CPU Scheduling

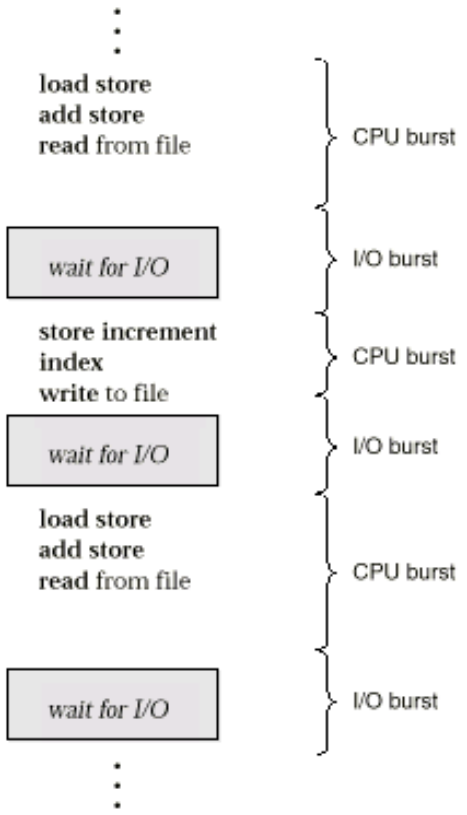
Chapter 5

CPU Scheduling

- The short-term scheduler (A.K.A. the CPU scheduler)
  - Choose a process (thread) from the ready queue
  - May have many to choose from
  - How do we choose?
- We want to:
  - Maximize resource utilization?
  - Provide uniform fractions of CPU time?
  - Keep users happy?
  - Be fair (or at least avoid starvation)?
  - Prevent nuclear meltdown?

How do Processes Behave?

- Processes exhibit a CPU-I/O Burst Cycle
- CPU scheduling determines when the next CPU burst can make progress



The CPU Scheduler

- Choose a ready process and give it the CPU
- When does the CPU scheduler run?
  - A process terminates (system call, relatively infrequent)
  - A process voluntarily yields the CPU (system call, uncommon)
  - A process blocks (system call)
  - A process has had enough CPU time for now (timer interrupt)
  - Another process finishes waiting (hardware interrupt)
- Will the kernel take the CPU from a running process?
  - That's *preemption*
  - With just the first three cases above the scheduler is *non-preemptive*
  - If we include either of the last two, it's *preemptive*

What do we Want from the Scheduler?

- CPU Utilization: how often is the CPU running a user process
- Throughput: how many processes (or CPU bursts) are finished per time unit
- Turnaround time: time from when a job is submitted until its done
  - Or time from when a process becomes runnable to when the next CPU burst completes
- Waiting time: time a process spend in the ready state
- Response time: time from when a job becomes runnable to when it starts producing output
  - So, this depends on the OS and the process itself

Conflicting Criteria

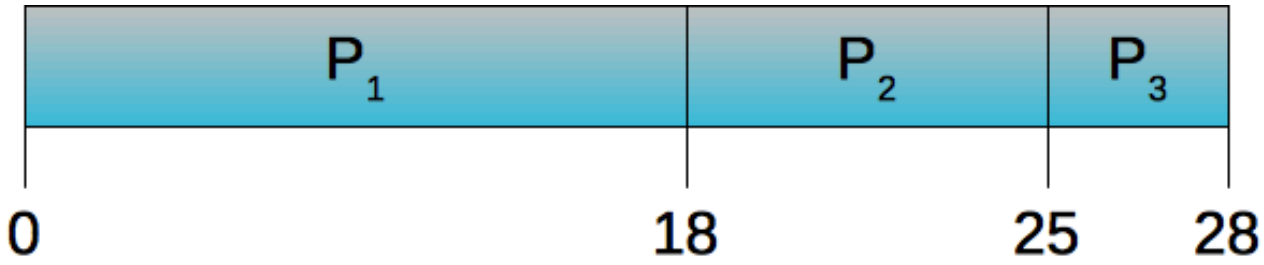
- Minimizing response time will require more context switches (we let a newly runnable process start earlier)
  - Increased scheduling and dispatch overhead
  - So, decreased throughput
- The right scheduling behavior depends on the system
  - Batch vs. interactive, response time is more important to an interactive system
  - Real-time requirements, maybe acceptable response has some real-time bound
  - We don’t expect a generic scheduling algorithm that’s suitable for all systems

First-Come, First-Served (FCFS)

- Ready queue:

Process	Burst Time
P_1	18
P_2	7
P_3	3

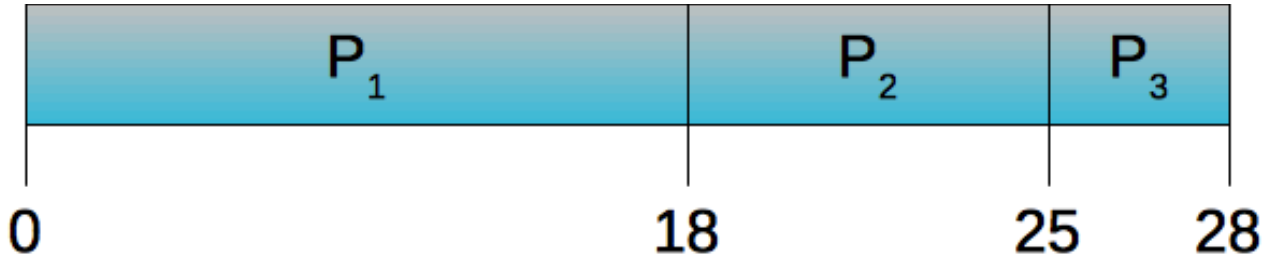
- Schedule processes in the order they arrived (P_1, P_2 then P_3)
- We can draw a Gantt Chart, to illustrate the schedule:



