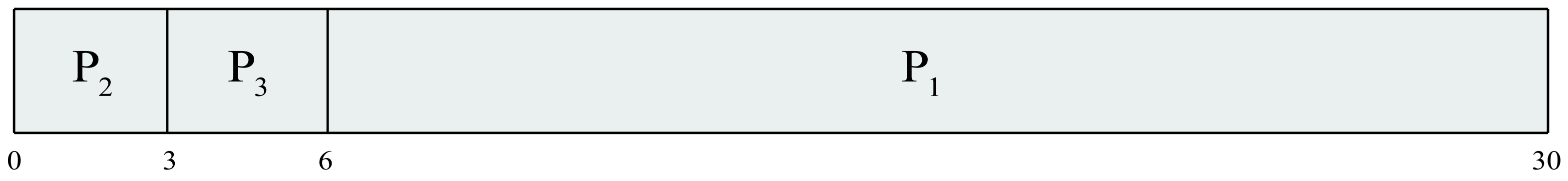
  P_2 , P_3 , P_1

- The Gantt chart for the schedule is:



First-Come, First-Served (FCFS) Scheduling

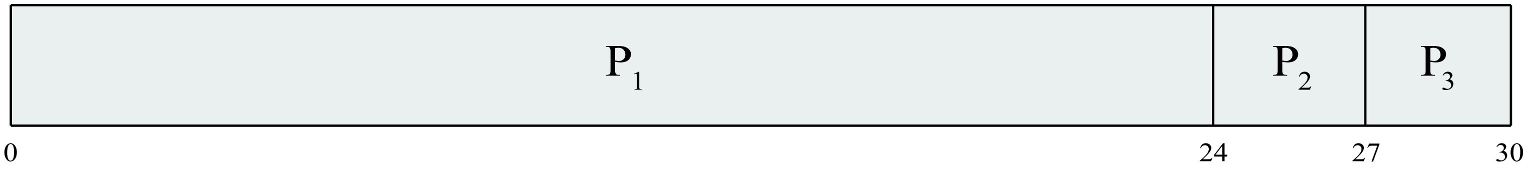
- The Gantt chart for the schedule is:



- Waiting time = start time – arrival time
  - $P_1 = 6; P_2 = 0; P_3 = 3$
  - Average waiting time: (6 + 0 + 3)/3 = 3
- Turnaround time = finish time – arrival time
  - $P_1 = 30; P_2 = 3; P_3 = 6$
  - Average turnaround time: (30 + 3 + 6)/3 = 13

FCFS Scheduling (Cont.)

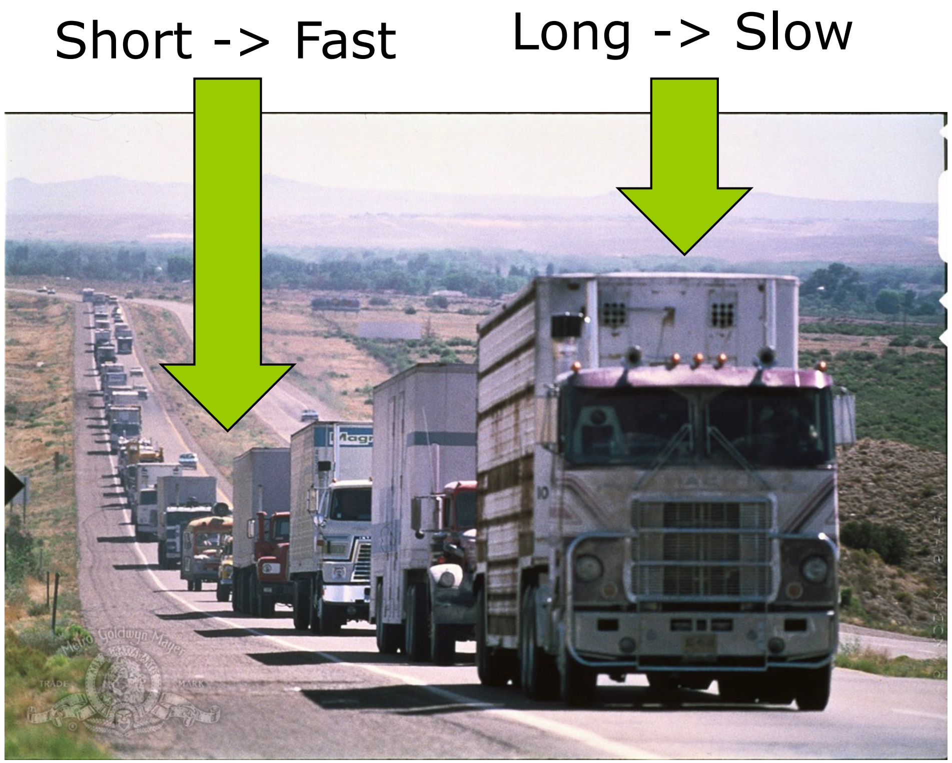
- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
The Gantt Chart for the schedule is:



- Average waiting time: (0 + 24 + 27)/3 = 17
  - Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average turnaround time: (24 + 27 + 30)/3 = 27
- Much worse than previous case
- **Convoy effect** - short process behind long process

FCFS Scheduling (Cont.)

- Convoy effect - short process behind long process
  - Long jobs slow down the whole system


Short -> Fast

Long -> Slow

Shortest-Job-First (SJF) Scheduling

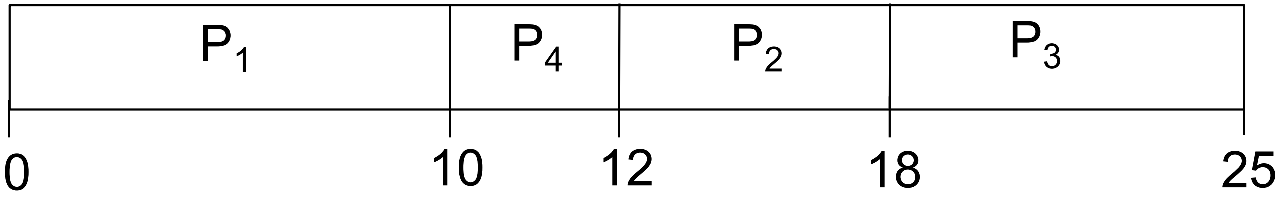
- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes

Shortest Job First (SJF)

- "Shortest-next-CPU-burst" algorithm
- Non-preemptive example:

Process	Arrival Time	Burst Time
P_1	0.0	10
P_2	2.0	6
P_3	4.0	7
P_4	5.0	2

- Gantt Chart:



P_1 P_4 P_2 P_3

0 10 12 18 25

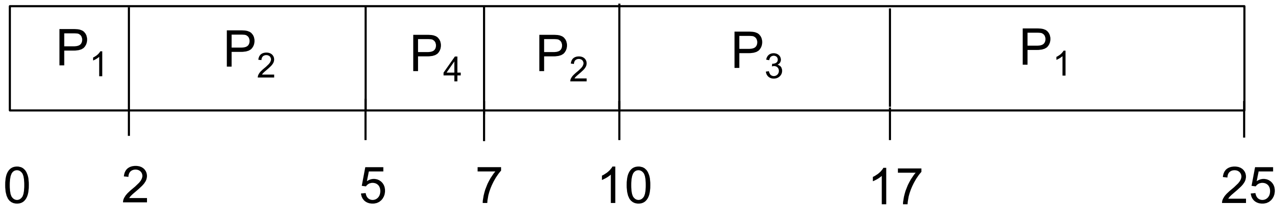
- Average waiting time = 7.25
  - P1=0, P2=10, P3=14, P4=5
- Average turnaround time = 13.5
  - P1=10, P2=16, P3=21, P4=7

Shortest Job First (SJF)

- "Shortest-next-CPU-burst" algorithm
- Preemptive - **shortest-remaining-time-first,** example:
-

Process	Arrival Time	Burst Time
P_1	0.0	10
P_2	2.0	6
P_3	4.0	7
P_4	5.0	2

- Gantt Chart:



