CPU Scheduling

CS201 Lecture 5

Jason Hibbeler
University of Vermont

Spring 2019

Motivation

The CPU is the heart of the system

- or maybe the brains of the system
- the heart and the brains

One of the central design goals of the OS:

- keep the CPU busy as much of the time as possible

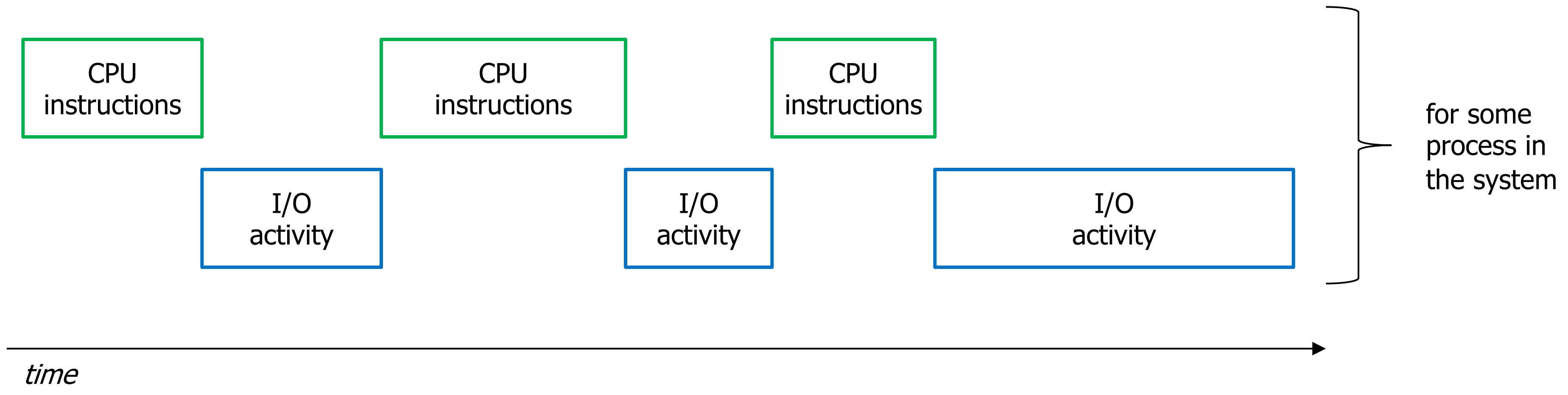
Question: for my program, when is the CPU not busy?

- what should the OS do about this?

The Multiprogrammed System

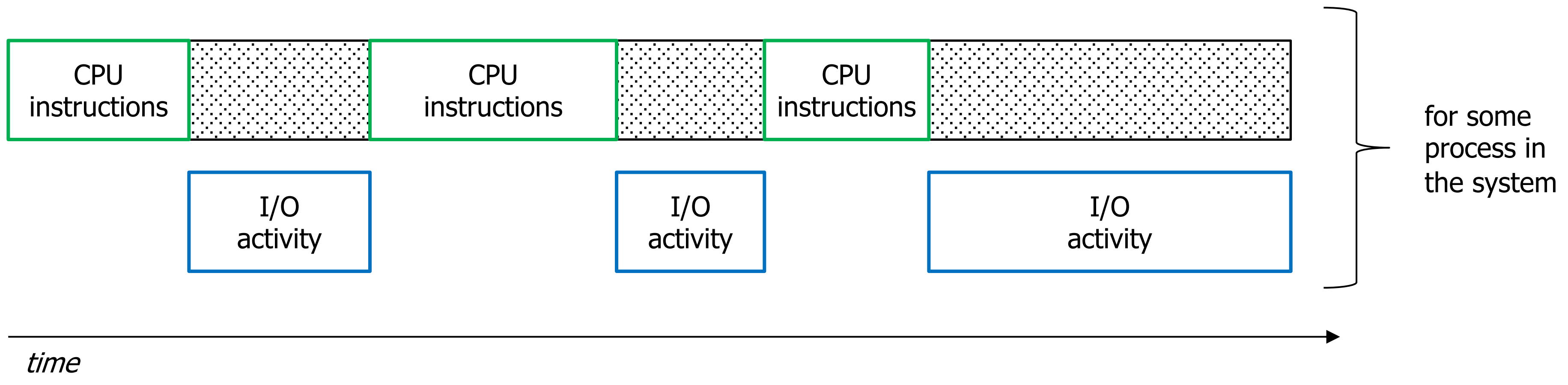
Many jobs are running concurrently in the system

