CPU Scheduling

ICS332

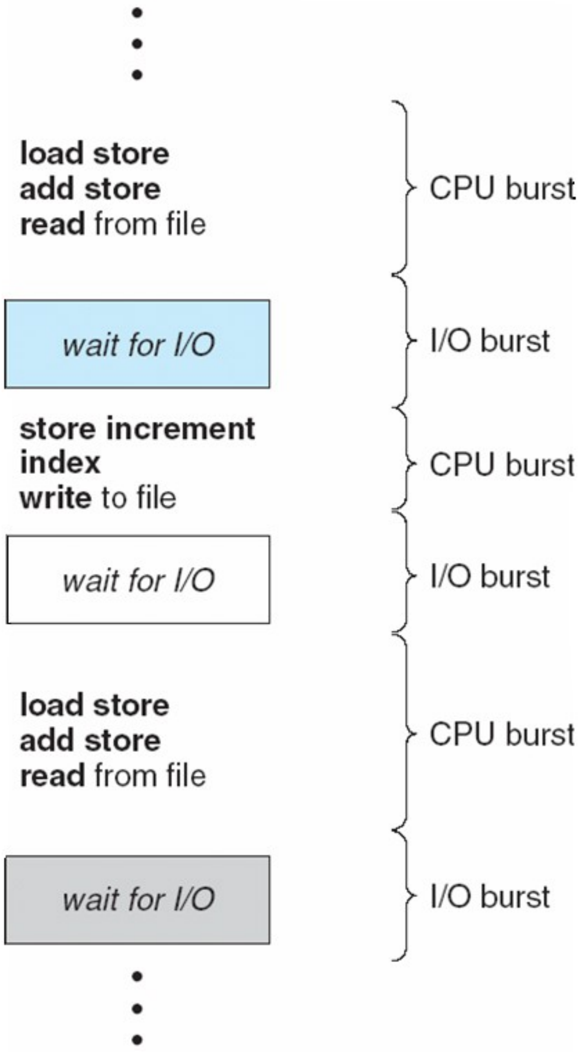
Operating Systems

CPU Scheduling

- CPU Scheduling: 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
  - Perhaps make processes “happy”
- 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)
- There are strong theoretical underpinnings here, but we’ll focus on pragmatic issues

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



The CPU Scheduler

- Whenever the CPU becomes idle, a ready process must be selected for execution
    - The OS keeps track of process states
    - This is called short-term scheduling
- Non-preemptive (or cooperative) scheduling: a process holds the CPU until it is willing to give it up
- 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
- Non-preemptive scheduling: #1, #4
  - Windows 3.x, Mac OS 9 (->2001)
- Preemptive scheduling: #1, #2, #3, #4, #5
  - Windows 95 and later, Max OS X, Linux

Preemptive Scheduling

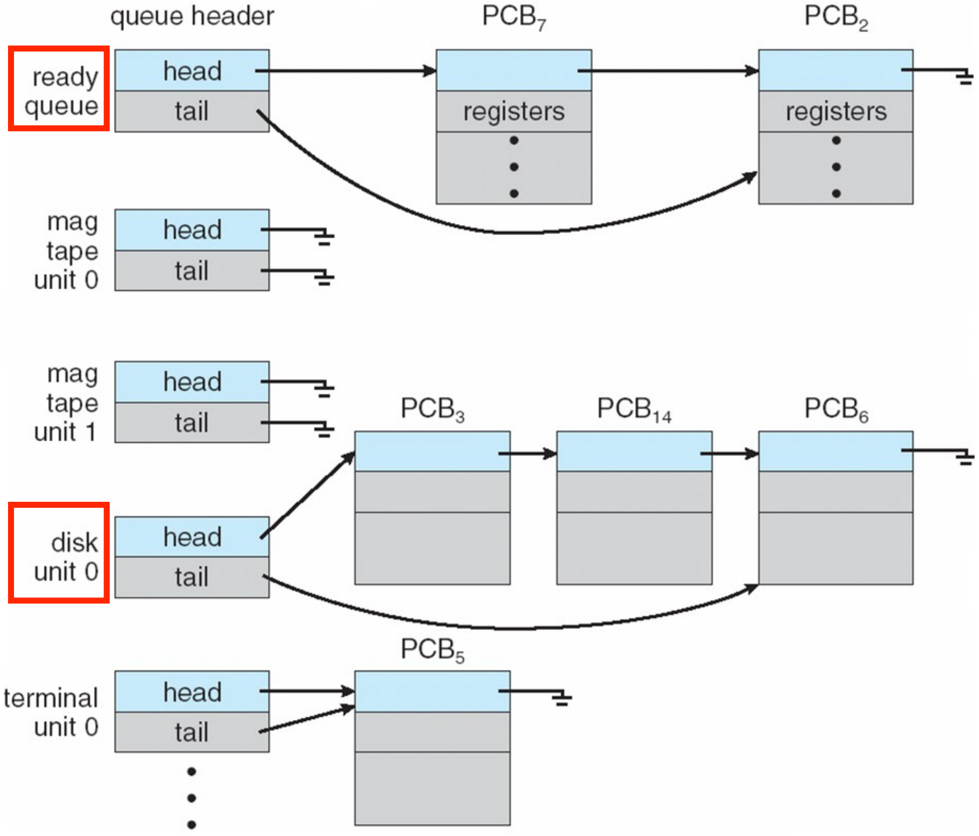
- Preemptive scheduling is good
  - No need to have processes willingly give up the CPU
  - The OS remains in control
- Preemptive scheduling is bad
  - Opens up many thorny issues having to do with process synchronization
    - If a process is in the middle of doing something critical and gets preempted, then bad things could happen
  - What if a process is preempted in the middle of a system call during which the Kernel’s updating its own data structures?
    - Disabling interrupts each time one enters the kernel is generally not a good idea