About FCFS

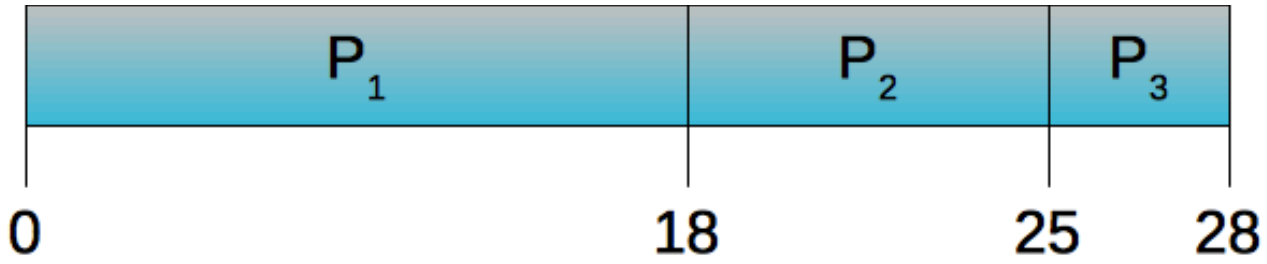
- FCFS is fast (well, it can select a process quickly)
  - Manage the ready queue like a …, well, like a queue.
  - Constant time to append a new process
  - Constant time to select the next runnable process

Is this Schedule Any Good?



- We can check turnaround time
  - For each process: $P_1 = 18$, $P_2 = 25$, $P_3 = 28$
  - Average turnaround time:
    (18 + 25 + 28) / 3 = 23.67
  - Of course, a lot of this time isn't the scheduler's fault.

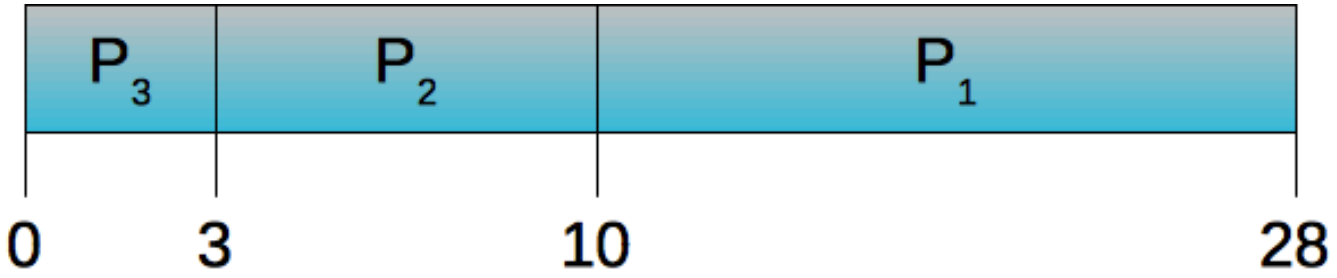
Is this Schedule Any Good?



- We can check waiting time
  - Waiting time for each process: $P_1 = 0$, $P_2 = 18$, $P_3 = 25$
  - Average waiting time: (0 + 18 + 25) / 3 = 14.33
  - Seems like waiting time might depend on our scheduling choices.

FCFS, Alternative Arrival Order

- What if processes were queued as P_3 , P_2 , P_1 ?



- Now, waiting time $P_1 = 10$, $P_2 = 3$, $P_3 = 0$
- Average waiting time: (10 + 3 + 0) / 3 = 4.33
- Much better than previous ordering
- FCFS can lead to the *convoy effect:* short processes behind long processes

A Provably Optimal CPU Scheduler

- Is it better to run a shorter CPU burst or a longer one first?
- It's like the checkout line at the grocery store
- Running shorter bursts first is provably optimal
  - With respect to what?
  - Average waiting time and turnaround time

Shortest Job First (SJF) Scheduling

- SJF, always choose the process with the shortest CPU burst
- Is it preemptive?
  - Nope, SJF is non-preemptive
  - But, its good friend, Shortest-Remaining-Time First (SRTF) is
    - Always run the process with the least time remaining in its CPU burst
    - May preempt a longer burst if a shorter one arrives in the ready queue

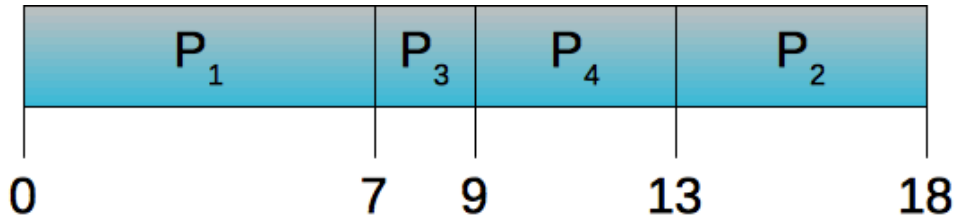
Example of SJF (non-preemptive)