- they have a period of CPU activity followed by I/O activity
- each bit of CPU activity is called a *CPU burst*; each bit of I/O activity is called an *I/O burst*



run for a while; then do I/O; then run for a while; then do I/O

The Multiprogrammed System

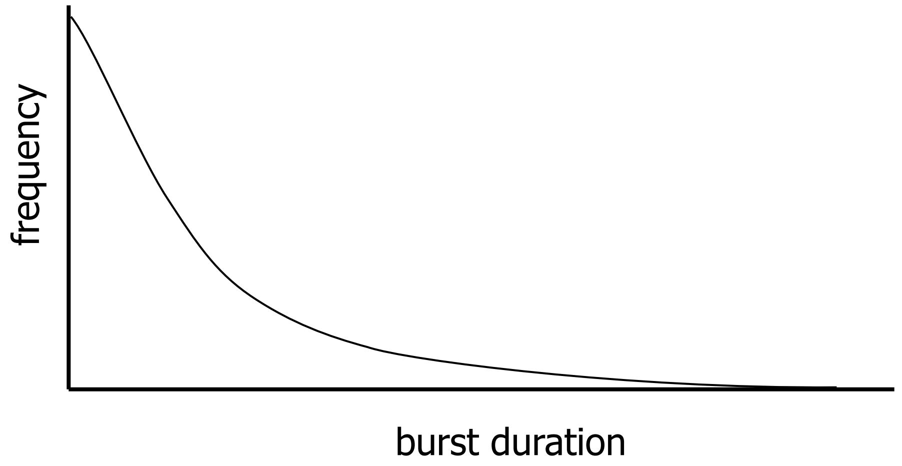


The key question: what should happen when this process is not using the CPU (the dotted regions)?

Burst Times

We can model the distribution of CPU and I/O burst times in a system

- one model has an exponential distribution for CPU burst times: lots of short burst times and few long burst times
- we must distinguish between *I/O-bound* and *CPU-bound* processes



CPU-Bound vs. I/O-Bound

CPU-bound process

- the process spends most of its time doing computation (on the CPU)
- think of a complex calculation

I/O-bound process

- the process spends most of its time doing I/O
- example: copying a file

CPU Scheduler

The short-term scheduler (CPU scheduler) selects a process that is in the ready queue and dispatches it to the CPU

- important point: the ready queue is not necessarily a FIFO queue

CPU Scheduling Criteria

The CPU scheduler can make its decisions based on several different criteria. Let's discuss.

- think of a grocery store, with customers and cashiers

What is an important criterion—a condition we want the system to satisfy?

- think of the grocery store: what's a condition we want to satisfy?

The Grocery Store

who should go first?

CPU Scheduling Criteria

1. CPU utilization

- Keep the CPU as busy as possible. Values can range from 0% to 100%.
- In practice, it will probably range between 40% and 90%.

```
$  uptime
 11:01:38 up 404 days,  3:00,  1 user,  load average: 0.00, 0.02, 0.00
```

`uptime` shows the load average: essentially, the average number of jobs that want to use the CPU, averaged over the past 1 minute, 5 minutes, and 15 minutes

CPU Scheduling Criteria

2. Throughput

- Enable processes to make progress.
- One measure of throughput is #processes that are completed per time unit.
- Somewhat hard to analyze, since there will be a mix of short-running and long-running processes in the system.

CPU Scheduling Criteria

3. Turnaround time

- Ensure that processes submitted to the system can actually finish in a reasonable time.
- The turnaround time for a process is the total time that it is in the system, from start to finish
- This includes CPU activity, I/O activity, and time spent waiting.