Scheduling Objectives

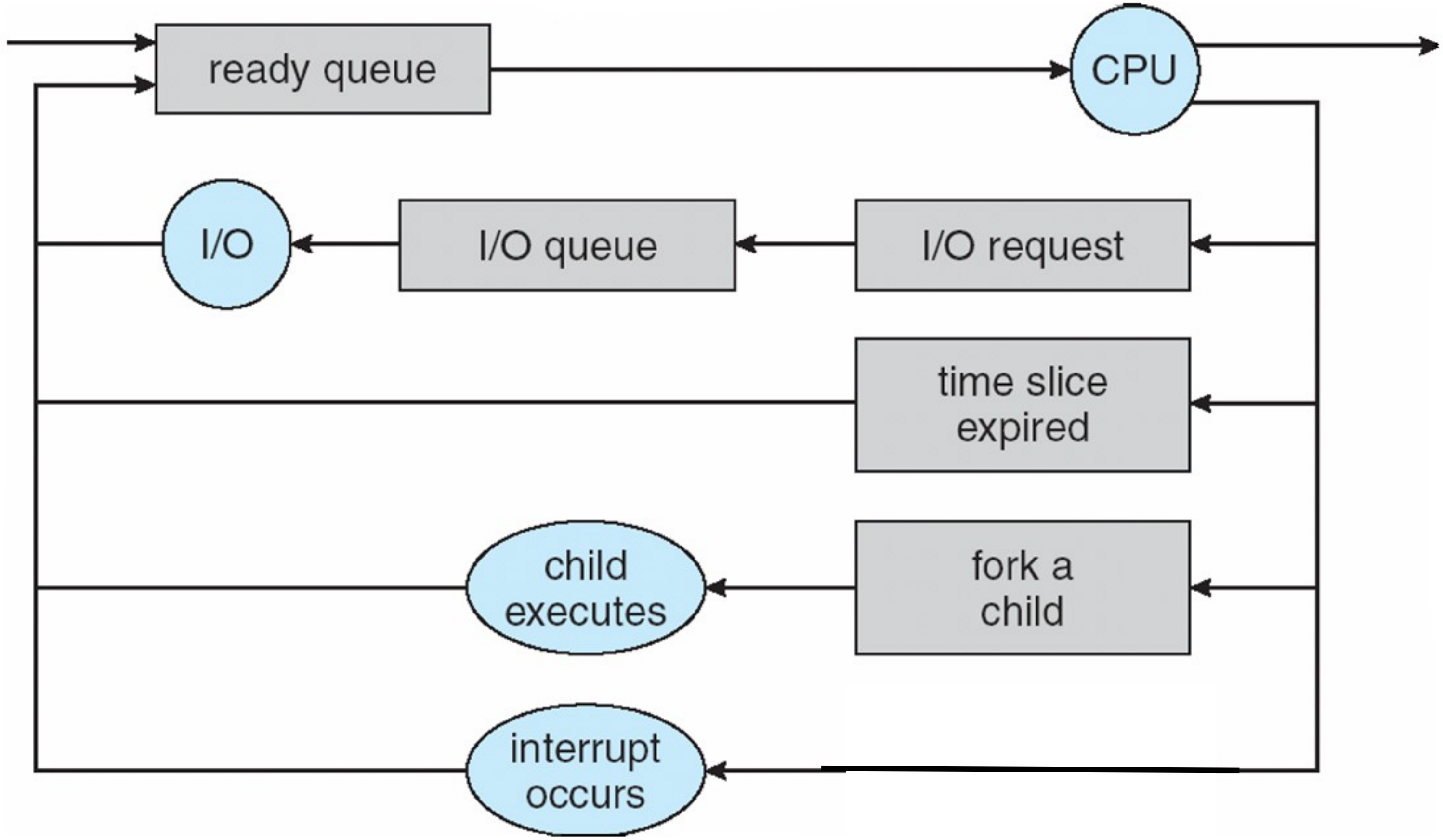
- Finding the right objective function is an open question
- There are many conflicting goals that one could attempt to achieve
  - Maximize CPU Utilization
    - Fraction of the time the CPU isn’t idle
  - Maximize Throughput
    - Amount of “useful work” done per time unit
  - Minimize Turnaround Time
    - 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
- The Ready Queue contains processes that are in the READY state
- Device Queues contain processes waiting for particular devices