Process	Burst Time	Arrival Time
P_1	7	0
P_2	5	2
P_3	2	3
P_4	4	8

Example of SJF (non-preemptive)

Process	Burst Time	Arrival Time
P_1	7	0
P_2	5	2
P_3	2	3
P_4	4	8

- Schedule:



- Waiting Time: (0 + 11 + 4 + 1) / 4 = 4

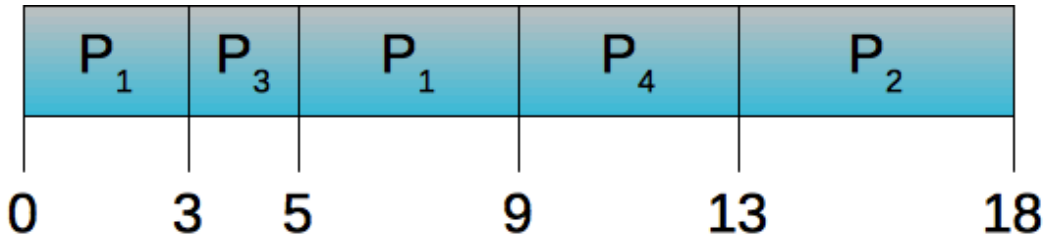
Example of SRTF (preemptive)

Process	Burst Time	Arrival Time
P_1	7	0
P_2	5	2
P_3	2	3
P_4	4	8

Example of SRTF (preemptive)

Process	Burst Time	Arrival Time
P_1	7	0
P_2	5	2
P_3	2	3
P_4	4	8

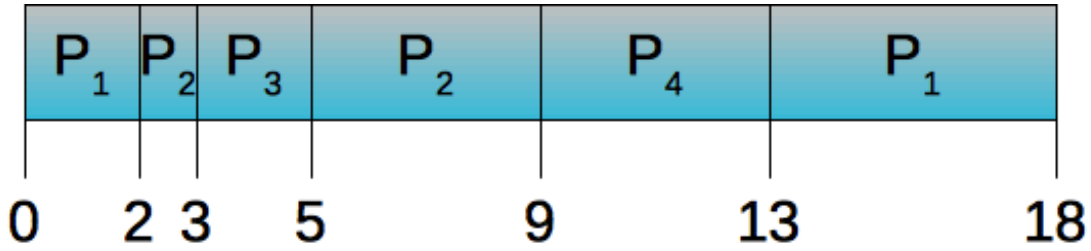
- Oops, there's a tie.
- If you stay with the running process:



Example of SRTF (preemptive)

Process	Burst Time	Arrival Time
P_1	7	0
P_2	5	2
P_3	2	3
P_4	4	8

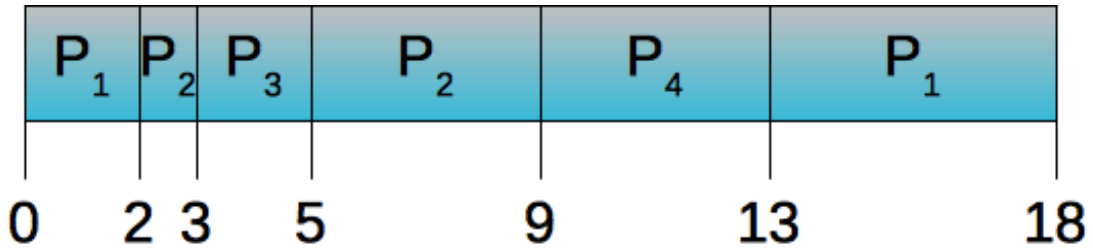
- If we switch to the new process:



Example of SRTF (preemptive)

Process	Burst Time	Arrival Time
P_1	7	0
P_2	5	2
P_3	2	3
P_4	4	8

- Schedule:



- Waiting Time: (11 + 2 + 0 + 1) / 4 = 3.5

Guessing the Future

- So, let's implement SJF / SRTF and we're done, right?
  - Not so easy, requires knowledge of the future.
  - Also, there's a risk of *starvation.*
- OS must estimate next CPU burst length
- How are we going to do that?

Guessing the Future

- How about this, we use the previous CPU burst length.
- Or let's use several of them
- Averaged together?
  - Easy to compute.
  - But a bad idea, process behavior can change
- Exponential average of previous burst lengths.
  - Great, it will adapt to reflect recent history.
  - Also, easy to compute. It only requires a little extra storage per process.

Exponential Average

- t_n : actual length of n^{th} CPU burst
- τ_n : previous (exponential) average burst length
- Want to compute τ_{n+1}, predicted next burst length
  - $\tau_{n+1} \leftarrow \alpha\, t_n + (1 - \alpha)\, \tau_n$
  - For parameter α, with $0 < \alpha < 1$
- Effectively, we get a weighted average of all previous bursts.
  - Expand it and see:
    $\alpha\, t_n + \alpha(1 - \alpha)\, t_{n-1} + \alpha(1 - \alpha)^2\, t_{n-2} + \alpha(1 - \alpha)^3\, t_{n-3} \ldots$
  - With $\alpha = 0.5$, that's:
    $0.5\, t_n + 0.25\, t_{n-1} + 0.125\, t_{n-2} + 0.0625\, t_{n-2} \ldots$
  - If α is close to 1, favor recent history
  - If α is close to 0, more weight to older bursts

