

Operating Systems: CPU Scheduling

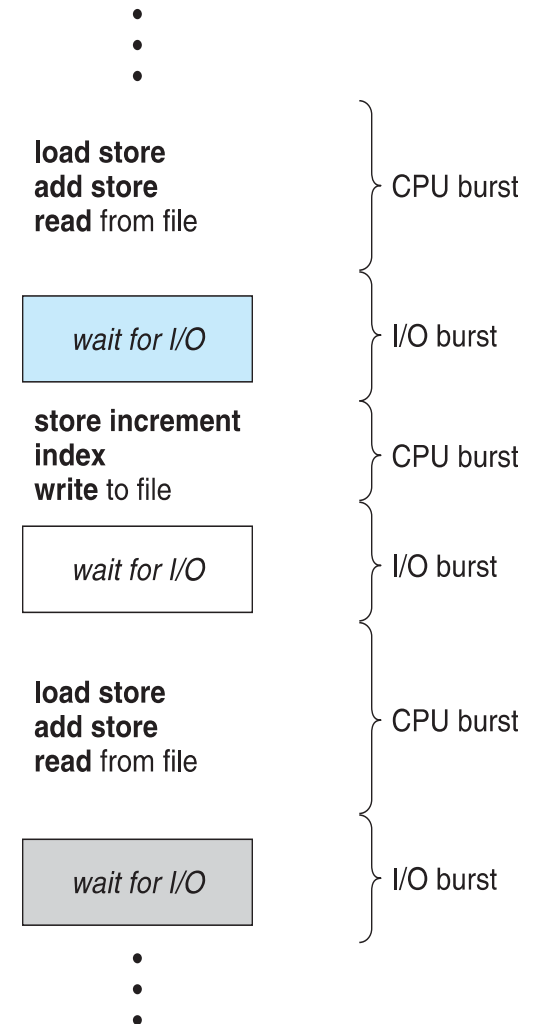
CSC-4320/6320 –Summer 2014

CPU Scheduling

- **Definition:** the decisions made by the OS to figure out which ready process/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 is never idle
 - Perhaps make processes “happy” to boot
- The **policy** is the scheduling strategy
- The **mechanism** is the **dispatcher**
 - A component of the OS that’s used to switch between process
 - That in turn uses the context switch mechanism
 - Must be lightning fast for time-sharing (dispatcher latency)
- There are strong theoretical underpinnings here,

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 processes
 - 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 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 “you’ve had enough” timer expires

Scheduling Decision Points

- Scheduling decision 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
- Preemptive scheduling: 1, 2, 3, 4, 5
 - Windows 95 and later, Mac OS X, Linux

Preemptive Scheduling

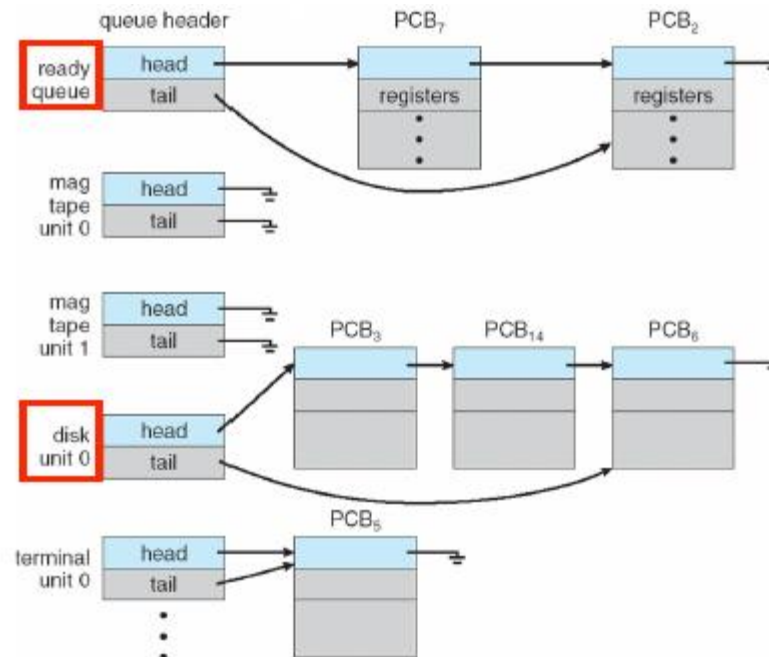
- Preemptive scheduling is good
 - No need to have processes willingly give up the CPU
 - The OS remains in control
- Preemptive scheduling is bad
 - Opens up many thorny issues having to do with process synchronization
 - If a process is in the middle of doing something critical and gets preempted,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