CPU Scheduling Criteria

4. Waiting time

- The total amount of time for CPU or I/O for a process is an intrinsic characteristic of the process itself—the CPU scheduler cannot change that.
- However, the CPU scheduler can determine the amount of time that a process spends sitting in the ready queue.
- The waiting time is the total time spent in the ready queue.

CPU Scheduling Criteria

5. Response time

- For interactive processes, the turnaround time isn't a particularly meaningful measure.
- The response time is the time from when a request is made to when the first response is produced.

CPU Scheduling Criteria

A good design goal for the CPU scheduler:

- minimize turnaround time, waiting time, and response time
- maximize CPU utilization and throughput

CPU Scheduling Criteria

However, we might want to bound the variance instead:

- optimize the min or max values of some or all of the quantities

Especially for interactive systems:

- minimizing the variance in response time leads to a more predictable system.

Preemptive vs. Nonpreemptive Scheduling

If the OS is allowed to grab the CPU from a process only when the process terminates or when it moves to the waiting state (e.g., for I/O)

- then the system is **nonpreemptive**

Preemptive vs. Nonpreemptive Scheduling

Otherwise, in **preemptive** scheduling:

- the OS can grab the CPU from a process when it wants
- for example if the process has already used up its allocated time slice on the CPU

Preemption makes life difficult for the OS.

Preemptive Scheduling

Suppose the OS is executing instructions in kernel mode on behalf of a user process.

- what happens if the OS is preempted at this point?
- if the OS is executing a system call and modifying kernel data and the preemption occurs, then the kernel must carefully clean up and leave the kernel data is a coherent state
- one way to do this is to prevent preemption while the OS is executing a system call
- this won’t work so well for real-time systems

Dispatcher

The dispatcher is the OS component that manages the assignment of processes to the CPU:

1. switches context (saves the PCB of current process then loads the PCB from the next process that will run)
2. switches to user mode
3. jumps to the proper location in the target program and reloads registers from the PCB

Dispatcher

The time that the dispatcher uses (called dispatch latency) is overhead in the system

- no process is making progress during dispatcher activities

Scheduling Algorithms

Simplest scheme is first-come-first-served (FCFS)

- processes are dispatched in the order in which they request the CPU
- easily managed with a FIFO queue

Seems reasonable; any potential drawbacks?

- think of the grocery store

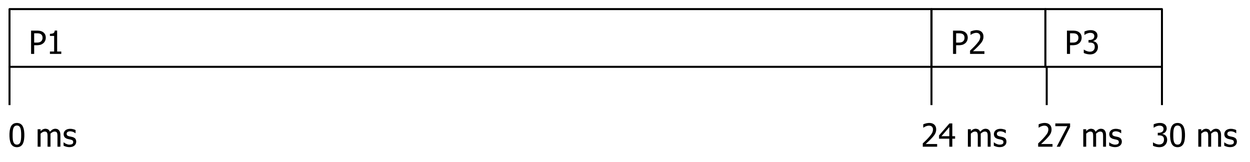
FCFS

Suppose that three processes arrive in the system in this order:

Process	Burst time (ms)
P1	24
P2	3
P3	3

burst time: the length of time that a process will want to use the CPU

ms = milliseconds



What's the average waiting time?

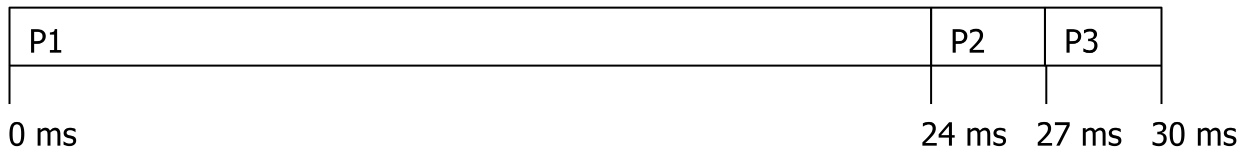
FCFS

Suppose that three processes arrive in the system in this order:

Process	Burst time (ms)
P1	24
P2	3
P3	3

burst time: the length of time that a process will want to use the CPU

ms = milliseconds



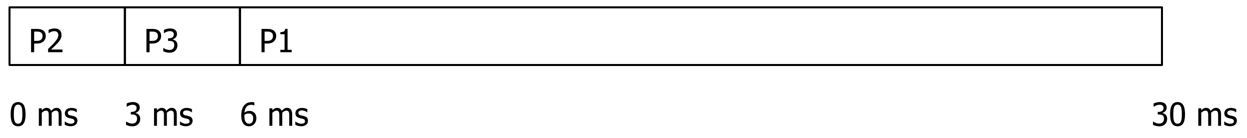
What's the average waiting time?