Scheduling and Queues



Short-Term, Long-Term

- So far what we've described characterizes short-term scheduling
  - Something happens, react to it the best you can
- Other options consist in building a plan for the future
  - Based on information on the processes, come up with a clever arrangement of them in time and space
  - e.g., come up with a good mix of I/O-bound and CPU-bound processes to run together
- A short-term scheduler should be fast
  - So that it can run every 100ms or so
  - Therefore it cannot make very sophisticated decisions
- A long-term scheduler can be slow
  - It doesn't need to run as often
  - Therefore it can make sophisticated decisions
  - But it needs reasonably accurate information about the job mix, which is often a steep challenge
    - This is really the crux of the problem

Short-Term, Long-Term

- Typically, an OS doesn’t include a long-term scheduler
    - Although including “long-term features” in the short-term scheduler is tempting and done to some extent
- Long-term schedulers are built outside of the OS as an application/service
    - e.g., a batch scheduler for a cluster
- There is a lot of knowledge, research, and software development targeted to (good) long-term scheduling
    - One overriding question: how good is the information we have about the job mix and how stationary is it?
    - How bad is the scheduling when done with bad information?
- “OS Scheduling” typically implies short-term
- Read Section 3.2.2 for further discussion of short-term vs. long-term scheduling

Short-Term Scheduling Algs

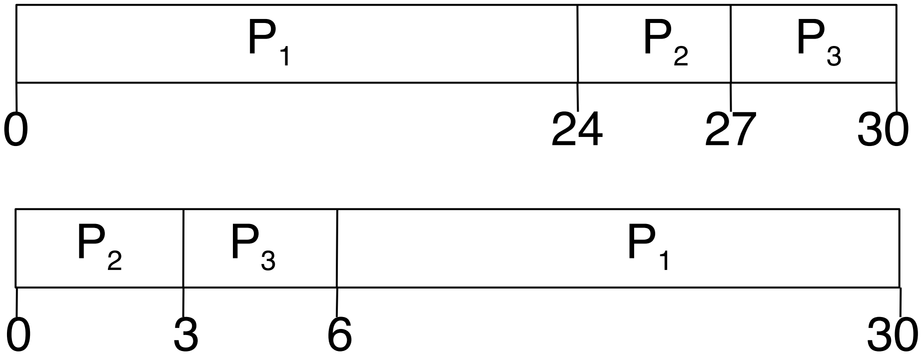
- Now that we understand the reasons and the mechanisms (queues, dispatcher, context switching) behind short-term scheduling, the question is: what’s a good policy?
  - i.e., what (good) algorithms should be implemented to decide on which process runs?
- Defining “good” is very difficult, due to the 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
- One thing is certain: the algorithms cannot be overly complicated so that they can be fast
- Let’s see a few standard algorithms

(Non-Preemptive) FCFS

- FCFS: First Come First Serve
- Straightforward: Implement the Ready Queue as a FIFO
- Problem: the average waiting time can be huge
- Textbook's example, assuming purely *CPU-bound* processes

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Gantt charts for two orders of (almost simultaneous) arrivals:



average wait time = 17

average wait time = 3

(Non-Preemptive) FCFS

- Consider the following situation
  - 1 CPU-bound process with only a few I/O bursts
  - n I/O-bound processes with frequent short CPU bursts
- The “convoy effect”
  - All I/O-bound processes block on I/O
  - The CPU-bound gets the CPU
  - All I/O devices do their work
  - All I/O-bound processes go back to READY
  - But now they can’t place their next I/O request because they need the CPU, which is hogged by the CPU-bound process
  - Result: I/O resources sit idle even though there are many processes who could use them
- Non-Premptive FCFS is just not a good idea

FCFS vs Objective Functions

- Maximize CPU Utilization: Excellent *(but no I/O!)*
- Maximize Throughput: Highly dependent on first submitted job(s) duration
- Minimize Turnaround Time: Highly dependent on first submitted job(s) duration
- Minimize Waiting Time: Highly dependent on first submitted job(s) duration
- Minimize Response Time: Highly dependent on first submitted job(s) duration