Scheduling Objectives

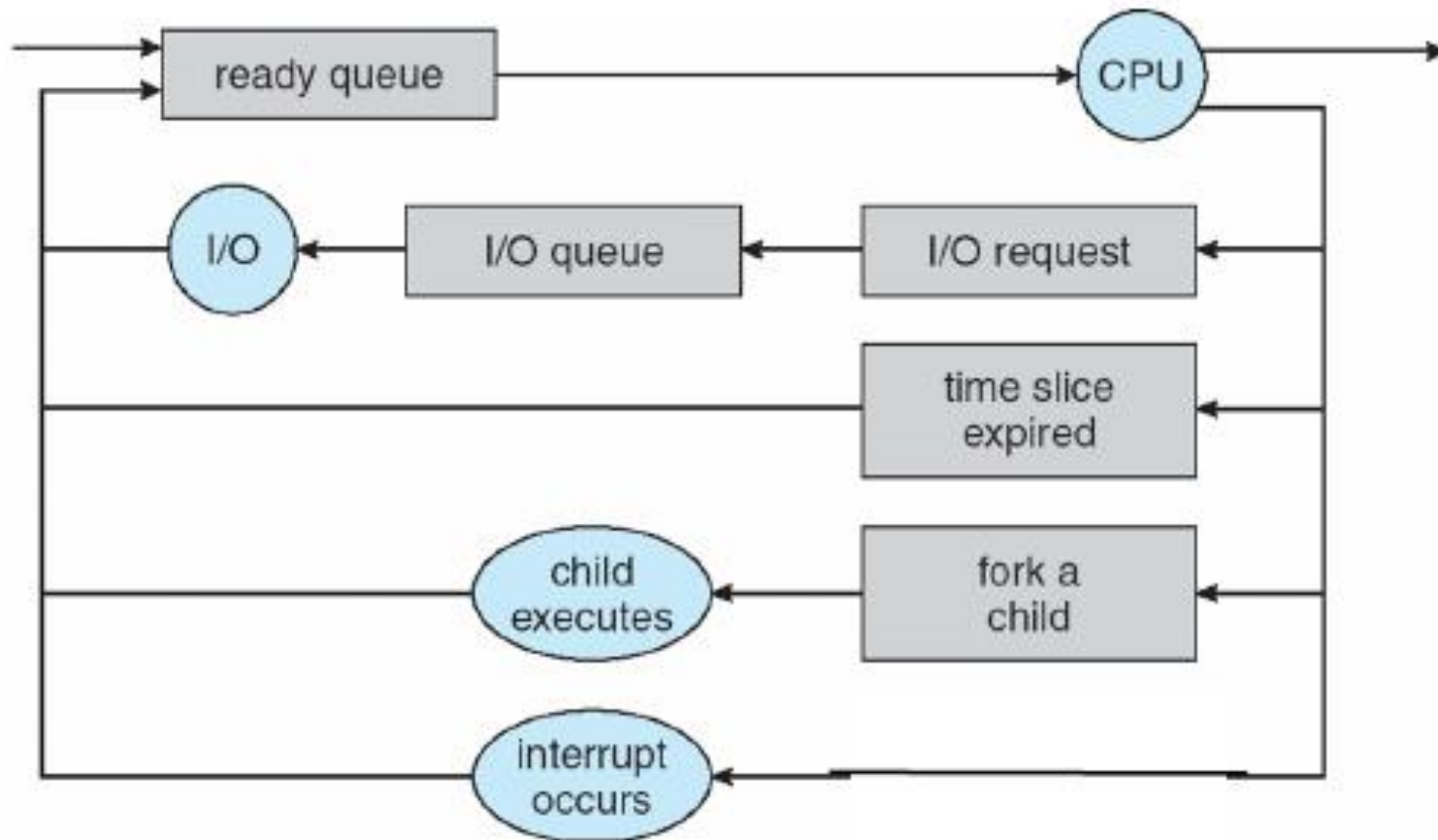
- Finding the right objectives is an open question
- There are many conflicting goals that one could attempt to achieve
 - Maximize **CPU Utilization**
 - Fraction of the time the CPU isn't idle
 - Maximize **Throughput**
 - Amount of “useful work” done per time unit
 - Minimize **Turnaround Time**
 - 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?
 - 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 the main issue