- (0 + 24 + 27) / 3 = 17 ms

FCFS

Suppose we changed the order in which we dispatch the processes:

same three processes, just in a different order!



What’s the mean waiting time?

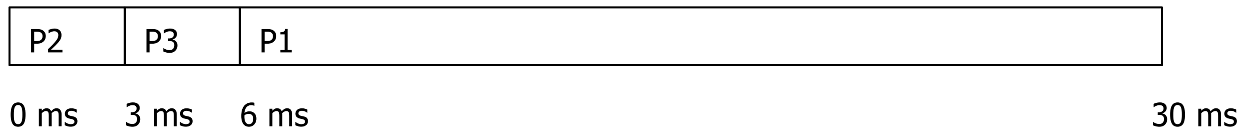
Conclusion: mean waiting time under FCFS can vary greatly

- this could be a concern in the operation of the system
- because in general, we want predictable behavior

FCFS

Suppose we changed the order in which we dispatch the processes:

same three processes, just in a different order!



What's the mean waiting time?

- (0 + 3 + 6) / 3 = 3 ms

Conclusion: mean waiting time under FCFS can vary greatly

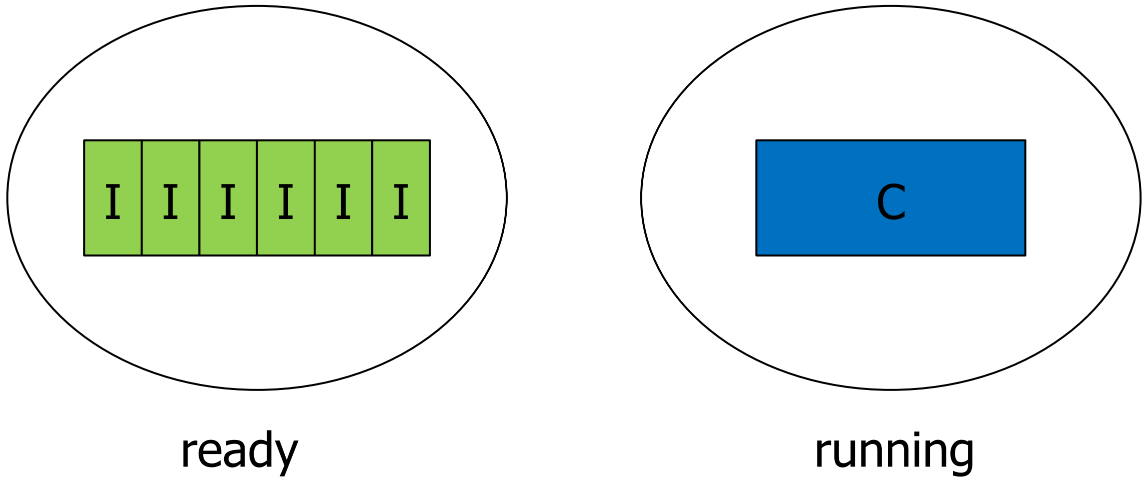
- this could be a concern in the operation of the system
- because in general, we want predictable behavior

FCFS

This profile is characteristic of a job mix in which there is one CPU-bound job and many I/O-bound jobs

- the I/O-bound jobs each use a little CPU and then move to the I/O queue
- the CPU-bound job then gets the CPU and holds it for a while; for so long that that the I/O-bound jobs finish their I/O and move back to the ready queue
- the CPU-bound job then makes an I/O request and moves off the CPU
- the I/O-bound jobs then quickly finish their CPU activity while the CPU-bound job is doing its I/O, and they move back to an I/O queue
- and then the CPU-bound job gets the CPU again
- this leads to periods of inactivity of both the CPU and the I/O devices
- as described, this is a nonpreemptive scheme
- this is called the **convoy effect**