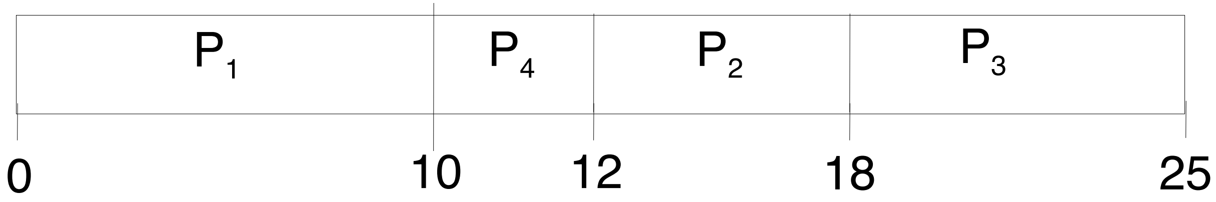
- Non-Premptive FCFS is in general just not a good idea

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:



average wait time = 10

average elapsed time = 16.25

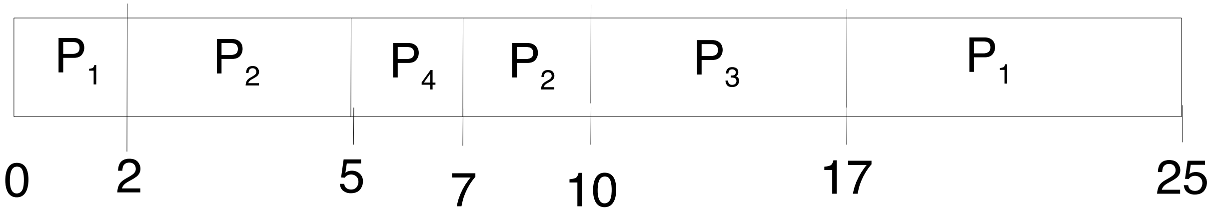
average turnaround time = 13.5

Shortest Job First (SJF)

- “Shortest-next-CPU-burst” algorithm
- 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:



average wait time = 5.75

average elapsed time = 12

average turnaround time = 14.75

SJF vs Objective Functions

- Maximize CPU Utilization: Excellent *(but still no I/O!)*
- Maximize Throughput: NonPreemptive: OK / Preemptive: Good
- Minimize Turnaround Time: NonPreemptive: OK / Preemptive: Good
- Minimize Waiting Time: Best (!) *(see next slide)*
- Minimize Response Time: NonPreemptive: OK / Preemptive: OK

- Non-Premptive SJF is OK; Preemptive is better but is it the best?

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 and for particular hypotheses
- 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

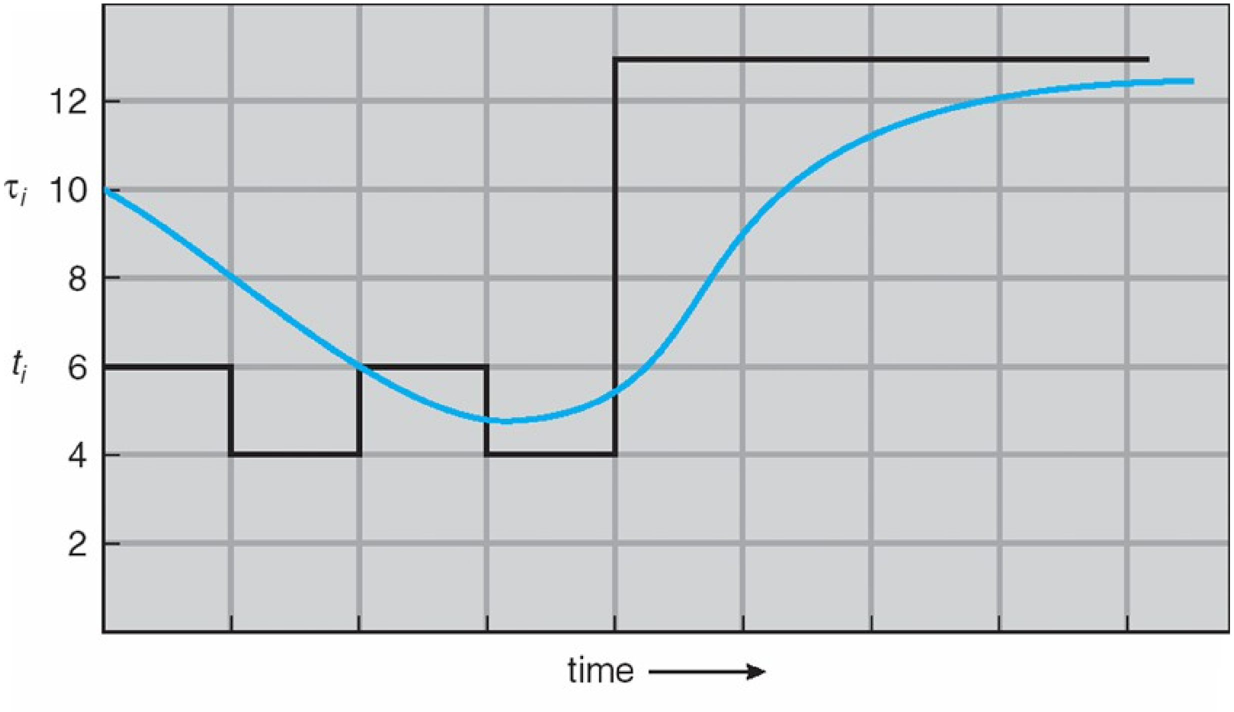
parameter between 0 and 1
0: don’t care about most recent history
1: care only about the most recent history
.5: some compromise