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 (good) long-term scheduling
 - On 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 reasons and the mechanism (queues, dispatchers, 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

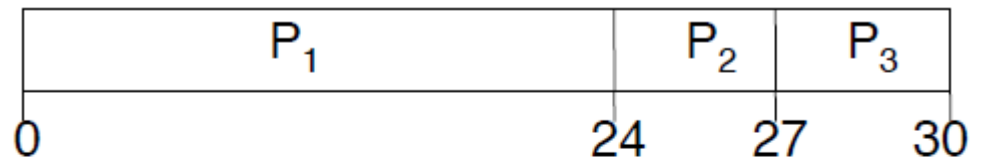
(Non-Preemptive) FCFS

- 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

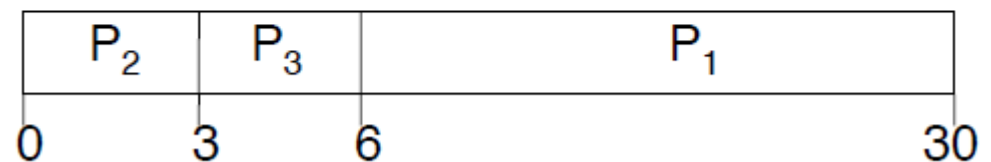
Process	Burst Time
P_1	24
P_2	3
P_3	3

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

average wait time=17



average wait time=3



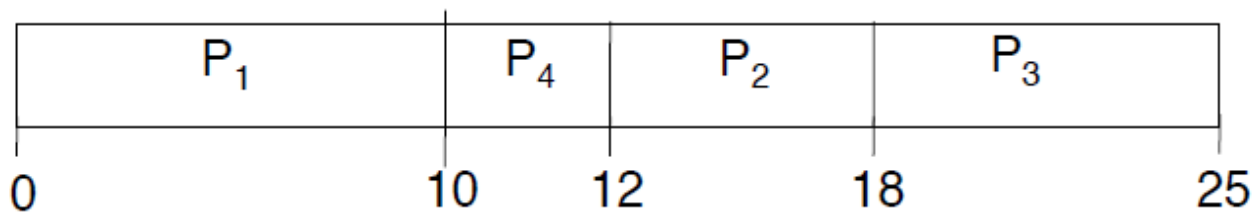
(Non-Preemptive) FCFS

- 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 process who could use them
- Non-Preemptive FCFS is 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