P_1 | P_2 | P_4 | P_2 | P_3 | P_1

0 2 5 7 10 17 25

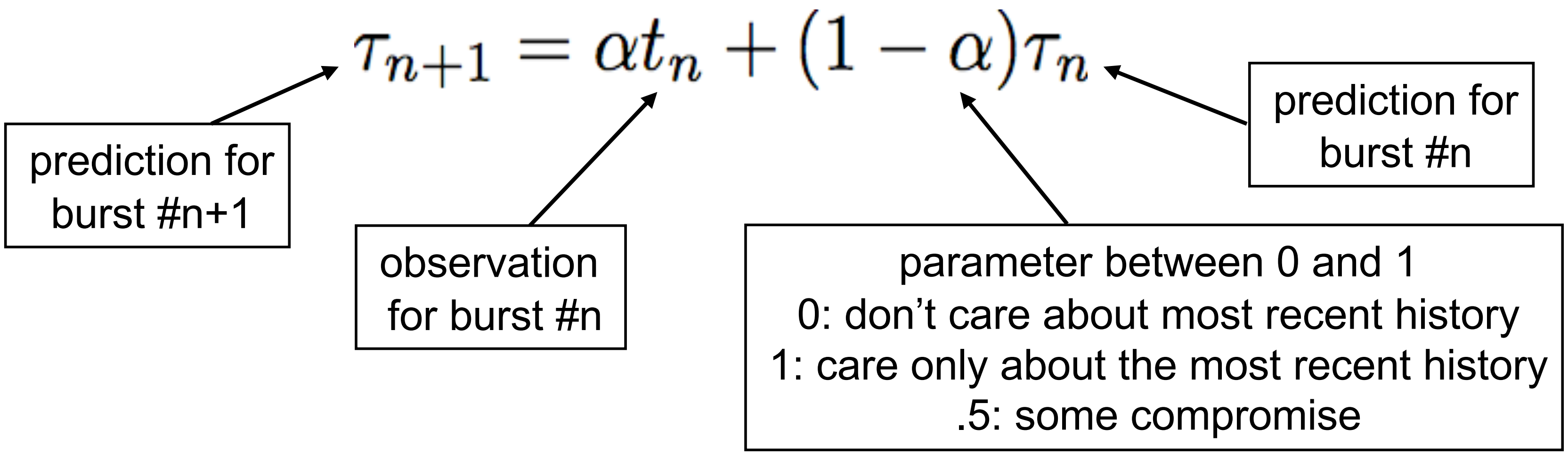
- Average waiting time = 5.75
  - P1=15, P2=2, P3=6, P4=0
- Average turnaround time = 12
  - P1=25, P2=8, P3=13, P4=2

Shortest Job First (SJF)

- Question: How good is a scheduling algorithm?
- In some cases, one can prove optimality for a given metric
- There is a HUGE theoretical literature on the relative merit of particular algorithms for particular metrics
- A known result is: SJF is provably optimal for average wait time
  - In the theoretical literature, called: SRPT (Shortest Remaining Processing Time)
  - Optimal with and without preemption
- Big Problem: How can we know the burst durations???
  - Perhaps doable for long-term scheduling, but known difficulties
    - e.g., rely on user-provided estimates???
  - This problem is typical of the disconnect between theory and practice
- Can we do any good prediction?

Predicting CPU burst durations

- One only knows the duration of a CPU burst once it's over
- Idea: predict future CPU bursts based on previous CPU bursts
- Exponential averaging of previously observed burst durations
  - Predict the future given the past
  - Give more weight to the recent past than the remote past



$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$$

prediction for burst #n+1

observation for burst #n

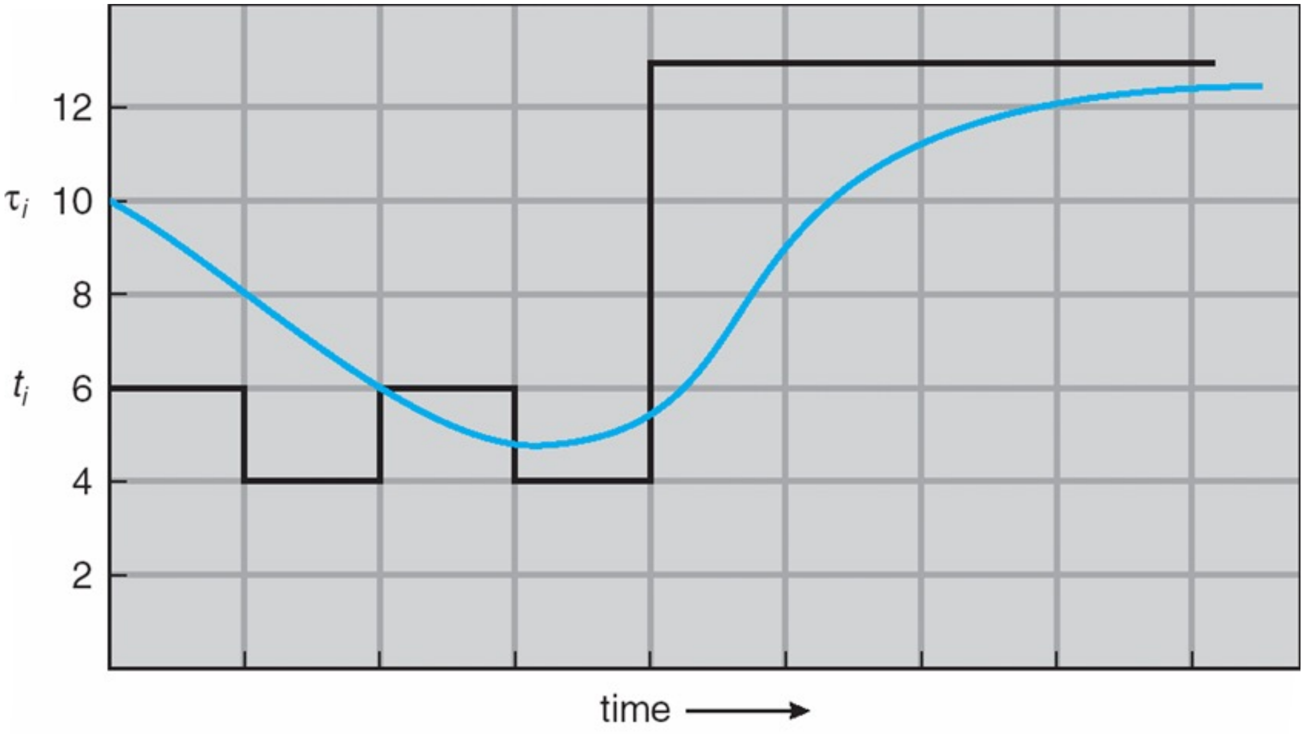
parameter between 0 and 1
0: don't care about most recent history
1: care only about the most recent history
.5: some compromise

prediction for burst #n

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + \cdots + (1-\alpha)^j \alpha t_{n-j} + \cdots + (1-\alpha)^{n+1}\tau_0$$

Exponential Averaging

$$\tau_0 = 10, \ \alpha = 0.5$$



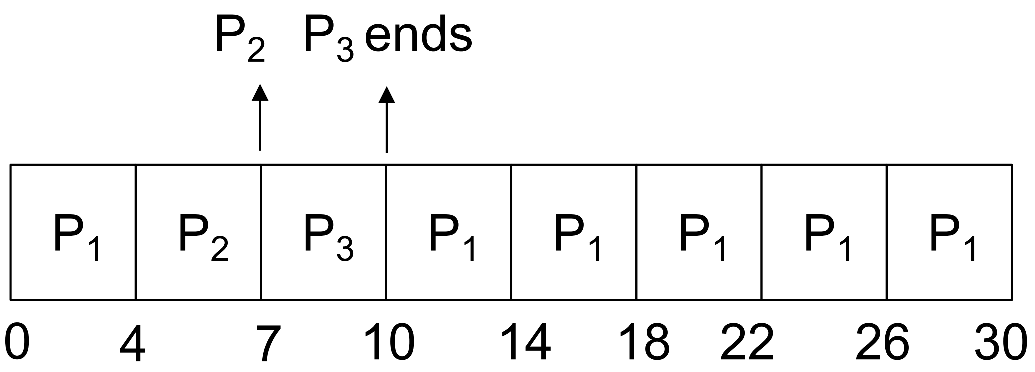
CPU burst (t_i)		6	4	6	4	13	13	13	. . .
"guess" (τ_i)	10	8	6	6	5	9	11	12	. . .