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	…

Priority Scheduling

- SJF is a special case of 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
- 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 6.3.3
- 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 (i.e. indefinite blocking)
  - 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

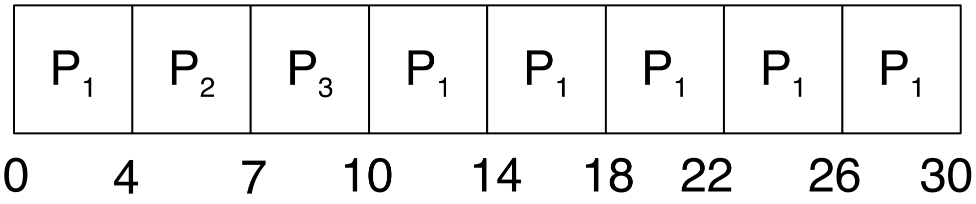
Round-Robin Scheduling

- 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

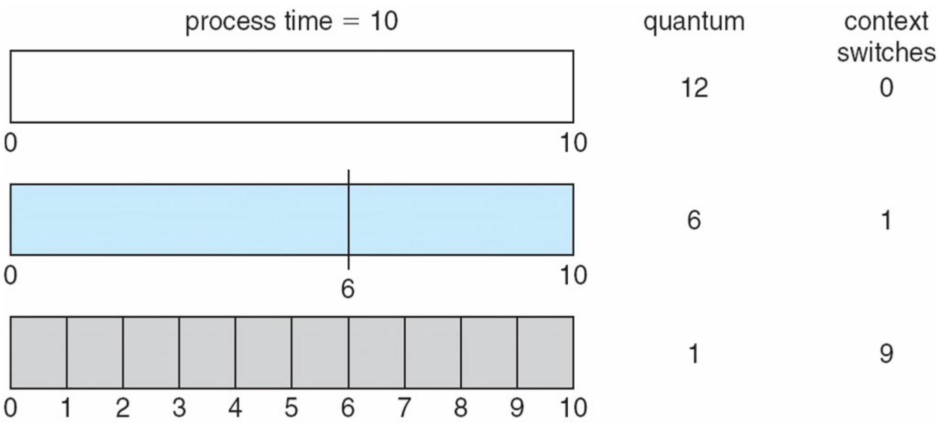
Process	Burst Time
P_1	24
P_2	3
P_3	3

quantum = 4



- Typically, higher average wait time than SJF, but better response time
  - And the wait time is bounded!