Approximating SJF / SRTF

- So, SJF and SRTF are good for average wait time
- But they are impossible to implement
- So, we must approximate them by observing process behavior
  - In general, this is going to be a reusable technique.
  - Watch process behavior to differentiate their scheduling

Priority Scheduling

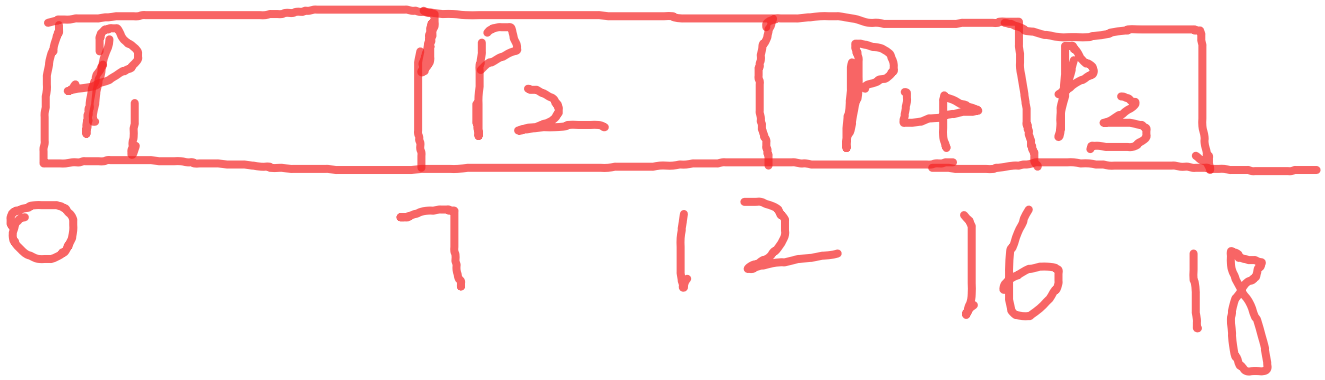
- The idea, give preference to more important jobs
- A priority number (integer) associated with each process
  - Pretend smallest number ⇒ highest priority
  - But this varies from system to system ☹
- CPU Scheduler chooses the highest-priority process
  - Could be preemptive
  - Or, non-preemptive would make sense also

Priority Scheduling

- Users (e.g., a system administrator) could set priority values
- Or the OS could compute them
  - Priority values may be static or dynamic
  - For example, approximate SJF is just priority scheduling based on predicted burst length
- Potential problems with priority?
  - Starvation – low priority processes may never execute
  - Solution: *aging* – as time progresses increase the priority of the process

Example of Priority, Non-Preemptive

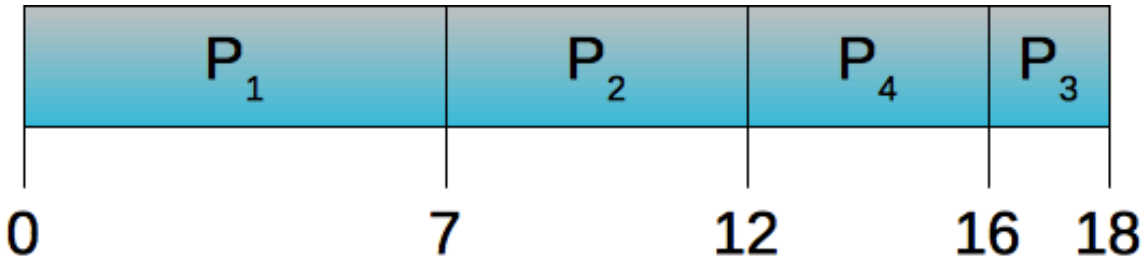
Process	Burst Time	Arrival Time	Priority
P_1	7	0	8
P_2	5	2	5
P_3	2	3	6
P_4	4	8	2



Example of Priority, Non-Preemptive

Process	Burst Time	Arrival Time	Priority
P_1	7	0	8
P_2	5	2	5
P_3	2	3	6
P_4	4	8	2

- Schedule:

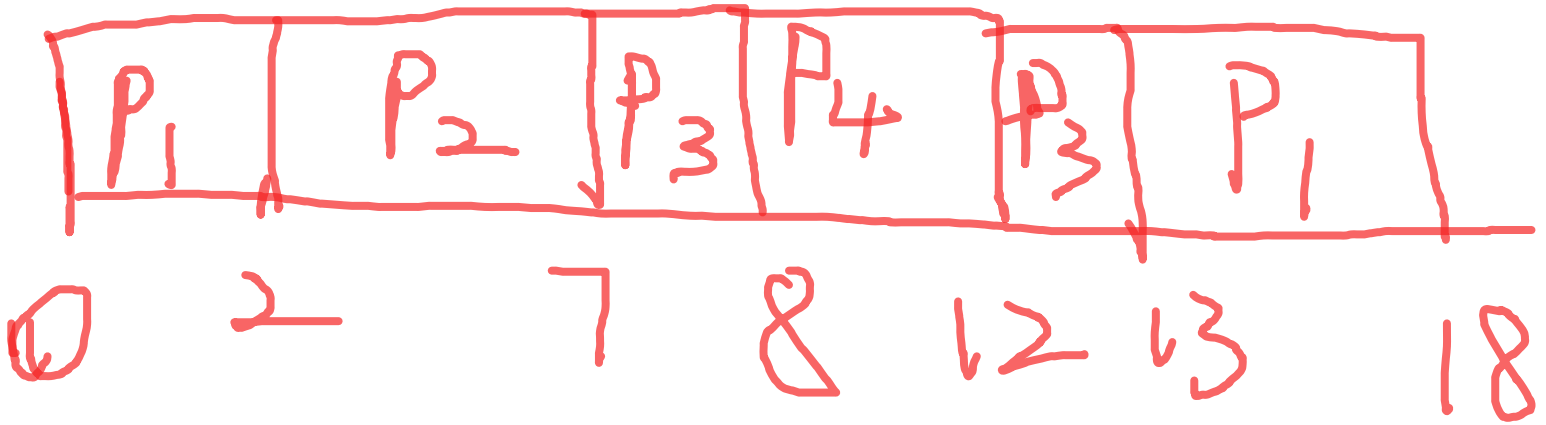


- Waiting Time: (0 + 5 + 13 + 4) / 4 = 5.5

Example of Priority, Preemptive

Process	Burst Time	Arrival Time	Priority
P_1	7 5	0	8
P_2	5	2	5
P_3	2 1	3	6
P_4	4	8	2

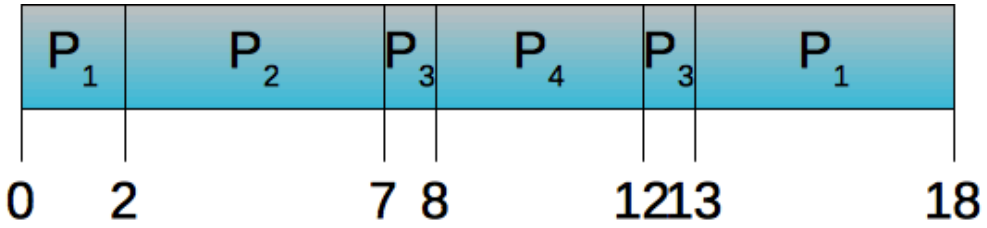
5<8
5<6
6<8
2<6



Example of Priority, Preemptive

Process	Burst Time	Arrival Time	Priority
P_1	7	0	8
P_2	5	2	5
P_3	2	3	6
P_4	4	8	2

- Schedule:



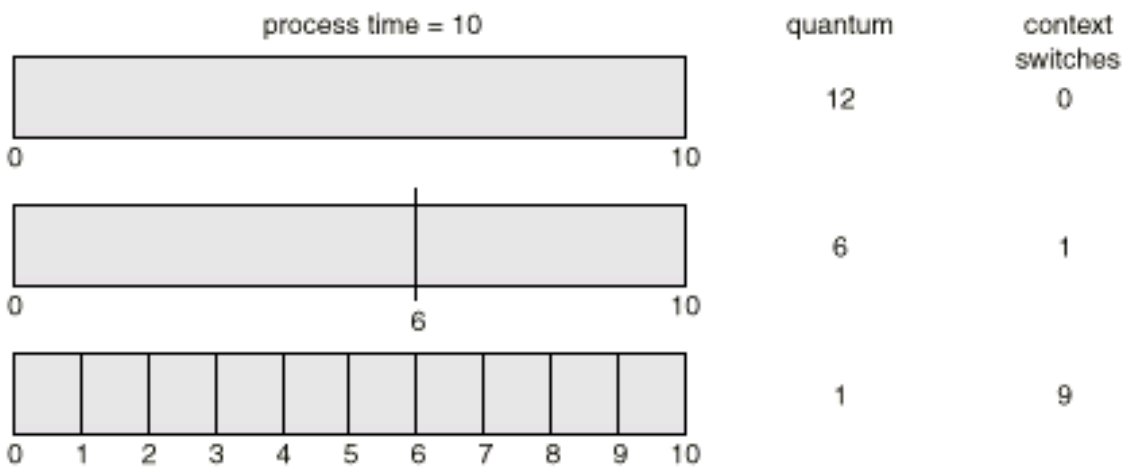
- Turnaround Time: (18 + 5 + 10 + 4) / 4 = 9.25
- Waiting Time: (11 + 0 + 8 + 0) / 4 = 4.75

Round Robin (RR) Scheduling

- Make runnable processes take turns on the CPU
  - A CPU time limit, the *time quantum*
  - Enforced via the timer interrupt
  - Typically, 10-100 milliseconds
- If there are *n* CPU-bound processes with a quantum *q*
  - Each gets *1/n* of the CPU time (in *q*-length blocks)
  - Wait time is bounded, at most *(n-1)q*
- Typically, higher average turnaround time than SJF or SRTF, but lower response time