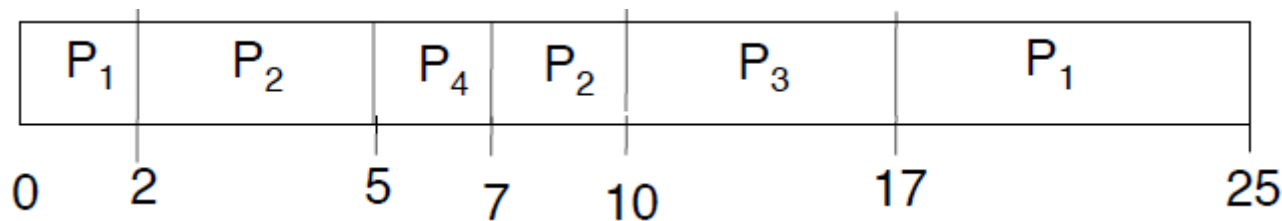
average wait time = 10

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



average wait time = 5.75

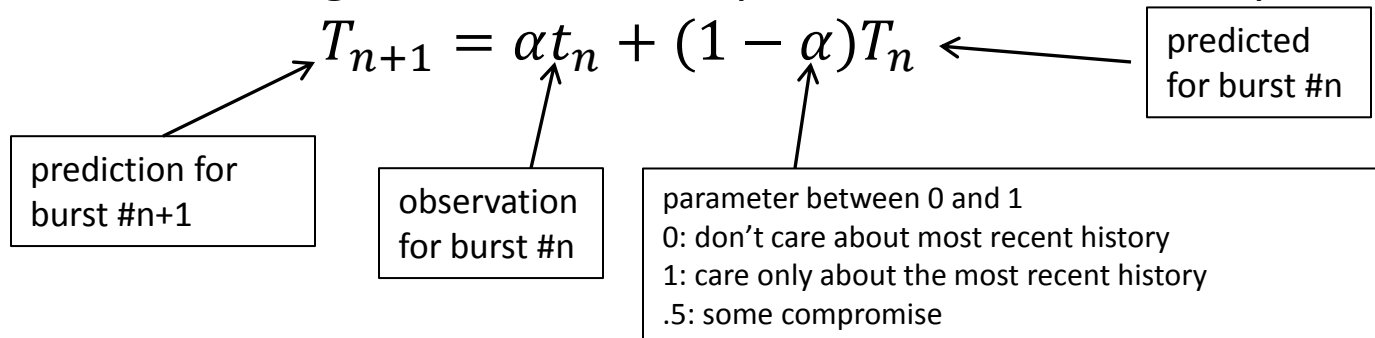
average turnaround time = 12

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 amount of 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., relies 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

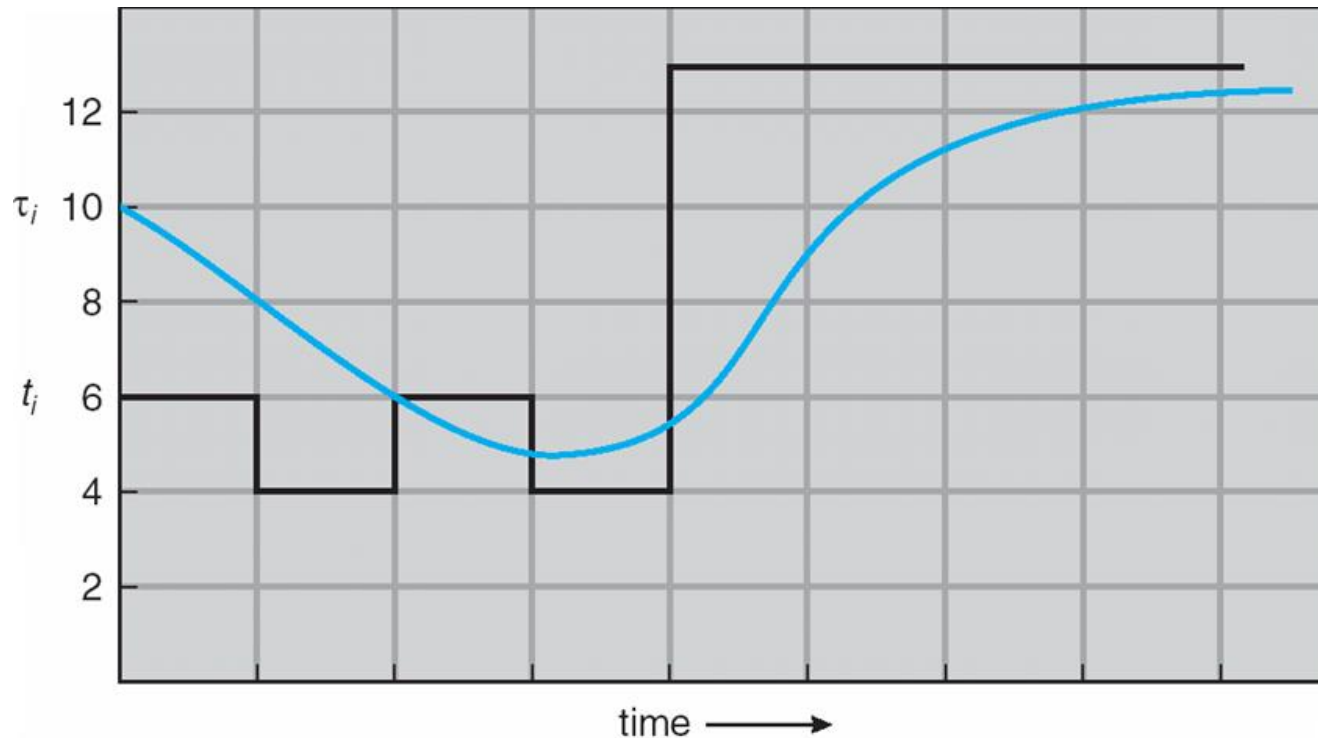
- One only knows the duration of a CPU burst once it is over
- Idea: predict future CPU bursts based on previous CPU bursts
- **Exponential averaging** of previously observed burst durations
 - Predict the future given the past
 - Give more weight to the recent past than the remote past