Round-Robin Scheduling

- RR Scheduling is preemptive and designed for time-sharing
- It defines a time quantum
  - A fixed interval of time (10-100ms)
- Unless a process is the only READY process, it never runs for longer than a time quantum before giving control to another ready process
  - It may run for less than the time quantum if its CPU burst is smaller than the time quantum
- Ready Queue is a FIFO
  - Whenever a process changes its state to READY it is placed at the end of the FIFO
- Scheduling:
  - Pick the first process from the ready queue
  - Set a timer to interrupt the process after 1 quantum
  - Dispatch the process

RR Scheduling Example

Process	Burst Time
P_1	24
P_2	3
P_3	3



P_2 P_3 ends

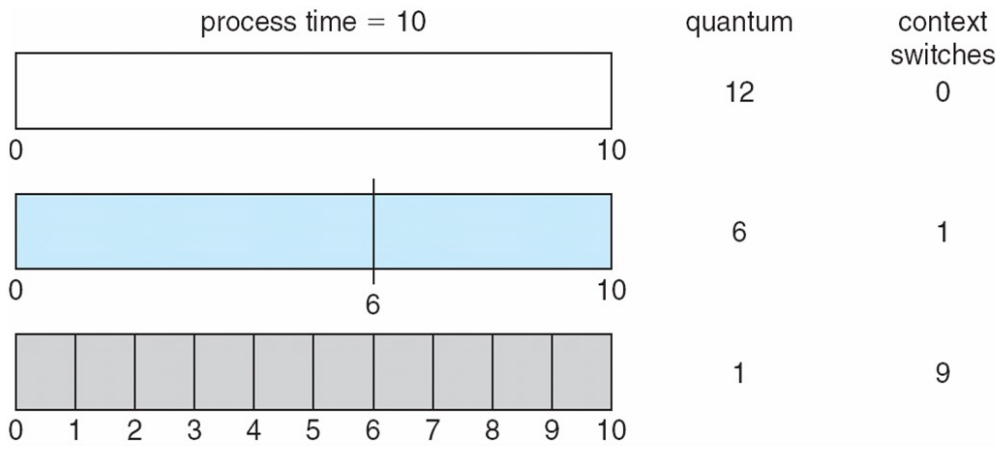
P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1

0 4 7 10 14 18 22 26 30

quantum = 4

- Typically, average waiting time worse than SJF, but better response time
  - No starvation, so better response time
  - And the wait time is bounded!
    - Know how long a process needs to wait
    - Average waiting time
      - P1=6, P2=4, P3=7

Picking the Right Quantum



quantum	context switches
12	0
6	1
1	9

- Trade-off:
  - Short quantum: great response/interactivity but high overhead
    - Hopefully not too high if the dispatcher is fast enough
  - Long quantum: poor response/interactivity, but low overhead
    - With a very long time quantum, RR Scheduling becomes FCFS Scheduling
- If context-switching time is 10% of time quantum, then the CPU spends >10% of its time doing context switches
- In practice, %CPU time spent on switching is very low
  - time quantum: 10ms to 100ms
  - context-switching time: 10μs

Priority Scheduling

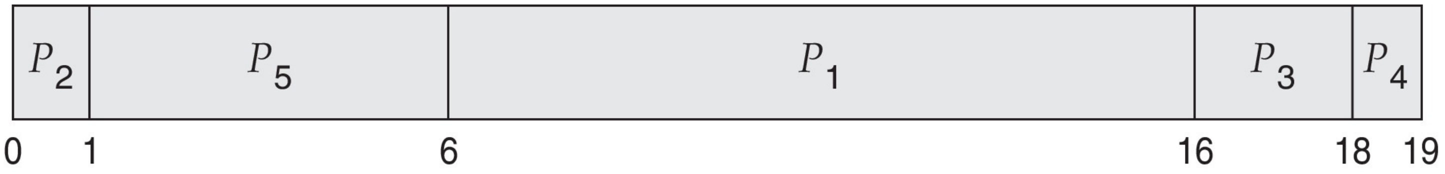
- Let us assume that we have jobs with various priorities
  - Priority: A number in some range (e.g., “0..9”)
  - No convention: low number can mean low or high priority
- Priorities can be internal:
  - e.g., in SJF it’s the predicted burst time, the number of open files
- Priorities can be external:
  - e.g., set by users to specify relative importance of jobs

- SJF is priority scheduling where priority is predicted next CPU burst time
  - Low number means high priority

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart



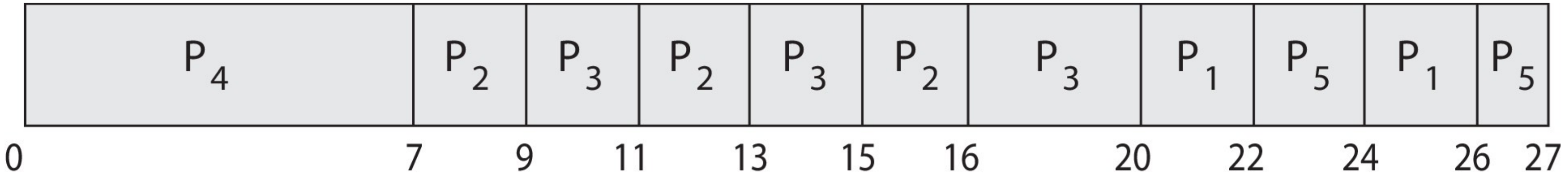
- Average waiting time
  - 8.2

Priority Scheduling w/ Round-Robin

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

- Run the process with the highest priority. Processes with the same priority run round-robin

- Gantt Chart with 2 ms time quantum



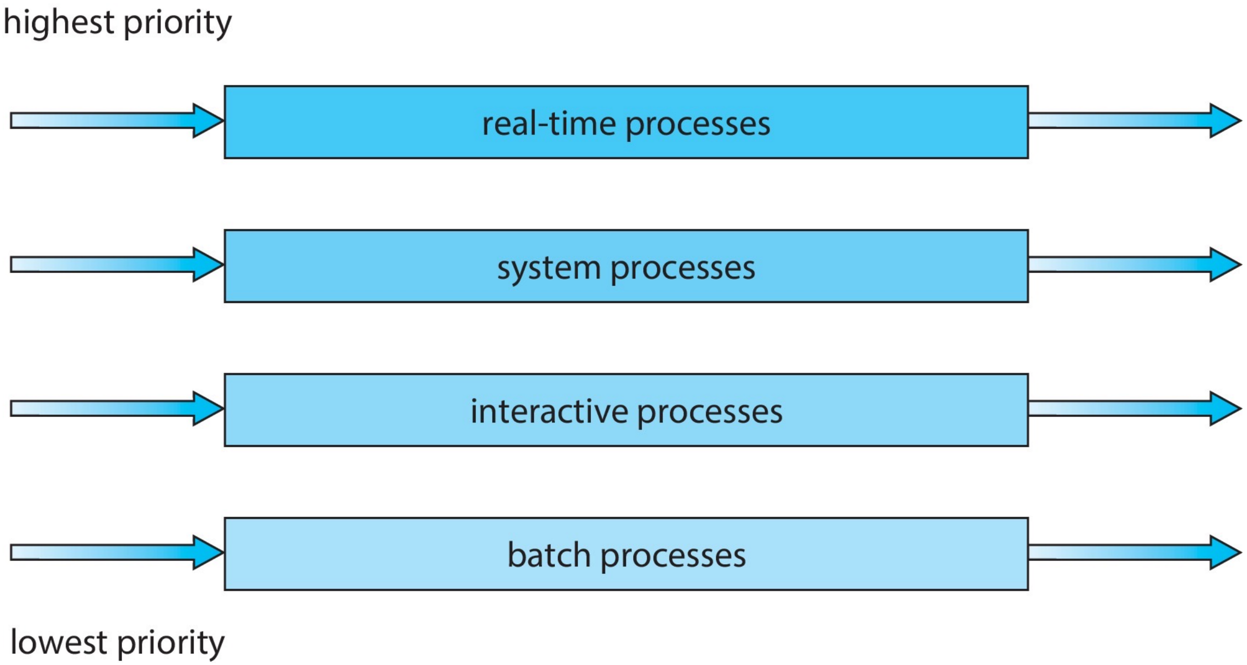
Priority Scheduling

- Simply implement the Ready Queue as a Priority Queue
- Like SJF, priority scheduling can be preemptive or non-preemptive
- See example in book, nothing difficult
- The problem: will a low-priority process ever run??
  - It could be constantly overtaken by higher-priority processes
  - It could be preempted by higher-priority processes
  - This is called starvation
  - Textbook anecdote/rumor: "When they shut down the IBM 7094 at MIT in 1973, they found a low-priority process that had been submitted in 1967 and had yet to run."
- A solution: Priority aging
  - Increase the priority of a process as it ages