Picking the Right Quantum

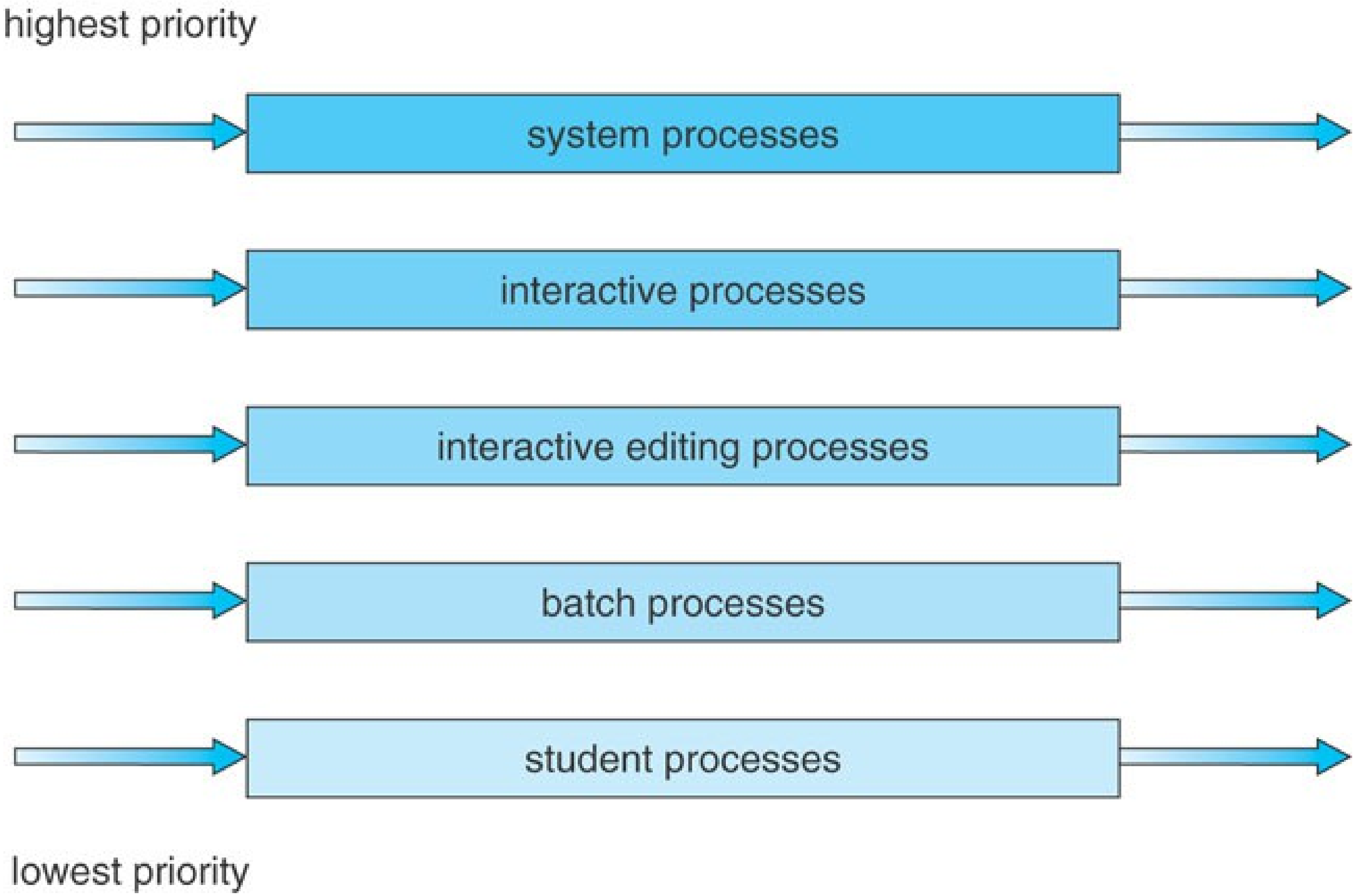


- Trade-off:
  - Short quantum: great response/interactivity but high overhead
    - Hopefully not too high if the dispatcher is fast enough
  - Long quantum: poor response/interactivity, but low overhead
    - With a very long time quantum, RR Scheduling becomes FCFS Scheduling
- If context-switching 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: 10 μs

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


highest priority

system processes

interactive processes

interactive editing processes

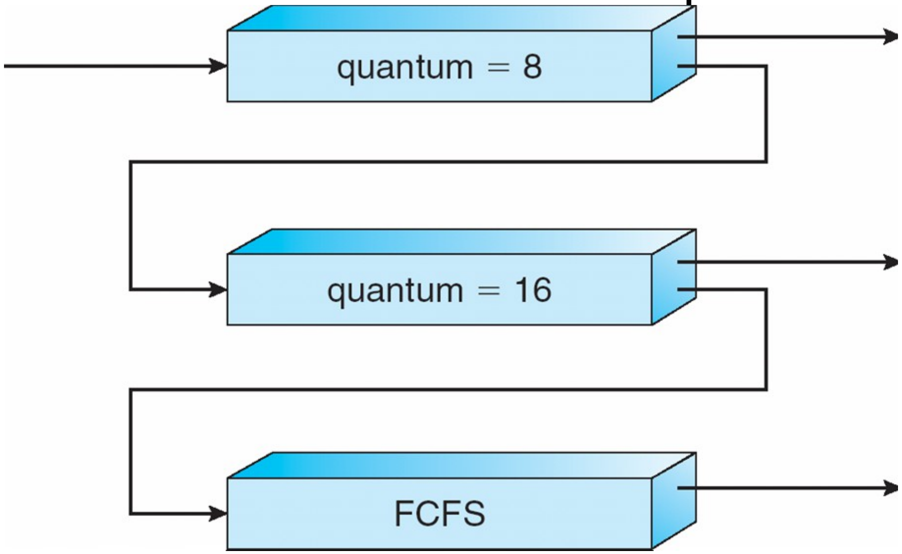
batch processes

student processes

lowest priority

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?