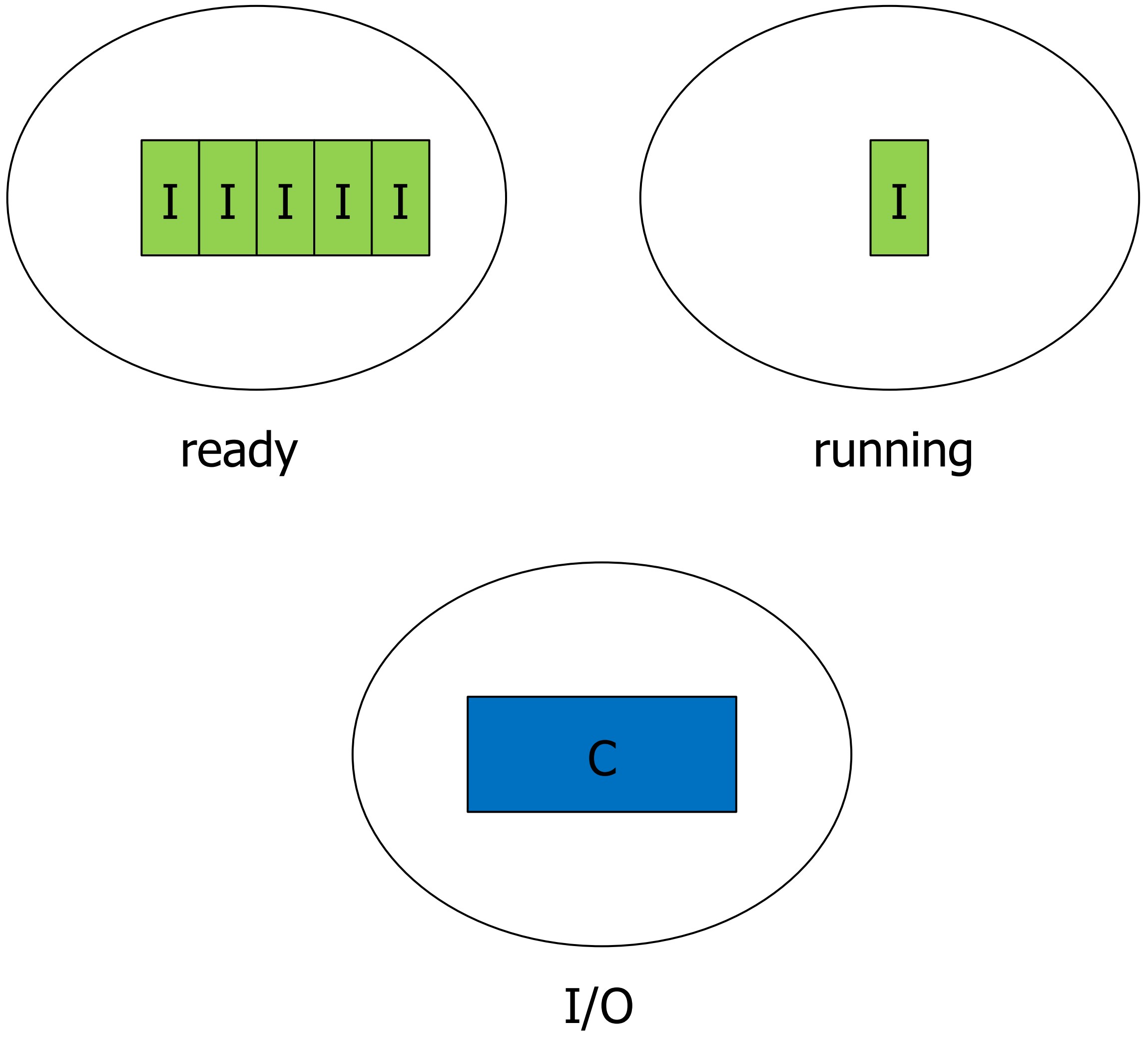
Convoy Effect



process C is running; the six I processes are waiting for the CPU