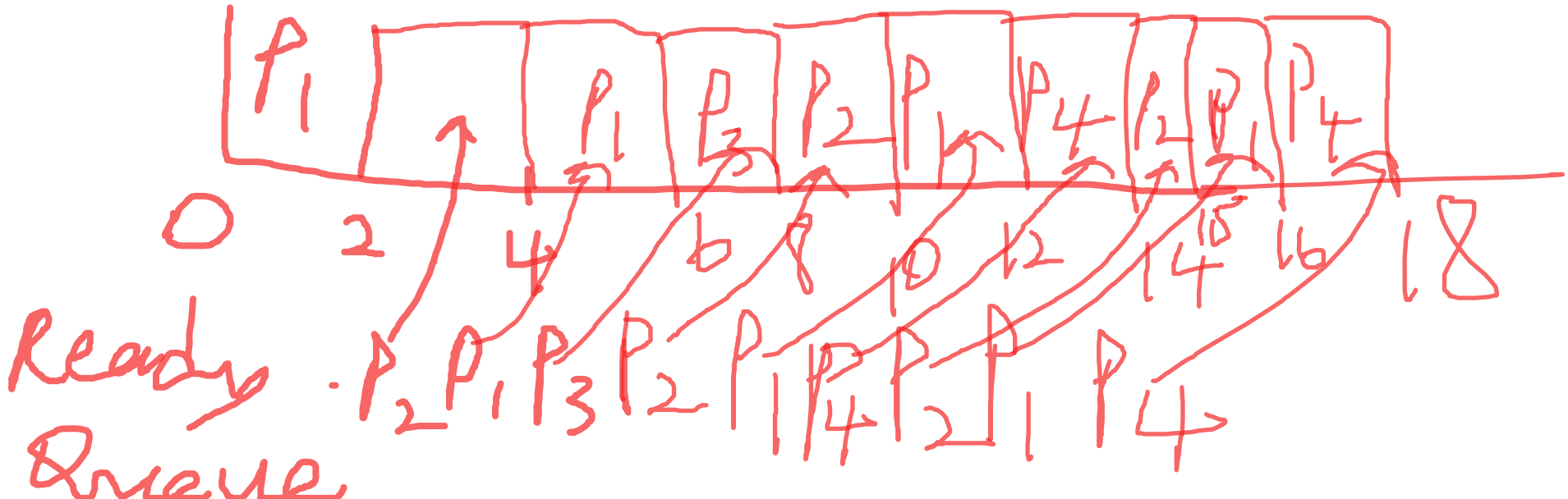
Round Robin

- Options for quantum length
  - Trade off time vs. overhead
  - Long quantum ⇒ behaves more like FCFS
  - Short quantum
    - More uniform CPU service (like your own CPU 1/n as powerful)
    - Better response time
    - More overhead



Example of Round Robin, q = 2

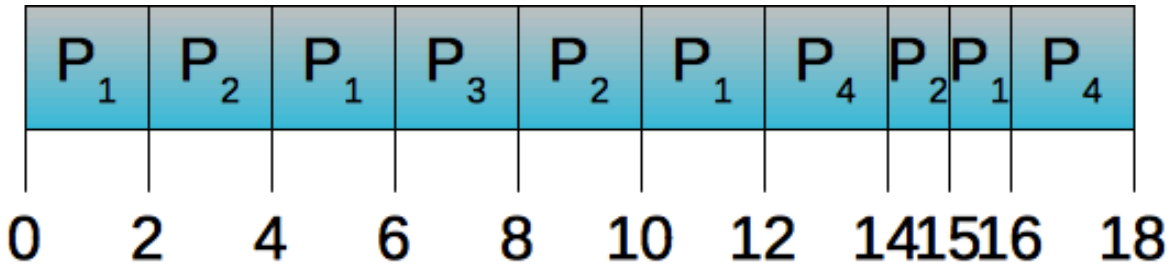
Process	Burst Time	Arrival Time	Priority
P_1	7	0	8
P_2	5	2	5
P_3	2	3	6
P_4	4	8	2



Example of Round Robin, q = 2

Process	Burst Time	Arrival Time	Priority
P_1	7	0	8
P_2	5	2	5
P_3	2	3	6
P_4	4	8	2

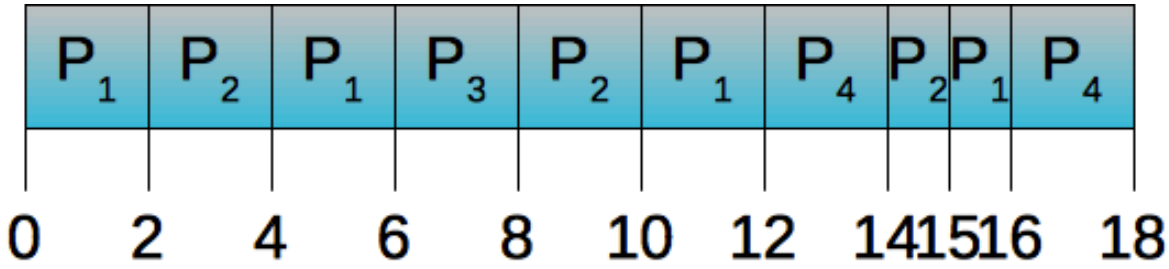
- Schedule:



Example of Round Robin, q = 2

Process	Burst Time	Arrival Time	Priority
P_1	7	0	8
P_2	5	2	5
P_3	2	3	6
P_4	4	8	2

- Schedule:



- Turnaround Time: (16 + 13 + 5 + 10) / 4 = 11
- Waiting Time: (9 + 8 + 3 + 6) / 4 = 6.5