Multilevel Queue Scheduling

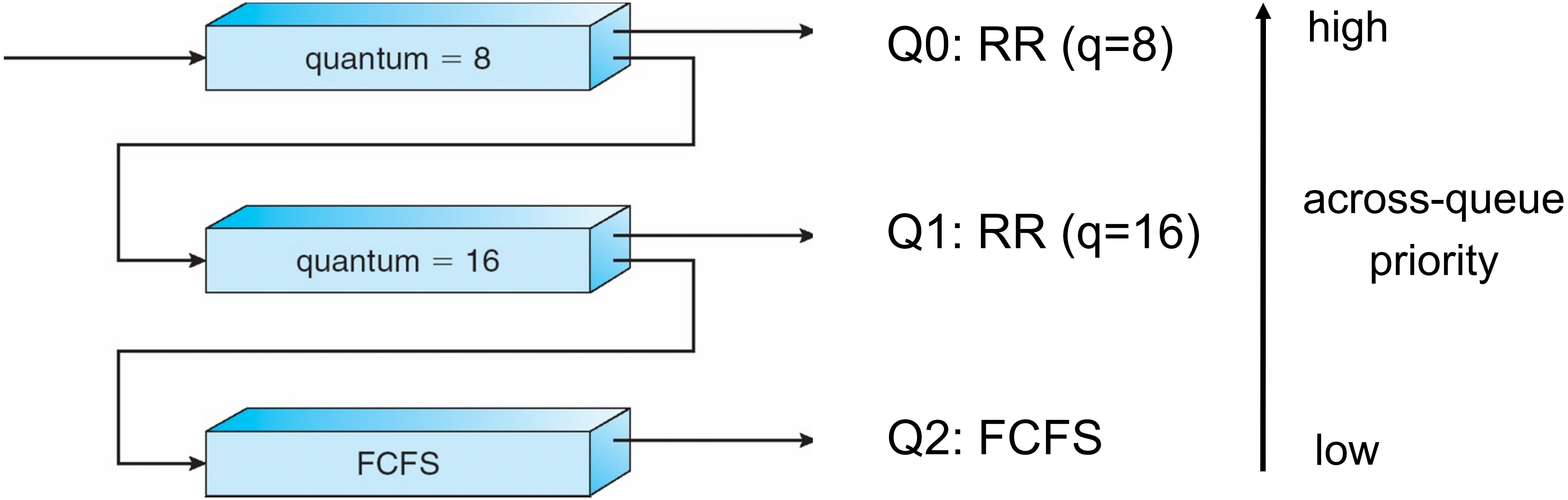
- The RR Scheduling scheme treats all processes equally
- In practice, one often wants to classify processes in groups, e.g., based on externally-defined process priorities
- Simple idea: use one ready queue per class of processes
  - e.g., if we support 10 priorities, we maintain 10 ready queues
- Scheduling within queues
  - Each queue has its own scheduling policy
  - e.g., High-priority could be RR, Low-priority could be FCFS
- Scheduling between the queues
  - Typically preemptive priority scheduling
    - A process can run only if all higher-priority queues are empty
  - Or time-slicing among queues
    - e.g., 80% to Queue #1 and 20% to Queue #2

Multi-Level Queue Example



Multilevel Feedback Queues

- Processes can move among the queues
  - If queues are defined based on internal process characteristics, it makes sense to move a process whose characteristics have changed
    - e.g., based on CPU burst length
  - It's also a good way to implement priority aging
- Let's look at the example in the textbook



Q0: RR (q=8)

Q1: RR (q=16)

Q2: FCFS

high

across-queue priority

low

Multilevel Feedback Queues

- This scheme implements a particular CPU scheduling “philosophy”
  - A new process arrives
  - It’s placed in Q0 and is, at some point, given a quantum of 8
  - If it doesn’t use it all, it’s likely a I/O-bound process and should be kept in the high-priority queue so that it is assured to get the CPU on the rare occasions that it needs it
  - If it does use it all, then it gets demoted to Q1 and, at some points, is given a quantum of 16
  - If it does use it all, then it’s likely a CPU-bound process and it gets demoted to Q2
  - At that point the process runs only when no non-CPU-intensive process needs the CPU
- Rationale: non-CPU-intensive jobs should really get the CPU quickly on the rare occasions they need them, because they could be interactive processes (this is all guesswork, of course)

Multilevel Feedback Queues

- The Multilevel Feedback Queues scheme is very general because highly configurable
  - Number of queues
  - Scheduling algorithm for each queue
  - Scheduling algorithm across queues
  - Method used to promote/demote a process
- However, what’s best for one system/workload may not be best for another
  - Systems configurable with tons of parameters always hold great promises but these promises are hard to achieve
- Also, it requires quite a bit of computation
  - We’ll see that (Linux) Kernel developers resort to cool hacks to speed it up

What’s a Good Scheduling Algorithm?

- Few analytical/theoretical results are available
  - Essentially, take two scheduling algorithms A and B, take a metric (e.g., wait time)
  - In rare cases you can show that an algorithm is optimal (e.g., SRPT for average wait time)
- Another option: Simulation
  - Test a million cases by producing Gantt Charts (not by hand)
  - Compare: A is better than B in 72% of the cases
- Finally: Implementation
  - Implement both A and B in the kernel (requires time!)
  - Use one for 10 hours, and the other for 10 hours for some benchmark workload
  - Compare: A is better than B because 12% more useful work was accomplished

Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as **process-contention scope** (**PCS**) since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope** (**SCS**) – competition among all threads in system

Pthread Scheduling

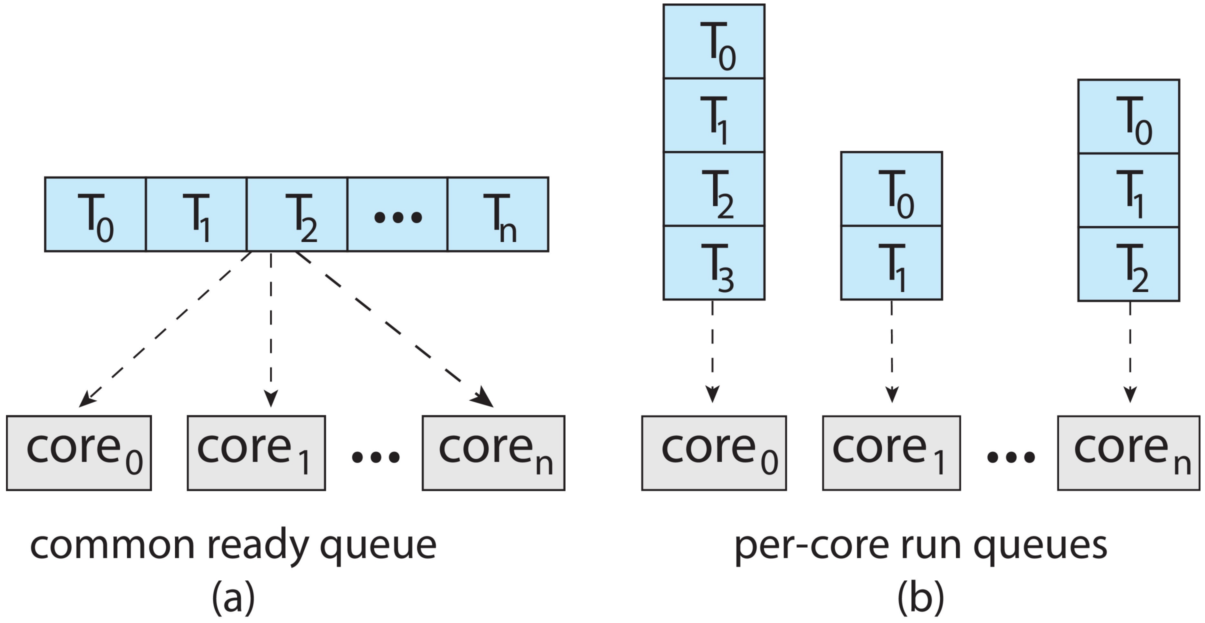
- API allows specifying either PCS or SCS during thread creation
  - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
  - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and macOS only allow PTHREAD_SCOPE_SYSTEM

Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available
- Multi-processor may be any one of the following architectures:
  - Multi-core CPUs
  - Multi-threaded cores

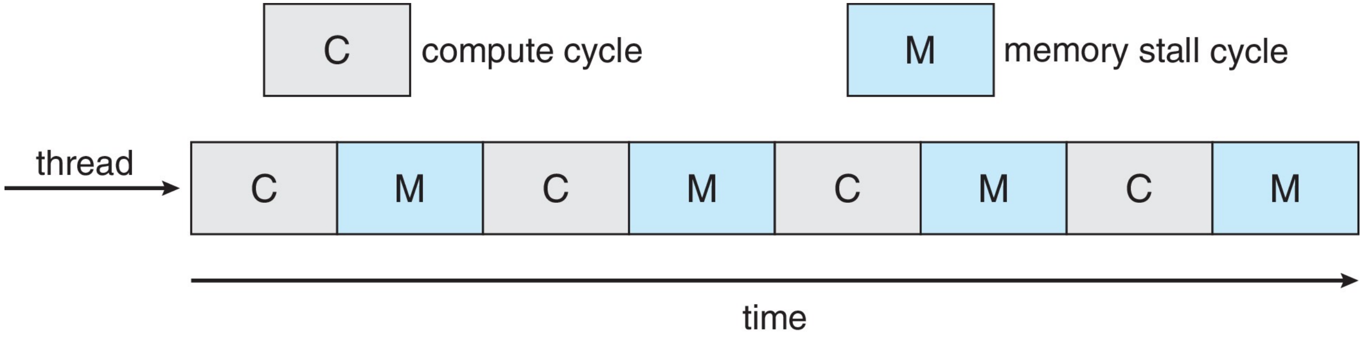
Multiple-Processor Scheduling

- Symmetric multiprocessing (SMP) is where each processor is self scheduling.
- All threads may be in a common ready queue (a)
- Each processor may have its own private queue of threads (b)