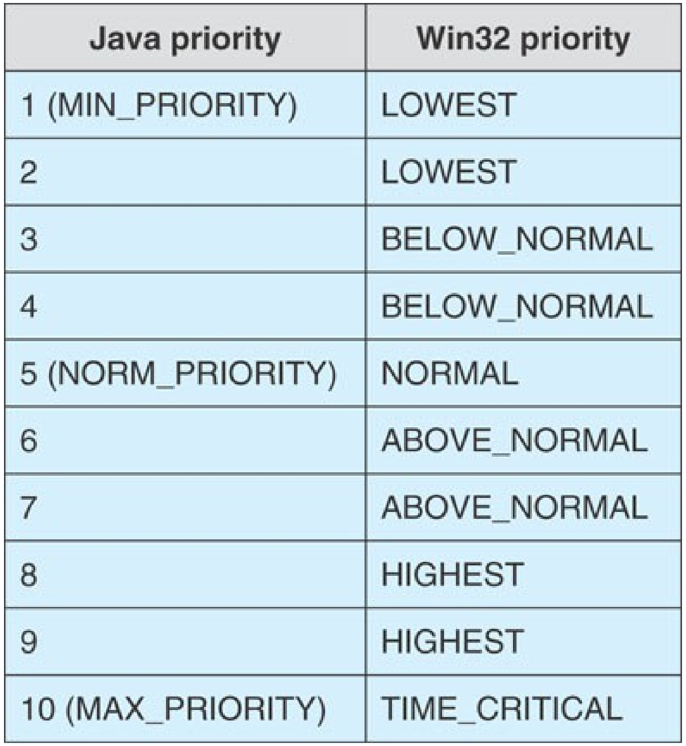
- Few analytical/theoretical results are available
  - Essentially, take two scheduling algorithms A and B, take a metric (e.g., wait time), and more likely than not you can find one instance in which A > B, and another instance in which A < B
  - In rare cases you can show that an algorithm is optimal (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 in Java

- The JVM defines a notion of thread priority
    - Vaguely defined, not necessarily preemptive
    - Essentially some “threads” are preferred over others, but you can’t rely on anything clear
    - But for very old ones JVMs do things that one would expect (e.g., preemptive multi-queue round-robin)
- A thread can yield control of the CPU by calling Thread.yield() *(… But don't do it)*
- The thread class has Thread.setPriority() and Thread.getPriotity()
    - Priorities are between Thread.MIN_PRIORITY (lowest) and Thread.MAX_PRIORITY (highest)

Thread Scheduling in Java

- The JVM uses the user-specified thread priorities to convey information to the OS, who makes the final calls
- Thread scheduling in the JVM is not portable (i.e., when writing code you cannot assume anything about thread scheduling)
    - Unless you use ThreadPool, in which case you can configure the thread pool to be scheduled precisely



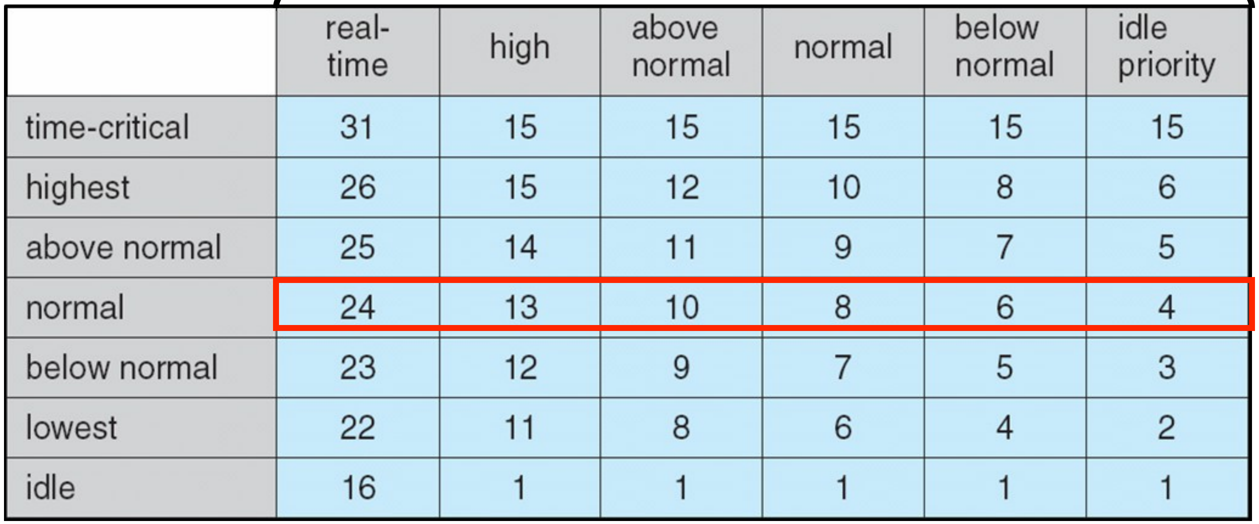
Java priority	Win32 priority
1 (MIN_PRIORITY)	LOWEST
2	LOWEST
3	BELOW_NORMAL
4	BELOW_NORMAL
5 (NORM_PRIORITY)	NORMAL
6	ABOVE_NORMAL
7	ABOVE_NORMAL
8	HIGHEST
9	HIGHEST
10 (MAX_PRIORITY)	TIME_CRITICAL