Fun Scheduling Exercises

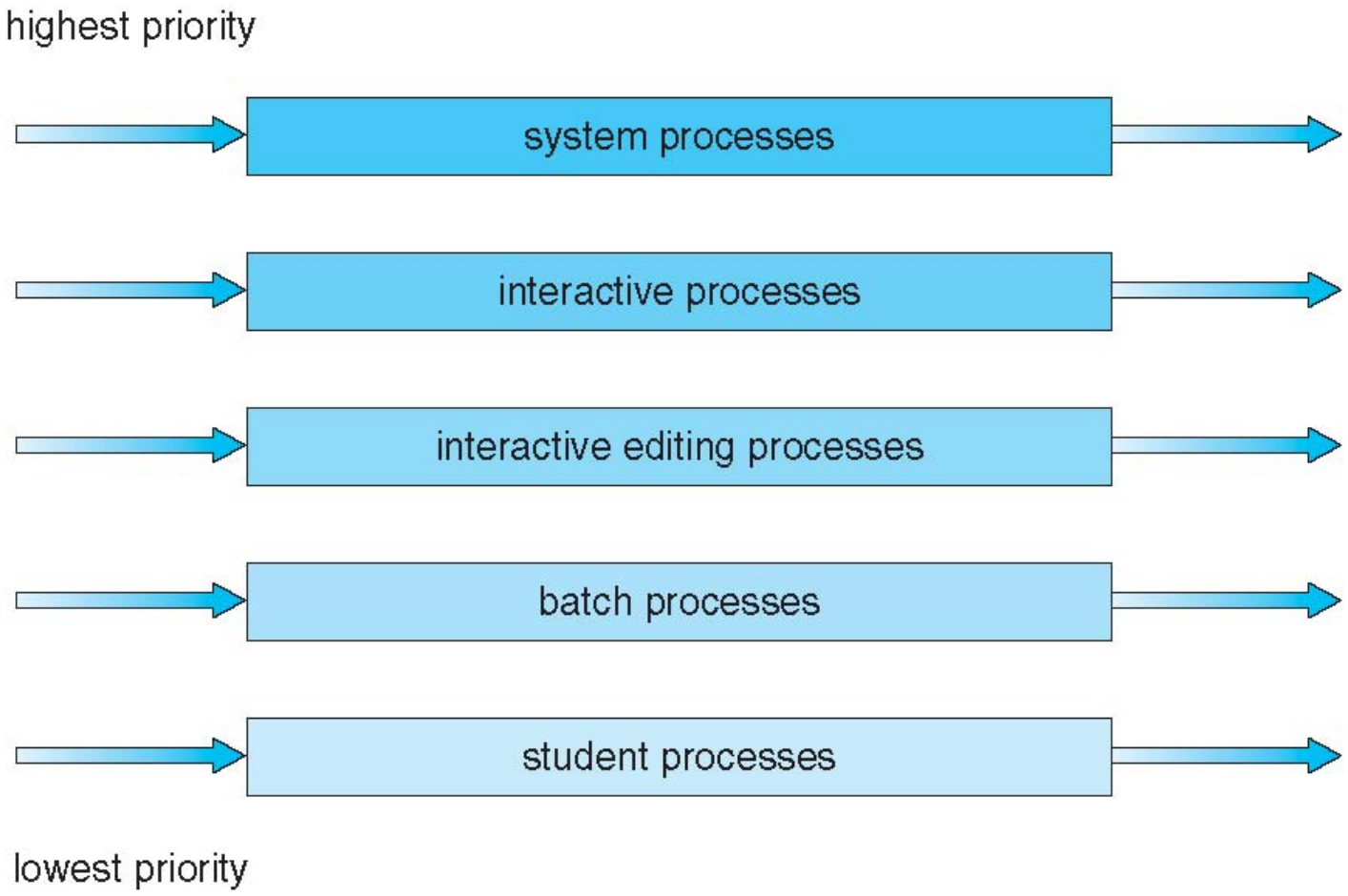
Process	Burst Time	Arrival Time	Priority
P_1	5	0	6
P_2	7	3	4
P_3	4	5	8
P_4	6	6	5

- Try it with:
  - FCFS (easy)
  - SJF (still easy)
  - Preemptive Priority (a little harder)
  - RR with q = 4 (fun!)

Multilevel Queue Scheduling

- The idea: use different schedulers for different kinds of jobs
- Ready queue is partitioned into multiple queues, each with a different scheduler
- Example:
  - Foreground, interactive jobs, Round Robin
  - Background, batch jobs, FCFS
- What if we have jobs in each queue?
  - We have to schedule between the queues
  - Fixed priority scheduling (e.g., always favor foreground)? Risk of starvation
  - Time slice (e.g., 80% to foreground, 20% to background)