Multicore Processors

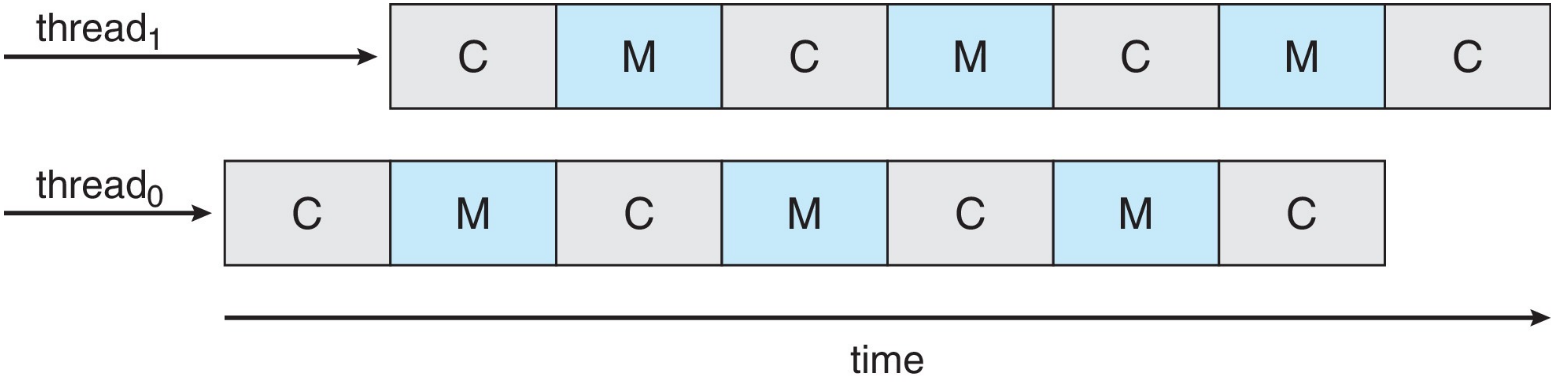
- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens



Multithreaded Multicore System

Each core has > 1 hardware threads.

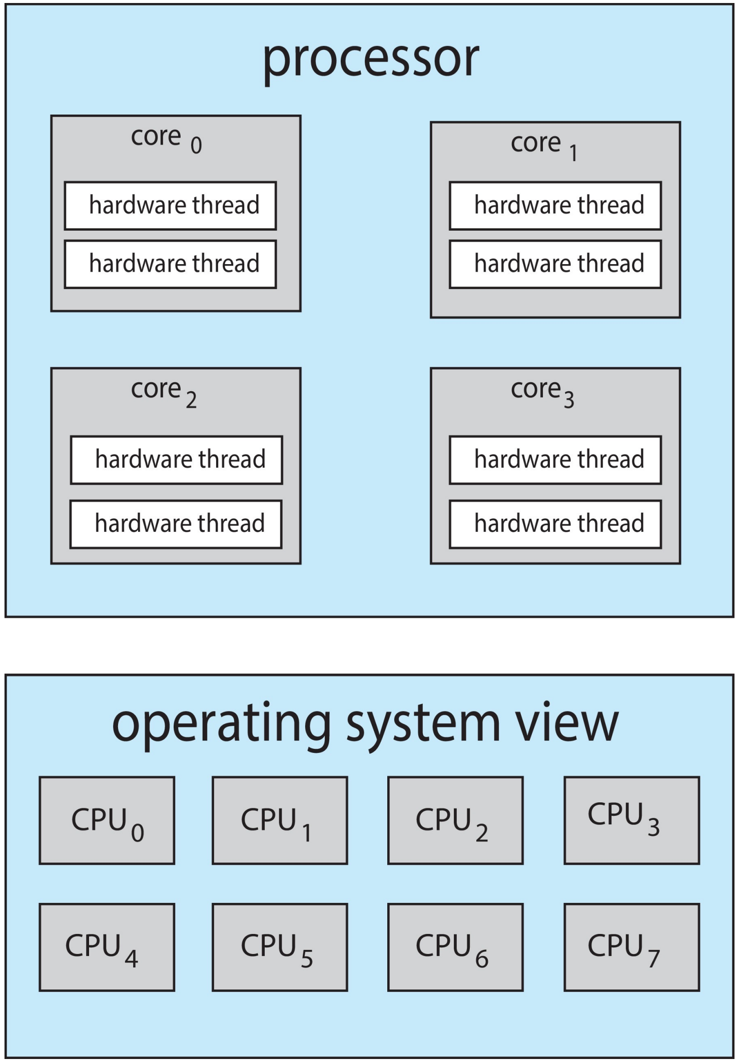
If one thread has a memory stall, switch to another thread!



Multithreaded Multicore System

- **Chip-multithreading** (CMT) assigns each core multiple hardware threads. (Intel refers to this as **hyperthreading**.)

- On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors.

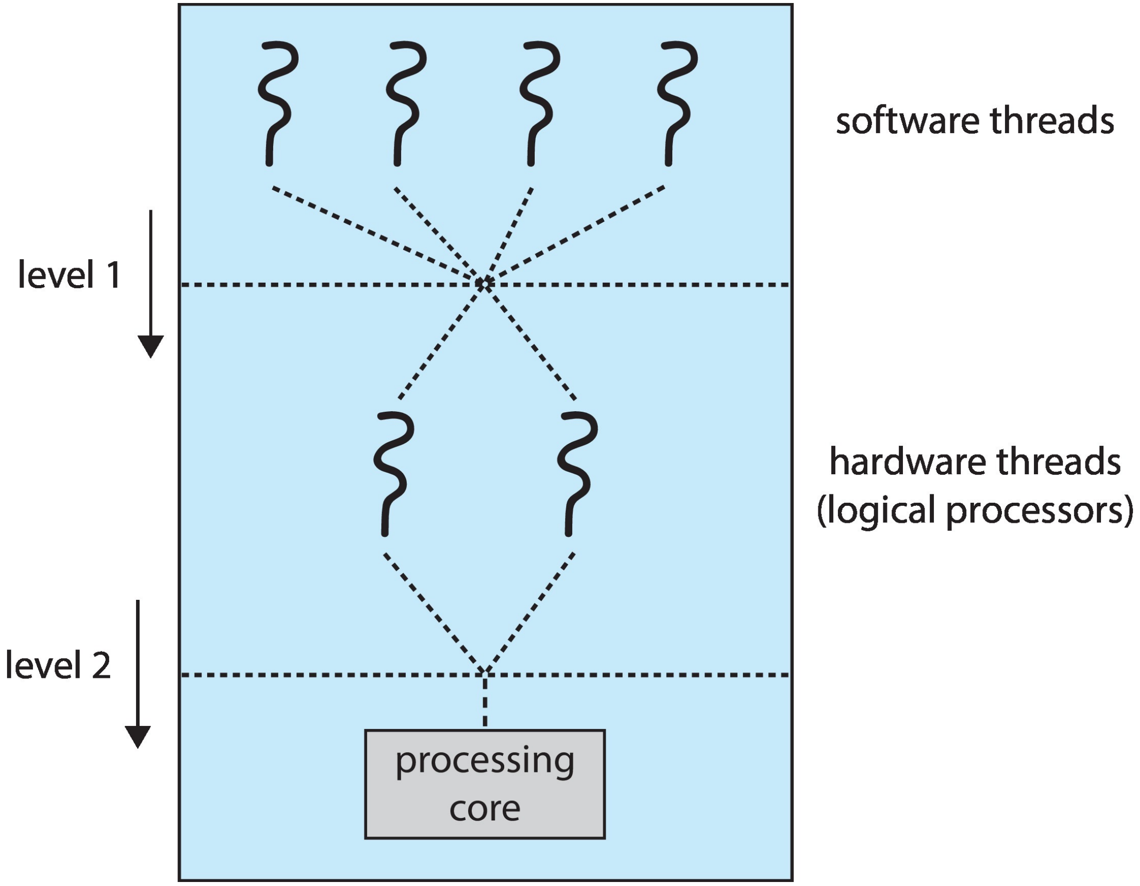


Multithreaded Multicore System

- Two levels of scheduling:

1. The operating system deciding which software thread to run on a logical CPU

2. How each core decides which hardware thread to run on the physical core.



Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor

Multiple-Processor Scheduling – Processor Affinity

- When a thread has been running on one processor, the cache contents of that processor stores the memory accesses by that thread.
- We refer to this as a thread having affinity for a processor (i.e. “processor affinity”)
- Load balancing may affect processor affinity as a thread may be moved from one processor to another to balance loads, yet that thread loses the contents of what it had in the cache of the processor it was moved off of.
- **Soft affinity** – the operating system attempts to keep a thread running on the same processor, but no guarantees.
- **Hard affinity** – allows a process to specify a set of processors it may run on.