Win XP (and beyond) Scheduling

- Priority-based, time quantum-based, multi-queue, preemptive scheduling (Section 5.6.2)
- 32-level priority scheme: high number, high priority
  - Variable class: priorities 1 to 15
  - Real-time class: priorities 16 to 31
  - (A special memory-management thread runs at priority 0)
- The Win32 API exposes abstract priority concepts to users, which are translated into numerical priorities

User-settable Priority Class

User-settable relative priority within a 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

Base Priorities for each class

Win XP (and beyond) Scheduling

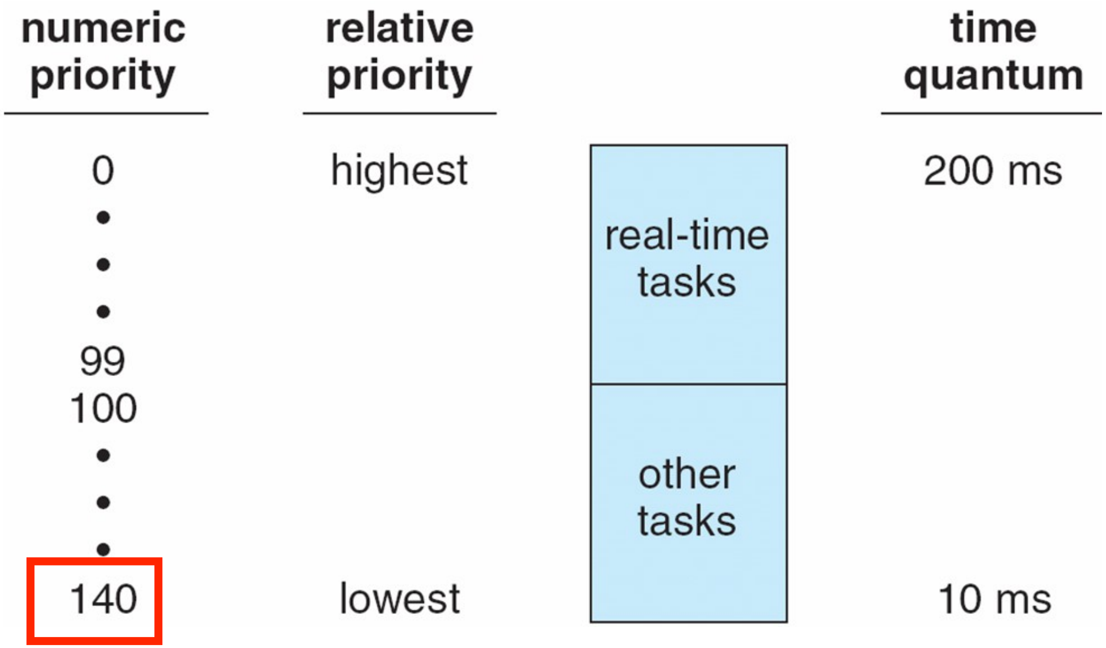
- When a thread's quantum runs out, unless the thread's in the real-time class (priority > 15), the thread's priority is lowered
  - This is likely a CPU-bound thread, and we need to keep the system interactive
- When a thread "wakes up", its priority is boosted
  - It's likely an IO-bound thread
- The 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 scheeme in textbook): preserving interactivity is a key concern
- The idle thread:
  - Win XP maintains a "bogus" idle thread (priority 1)
  - "runs" (and does nothing) if nobody else can run
  - Simplifies OS design to avoid the "no process is running" case

Linux Scheduling: 1.2 and 2.2

- The Linux kernel has a long history of scheduler development
- Kernel 1.2 (1995): simplicity and speed
  - Round-Robin scheduling
  - Implemented with a circular queue
- Kernel 2.2 (1999): toward sophistication
  - Scheduling classes
    - real-time, non-preemptible, non-real-time
  - Priorities within classes

Linux Priorities

- Priority scheme:
  - low value means high priority



Linux Scheduling: 2.4

- 2.4: 2001
- 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 user 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 didn’t make scheduling easier and schedulers changed dramatically in years

Linux Scheduling: 2.6.0 to 2.6.22

- Kernel 2.6 (2003) 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?
  - ICS312 if you're not familiar with bitmaps...

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
- 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
  - e.g., the BFS (Brain <expletive> Scheduler) for desktop/laptop machines that tries to be as simple as possible
    - One queue, no “interactivity estimators”, ...

Conclusions

- There are many options for CPU scheduling
- 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