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

Exponential Averaging

$$T_0 = 10, \alpha = 0.5$$



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

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., set by users to specify relative importance of jobs
- 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 all that 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

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 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

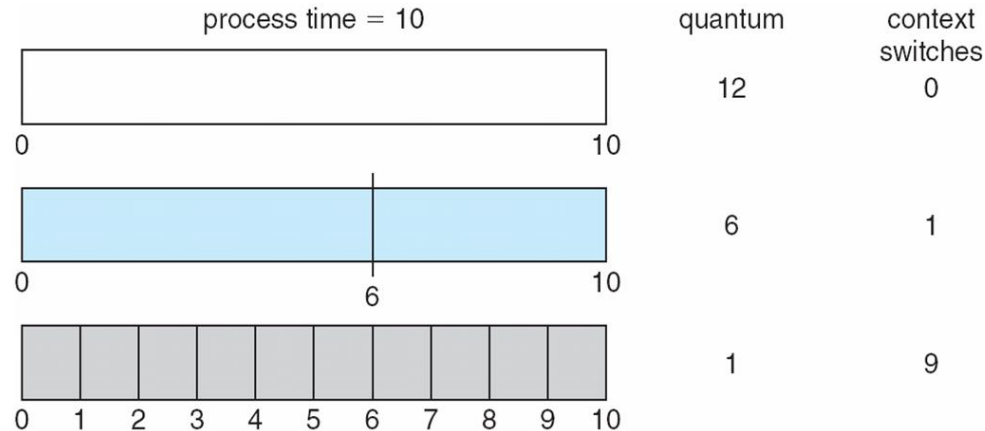
RR Scheduling Example

<u>Process</u>	<u>Burst Time</u>	
P_1	24	
P_2	3	
P_3	3	quantum = 4

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

- Typically higher average wait time than SJF, but better response time
 - The wait time is bounded though!