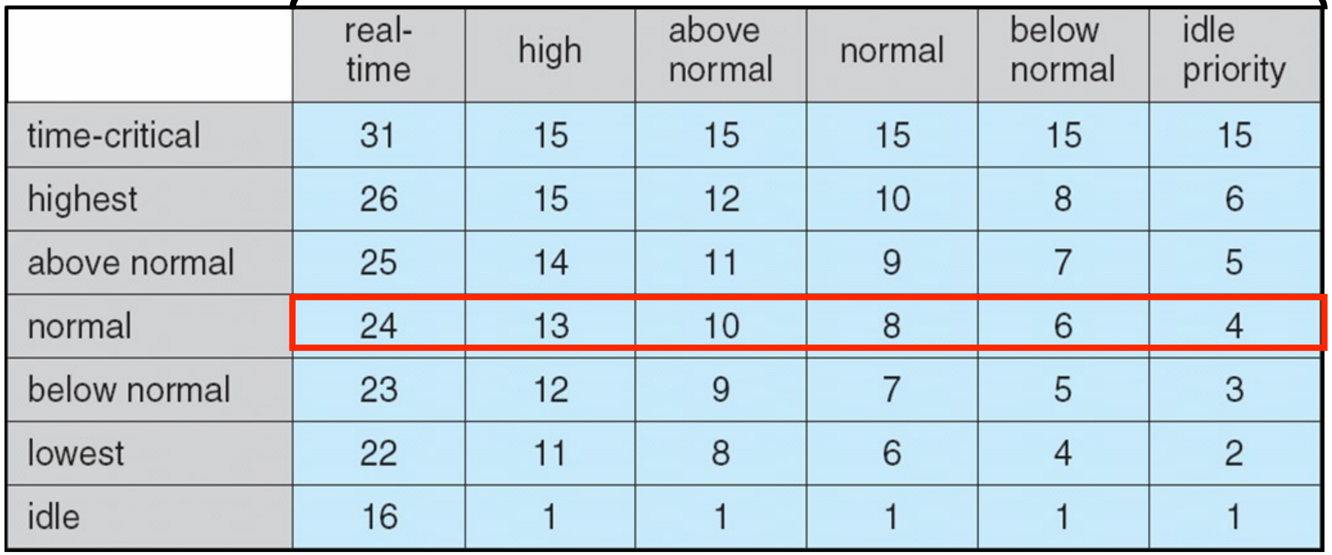
Operating System Examples

- Windows scheduling
- Linux scheduling

Win XP Scheduling

- Priority-based, time quantum-based, multi-queue, preemptive scheduling
- 32-level priority scheme: high number, high priority
  - Variable class: priorities 1 to 15
  - Real-time class: priorities 16 to 31
- The Win32 API exposes abstract priority concepts to users, which are translated into numerical priorities

User-settable Priority Class



	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

User-settable relative priority within a class

Base Priorities for each class

Win XP Scheduling

- When a thread’s quantum runs out, unless the thread’s in the real-time class (priority > 15), the thread’s priority is lowered
  - This is likely a CPU-bound thread, and we need to keep the system interactive
- When a thread “wakes up”, its priority is boosted
  - It’s likely an IO-bound thread
- The boost depends on what the thread was waiting for
  - e.g., if it was the keyboard, it’s definitely an interactive thread and the boost should be large
- These are the same general ideas as in other OSes (e.g., see Solaris priority scheme in textbook): preserving interactivity is a key concern
- The idle thread:
  - Win XP maintains a “bogus” idle thread
    - priority 0 or no priority
  - “runs” (and does nothing) if nobody else can run
  - Simplifies OS design to avoid the “no process is running” case

Priority in Linux

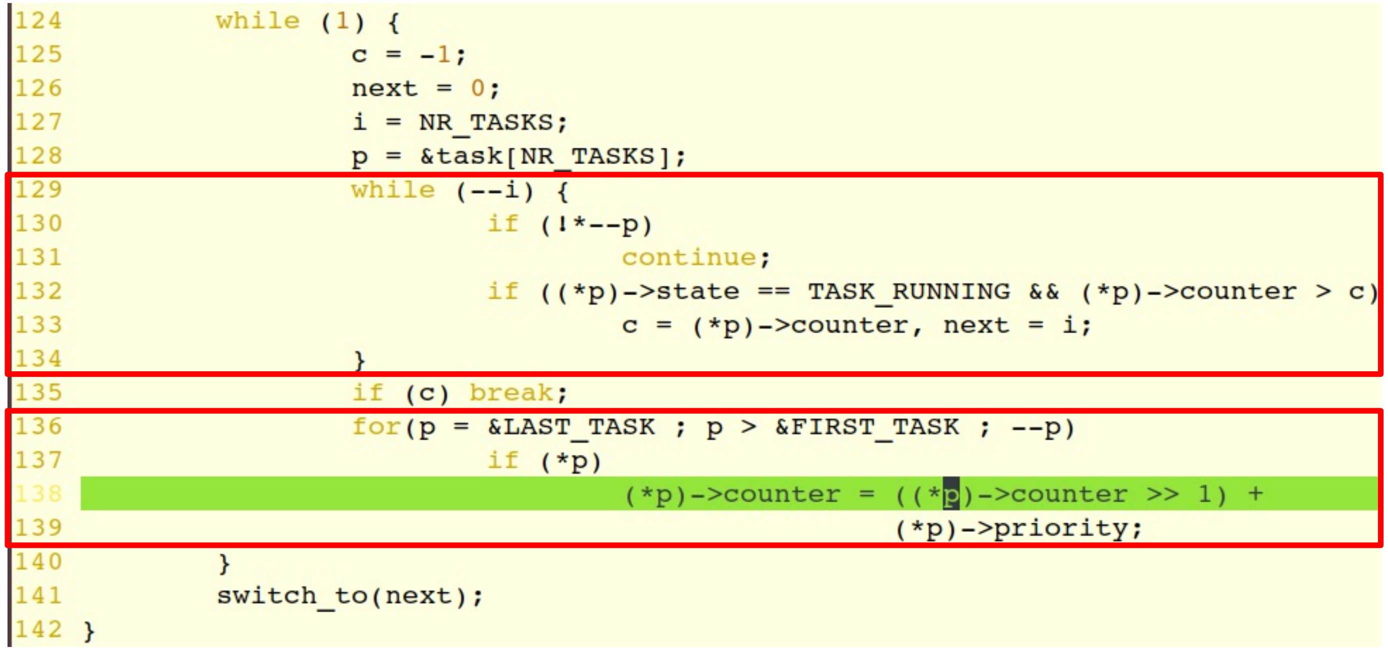
- Nice command
  - 数越小，优先级越高
  - ps -e -o uid,pid,ppid,pri,ni,cmd
  - nice -10 ./a.out
  - demo

Linux Scheduling

- The Linux kernel has a long history of scheduler development
- Linux 0.11 - 1991
- Linux 1.2 - 1995
- Linux 2.2 - 1999
- Linux 2.6.0 - 2003
- Linux 2.6.23 - 2007