Multilevel Queue Example



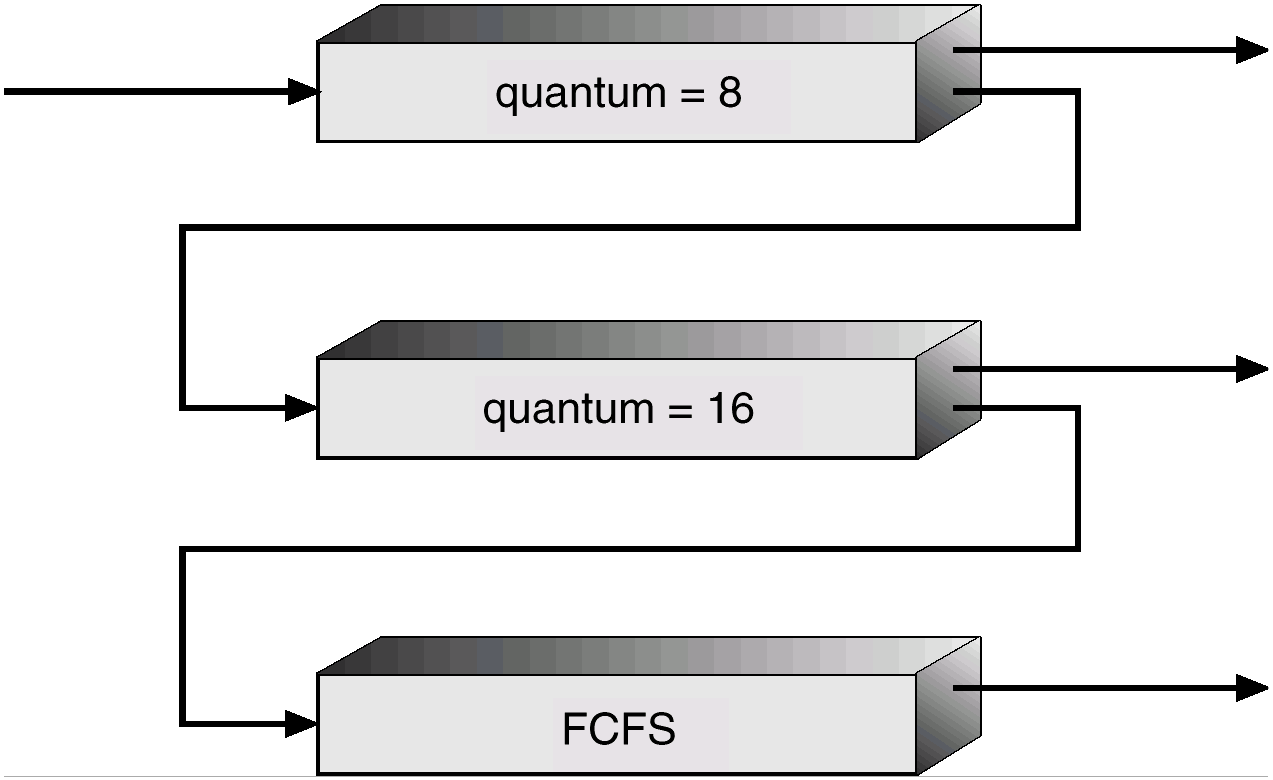
Multilevel Feedback Queue (MLFQ)

- MLFQ scheduler was first described by Corbato et al. in 1962. Turing Award in 1990.
- *Feedback*: A process can move among the queues
- Goal:
  - Optimize turnaround time without priori knowledge
  - Optimize response time for interactive users

Multilevel Feedback Queue Example

- Three queues
  - Q_0 : RR with a time quantum of 8 milliseconds
  - Q_1 : RR with a time quantum of 16 milliseconds
  - Q_2 : FCFS (with no time bound)
- Scheduling
  - Queues are selected via priority
  - A ready process always enters Q_0 (maybe it will finish quickly)
  - If it uses its whole quantum, move to Q_1
  - If it uses the whole 16 ms quantum, move to Q_2
  - Priority Boost: after some time period S, move all the processes in the system to the topmost queue.

Multilevel Feedback Queue Example



Multi-Processor Scheduling

- What can we do with all these CPUs or CPU cores?
- Are the cores all equivalent?
  - That's a *homogeneous* multi-processor system
  - vs. a *heterogeneous* multi-processor system
- We could let one processor manage the ready queue and other OS structures
  - That's *asymmetric multiprocessing*
  - Avoids hazards of concurrent modification of OS structures
- Alternatively, we could let every processor schedule itself
  - That's *symmetric multiprocessing (SMP)*
  - Processors must coordinate modification of shared kernel structures

I Like my CPU

- Recall, there are per-processor components to improve execution speed
  - On-package cache
  - Other stuff we'll discover later
- A smart OS will:
  - Try to keep a process on the same CPU
  - That's *soft processor affinity*
- An OS may provide mechanisms to:
  - Stipulate that a thread will always run on the same core
  - That's *hard processor affinity*

Thread Scheduling

- Remember
  - User-level threads, programmer can make scheduling decisions
  - Kernel threads, OS makes scheduling decisions
- *Contention Scope*
  - Other threads you competes with for CPU time
  - *Process contention scope*: just compete with other threads in the same process
  - *System contention scope*: compete with threads in other processes
- Pthreads lets us specify this

Pthread Scheduling

- API lets us specify contention scope
  - Part of the thread attribute structure
  - PTHREAD_SCOPE_PROCESS : schedules threads using process contention scope
  - PTHREAD_SCOPE_SYSTEM : schedules threads using system contention scope

```
pthread_attr_t attr;
/* get the default attributes */
pthread_attr_init(&attr);
/* choose contention scope */
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
/* set the scheduling policy - FIFO, RT, or OTHER */
pthread_attr_setschedpolicy(&attr, SCHED_OTHER);
/* create a thread */
pthread_create(&tid[i],&attr,runner,NULL);
```

- But we're always just using system scope

Scheduling Examples

- Scheduling behavior from different operating systems
  - Solaris
  - Windows XP
  - Linux
- Read Chapter 5.7 if you are interested, but it's not required.

What We've Learned

- Criteria for CPU scheduling
- Survey of CPU schedulers
- Be the scheduler
- Multi-level and multi-core scheduling