Picking the Right Quantum



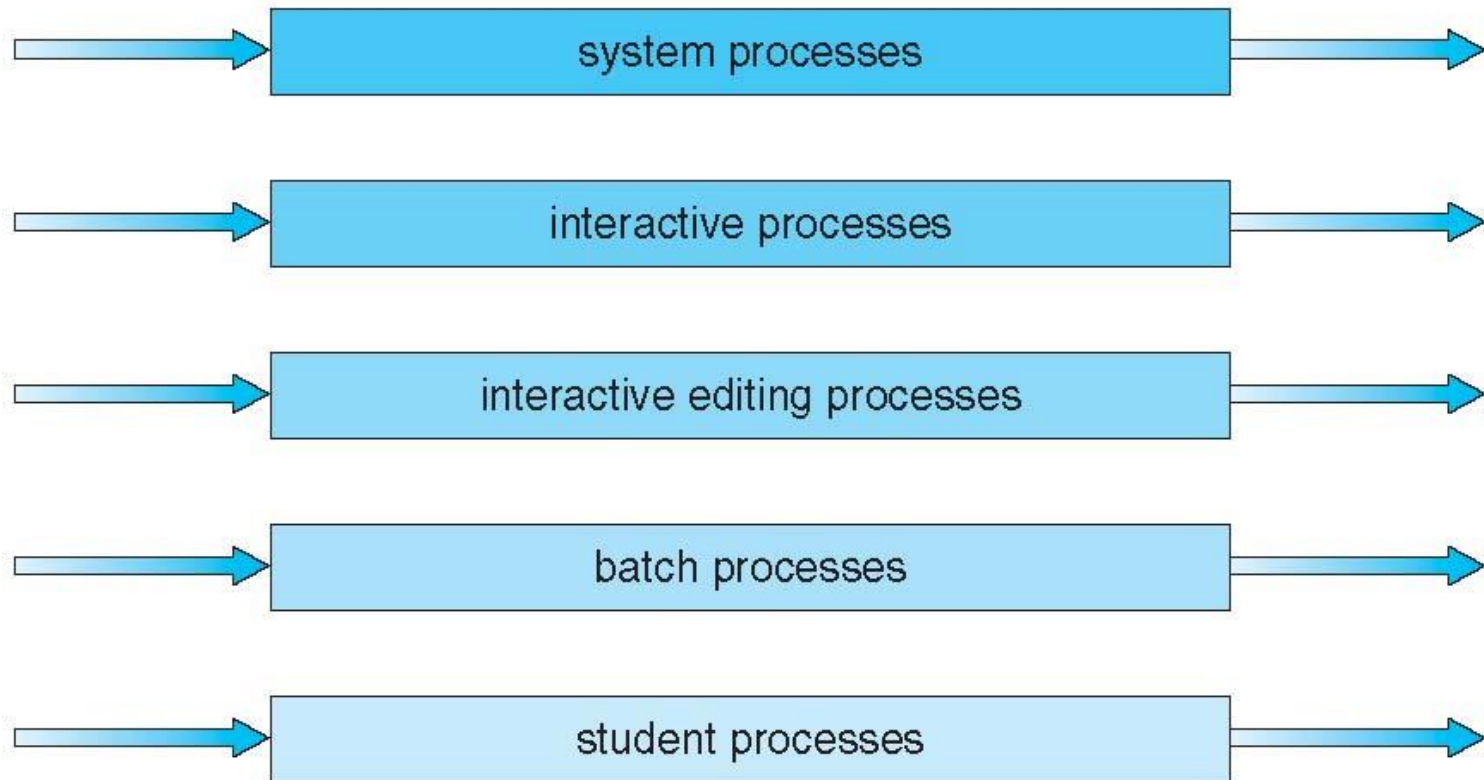
- Trade-off:
 - Short quantum: great response/interactivity but high overhead
 - Hopefully not too high if the dispatcher is fast enough
 - Long quantum: poor response/interactivity, but low overhead
 - With very long time quantum. RR Scheduling becomes FCFS Scheduling
- If context-switching time is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

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., Higher-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