Linux Scheduling: 0.11

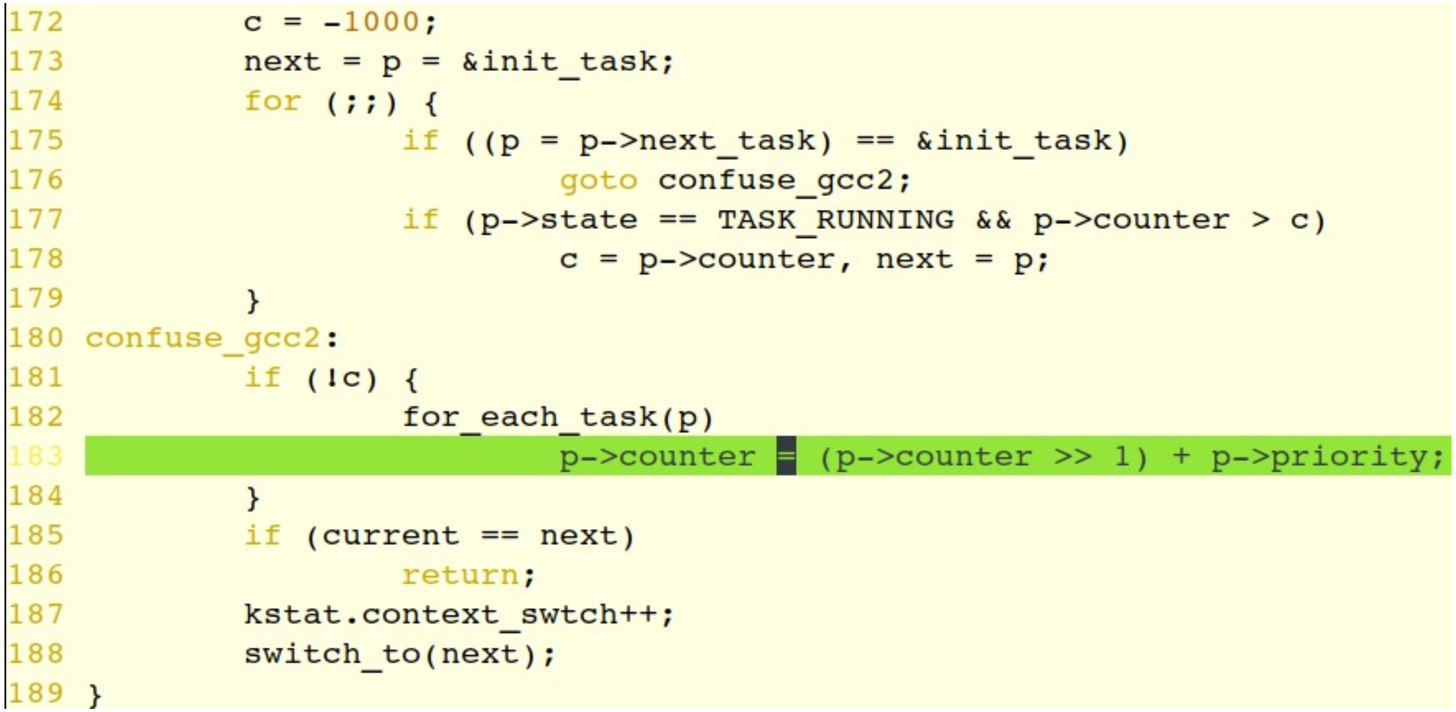
- Simplicity and speed
  - Round-robin + priority scheduling
  - Implemented with an array
  - TASK_RUNNING means ready



```
        while (1) {
                c = -1;
                next = 0;
                i = NR_TASKS;
                p = &task[NR_TASKS];
                while (--i) {
                        if (!*--p)
                                continue;
                        if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
                                c = (*p)->counter, next = i;
                }
                if (c) break;
                for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
                        if (*p)
                                (*p)->counter = ((*p)->counter >> 1) +
                                                (*p)->priority;
        }
        switch_to(next);
}
```

Linux Scheduling: 1.2

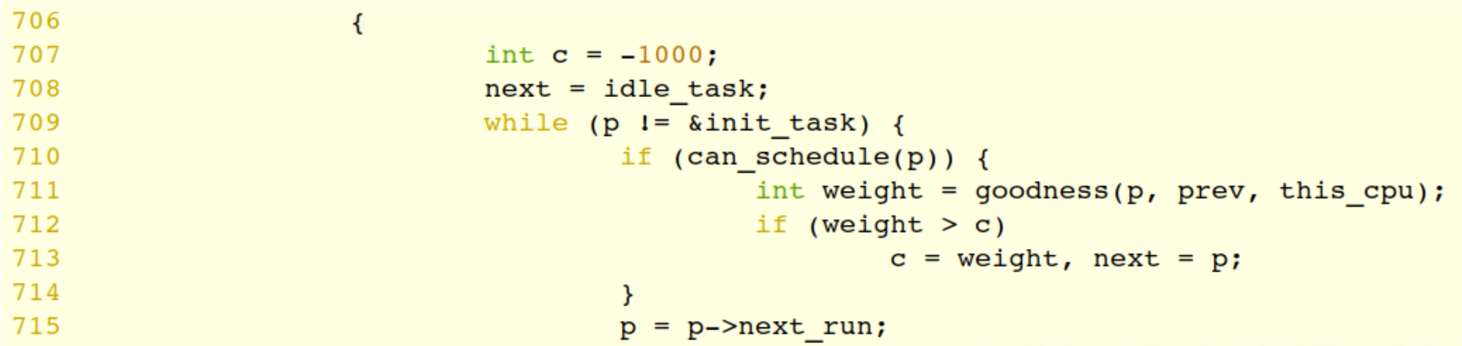
- Simplicity and speed
  - Round-robin + priority scheduling
  - Implemented with a circular queue



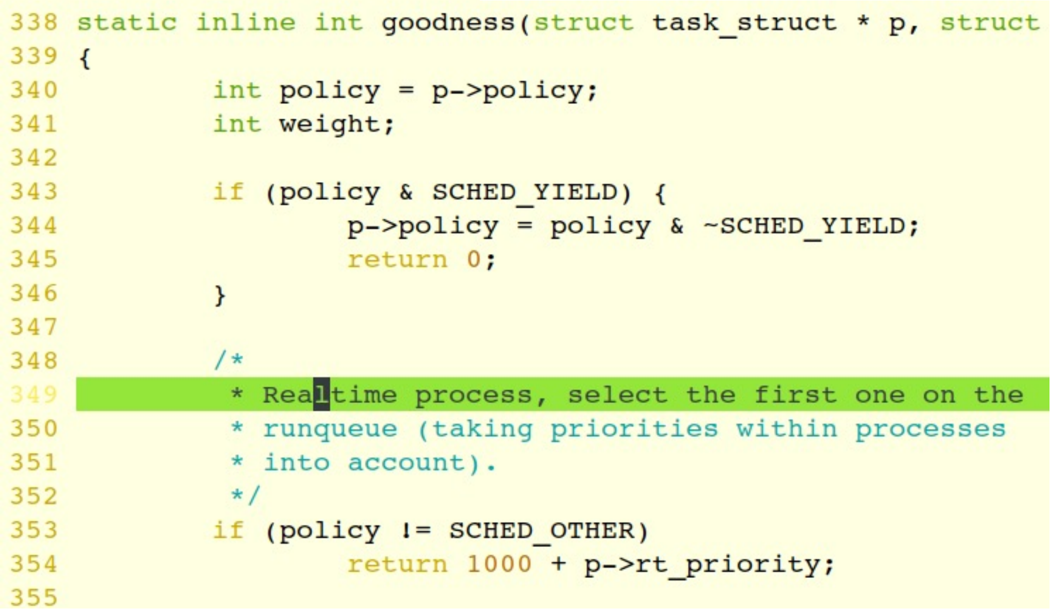
```
        c = -1000;
        next = p = &init_task;
        for (;;) {
                if ((p = p->next_task) == &init_task)
                        goto confuse_gcc2;
                if (p->state == TASK_RUNNING && p->counter > c)
                        c = p->counter, next = p;
        }
confuse_gcc2:
        if (!c) {
                for_each_task(p)
                        p->counter = (p->counter >> 1) + p->priority;
        }
        if (current == next)
                return;
        kstat.context_swtch++;
        switch_to(next);
}
```

Linux Scheduling: 2.2

- Kernel 2.2: toward sophistication
  - Scheduling classes
    - real-time, non-preemptible, non-real-time
  - Priorities within classes



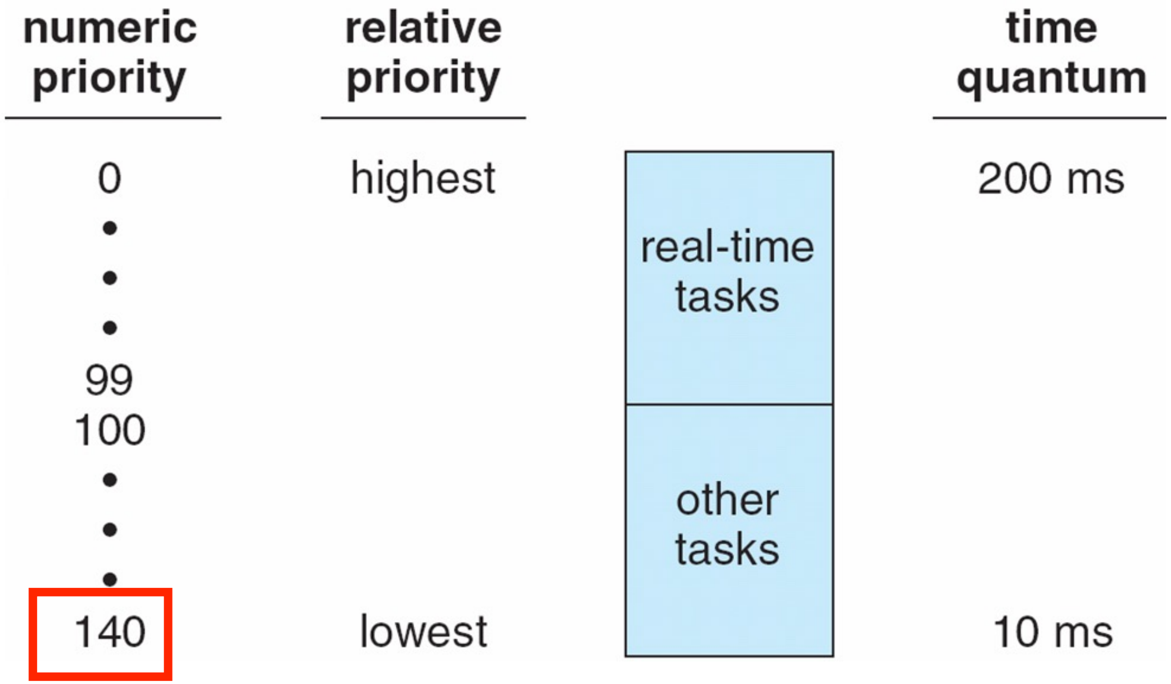
```
                {
                        int c = -1000;
                        next = idle_task;
                        while (p != &init_task) {
                                if (can_schedule(p)) {
                                        int weight = goodness(p, prev, this_cpu);
                                        if (weight > c)
                                                c = weight, next = p;
                                }
                                p = p->next_run;
                        }
```



```
static inline int goodness(struct task_struct * p, struct
{
        int policy = p->policy;
        int weight;

        if (policy & SCHED_YIELD) {
                p->policy = policy & ~SCHED_YIELD;
                return 0;
        }

        /*
         * Realtime process, select the first one on the
         * runqueue (taking priorities within processes
         * into account).
         */
        if (policy != SCHED_OTHER)
                return 1000 + p->rt_priority;

```

Linux Priorities

- Priority scheme:
  - low value means high priority



Linux Scheduling: 2.4

- The schedule proceeds as a sequence of epochs
- Within each epoch, each task is given a time slice of some duration
  - Time slice durations are computed differently for different tasks depending on how they used their previous time slices
- A time slice doesn’t have to be used “all at once”
  - A process cant get the CPU multiple times in an epoch, until its time slice is used
- Once all READY processes have used their time slice, then the epoch ends, and a new epoch begins
  - Of course, some processes are still blocked, waiting for events, and they’ll wake up during an upcoming epoch

Linux Scheduling: 2.4

- How to compute time slices?
  - If a process uses its whole time slice, then it will get the same one
  - If a process hasn’t used its whole time slice (.e.g., because blocked on I/O) then it gets a larger time slice!
- This may seem counter-intuitive but:
  - Getting a larger time slice doesn’t mean you’ll use it if you’re not READY anyway
  - Those processes that block often will thus never use their (enlarged) time slices
  - But, priorities between threads (i.e., how the scheduler picks them from the READY queue) are computed based on the time slice duration
    - A larger time slice leads to a higher priority

Linux Scheduling: 2.4

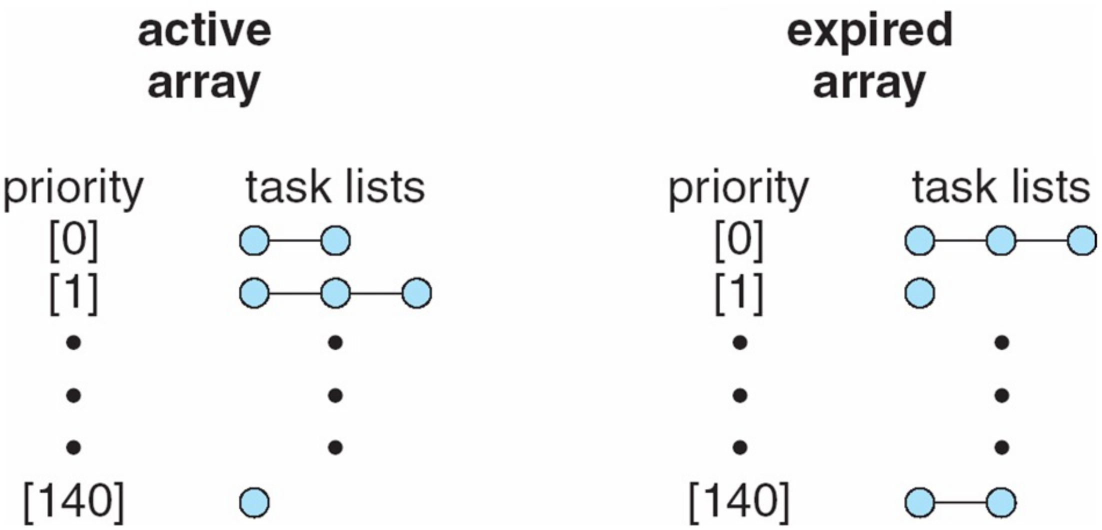
- Problem: O(n) scheduling
  - At each scheduling event, the scheduler needs to go through the whole list of ready tasks to pick one to run
  - If n (the number of tasks) is large, then it will take long to pick one to run
  - “Instead of spending your time thinking about it and wasting time, just run some task already!”
- There were other problems with 2.4 scheduling, e.g. multi-core machine
  - Increasing numbers of cores doesn’t make scheduling easier and schedulers will likely change dramatically in upcoming years

Linux Scheduling: 2.6

- Kernel 2.6 tries to resolve the O(n) problem
  - And a few others
- The so-called “O(1) scheduler”
  - Can be seen as implementation tricks so that one never need to have code that looks like “for all ready tasks do....”
- During an epoch, a task can be active or expired
  - active task: its time slice hasn’t been fully consumed
  - expired task: has used all of its time slice

Linux Time Slices

- The kernel keeps two arrays of round-robin queues
  - One for active tasks: one Round Robin queue per priority level
  - One for expired tasks: one Round Robin queue per priority level



O(1) Scheduling

- The priority array data structure in the Kernel’s code:

```
struct prio_array {
        int     nr_active;                         // total num of tasks
        unsigned long bitmap[5];                   // priority bitmap
        struct list_head queue[MAX_PRIO];          // the queues
 }
```

- What’s that bitmap thing?

Using a Bitmap for Speed

- The bitmap contains one bit for each priority level
  - 5*32 = 160 > 141 priority levels
- Initially all bits are set to zero
- When a task of a given priority becomes ready, the corresponding bit in the bitmap is set to one
  - Build a bit mask that looks like 0...010...0
  - Do a logical OR
- Finding the highest priority for which there is a ready task becomes simple: just find the first bit set to 1 in the bitmap
  - This doesn’t depend on the number of tasks in the system
  - Many ISAs provide an instruction to do just that
    - On x86, the instruction’s called bsfl (bit scan forward - from right to left)
- Finding the next task to run (in horrible pseudo-code) is then done easily:
  - prio_array.head_queue[bsfl(bitmap)].task_struct
  - No looping over all priority levels, so we’re O(1)

Recalculating Time Slices

- When the time slice of a task expires it is moved from the active array to the expired array
- At this time, the task’s time slice is recomputed
  - That way we never have a “recompute all time slices” which would monopolize the kernel for a while and hinder interactivity
  - Maintains the O(1)-time property
- When the active array is empty, it is swapped with the expired array
  - This is a pointer swap, not a copy, so it’s O(1)-time
- Time-slice and priority computations attempt to identify more interactive processes
  - Keeps track of how much they sleep
  - Uses priority boosts
  - And other bells, and whistles
- All details in “Linux Kernel Development”, Second Edition, by R. Love (Novell Press)

Linux ≥ 2.6.23

- Problem with the O(1) scheduler: the code in the kernel became a mess and hard to maintain
  - Seems to blur “policy” and “mechanism”?
- CFS: Completely Fair Scheduler
  - Developed by the developer of O(1), with ideas from others
- Main idea: keep track of how fairly the CPU has been allocated to tasks, and “fix” the unfairness
- For each task, the kernel keeps track of its virtual time
  - The sum of the time intervals during which the task was given the CPU since the task started
  - Could be much smaller than the time since the task started
- Goal of the scheduler: give the CPU to the task with the smallest virtual time
  - i.e., to the task that’s the least “happy”

Linux ≥ 2.6.23

- Tasks are stored in a red-black tree
  - O(log n) time to retrieve the least happy task
  - O(1) to update its virtual time once it’s done running for a while
  - O(log n) time to re-insert it into the red-black tree
- As they are given the CPU, tasks migrate from the left of the tree to the right
- Note that I/O tasks that do few CPU bursts will never have a large virtual time, and thus will be “high priority”
- Tons of other things in there controlled by parameters
  - e.g., how long does a task run for?

Linux Scheduling

- Not everybody loves CFS
  - Some say it just will not work for running thousands of processes in a “multi-core server” environment
  - But then the author never really said it would
- At this point, it seems that having a single scheduler for desktop/laptop usage and server usage is just really difficult
- Having many configuration parameters is perhaps not helpful
  - How do you set them?
- Other schedulers are typically proposed and hotly debated relatively frequently

Takeaway

- There are many options for CPU scheduling
  - FCFS
  - SJF
  - SRPT
  - RR
  - …
- Modern OSes use preemptive scheduling
- Some type of multilevel feedback priority queues is what most OSes do right now
- A common concern is to ensure interactivity
  - I/O bound processes often are interactive, and thus should have high priority
  - Having “quick” short-term scheduling is paramount