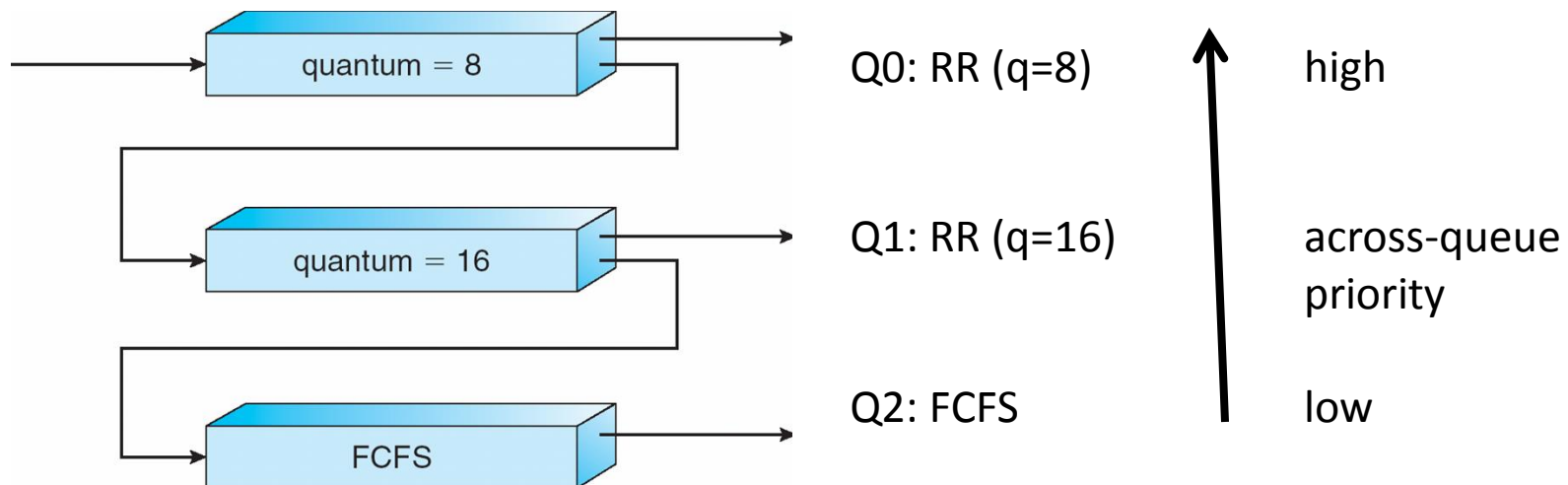
highest priority



lowest priority

Multi-Level 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 is also a good way to implement priority aging



Multi-Level Feedback Queues

- The scheme implements a particular CPU scheduling “philosophy”
 - A new process arrives
 - It is placed in Q0 and is, at some point, given a quantum of 8
 - If it does not use it all, it is likely an 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 point, is given a quantum of 16
 - If it does use it all, then it is likely a CPU-bound process and gets demoted to Q2
 - At that point the process runs only when no non-CPU intensive process need 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)

Multi-Level Feedback Queues

- The Multi-Level Feedback Queues scheme is very general because it is 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 promise but these promises are hard to achieve
- Also, it requires quite a bit of computation

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 where $A < B$
 - 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
 - Modern 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()`
- The thread class has `Thread.setPriority()` and `Thread.getPriority()`
 - 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 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 16-31
 - (A special memory-management thread runs at priority 0)
- The Win32 API exposes abstract priority concepts to users, which are translated into numerical priorities

	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

Win XP Scheduling

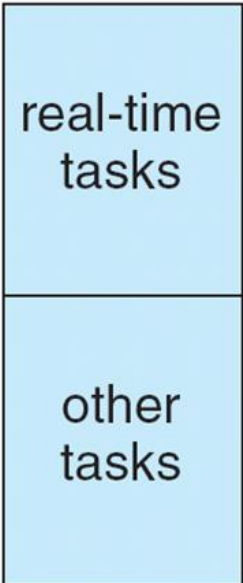
- When a thread's quantum runs out, unless the thread is in the real-time class (priority>15), the thread's priority is lowered
 - This is likely a CPU-bound thread, and we need to keep the system interactive
- When a thread “wakes up”, its priority is boosted
 - It is likely an I/O-bound thread
- The boost depends on what the thread was waiting for
 - e.g., if it was waiting on the keyboard, it is definitely an interactive thread and the boost should be large
- These are the same general ideas as the 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 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 bit of a history of scheduler development
- Kernel 1.2: simplicity and speed
 - Round-robin scheduling
 - Implemented with a circular queue
- Kernel 2.2: toward sophistication
 - Scheduling classes
 - real-time, non-preemptable , non-real-time
 - Priorities within classes

Linux Priorities

- Priority scheme:
 - low values means high priority

<u>numeric priority</u>	<u>relative priority</u>		<u>time quantum</u>
0	highest		200 ms
•			
•			
•			
99			
100			
•			
•			
•			
140	lowest		10 ms

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 cannot 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 slice
 - 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 need 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 a long time 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 machines
 - Increasing numbers of cores does not 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 needs 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
 - $5 \times 32 = 160 > 141$ 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 is 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 have $O(1)$

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 “re-compute 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 is $O(1)$ -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. Lovel (Novell Press)

Linux \geq 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 sums 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 \geq 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 is 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 queue 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
- Homework #2 is **Due Tomorrow!**