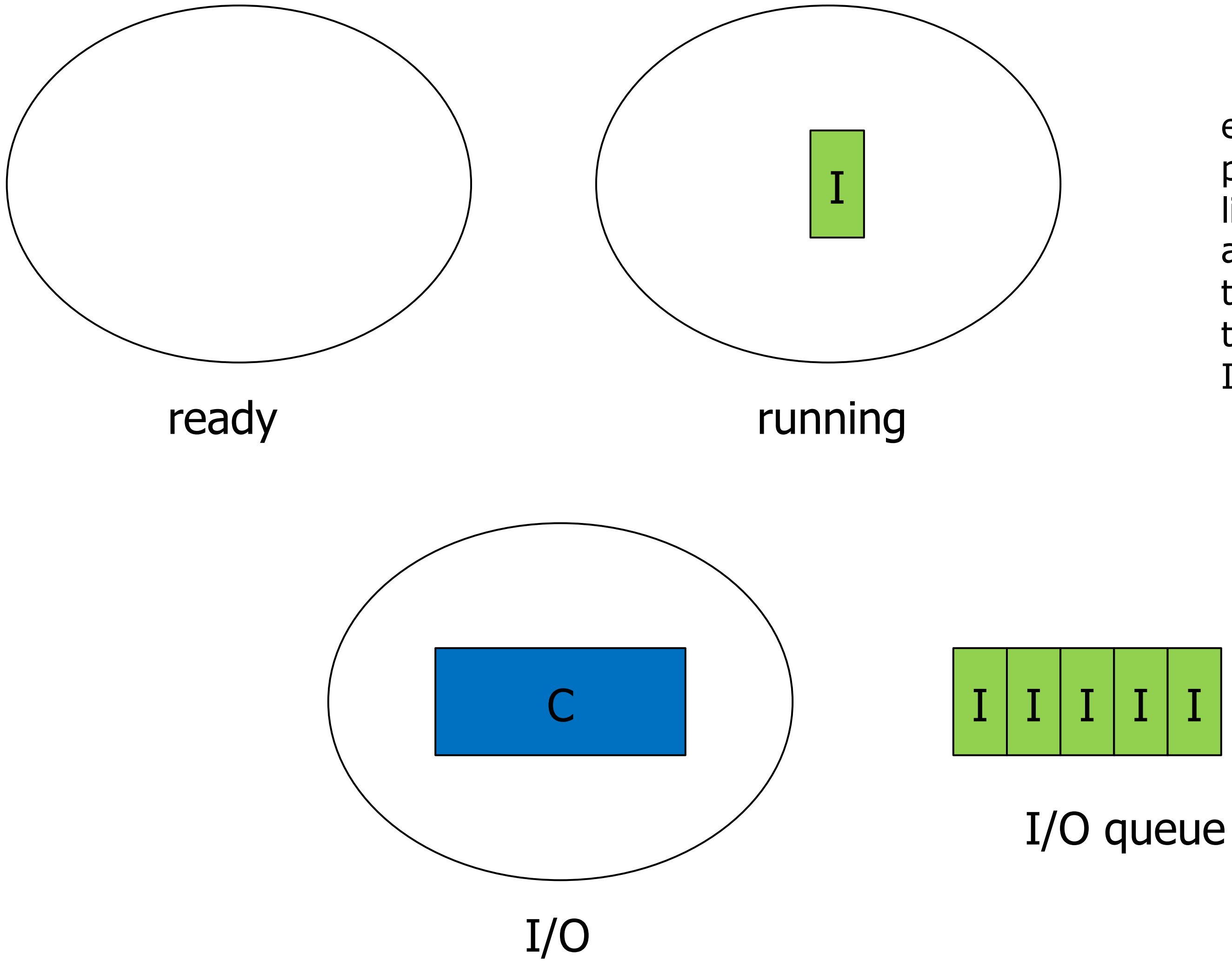
Here is one CPU-intensive process (C) and six I/O-intensive processes (I)

Convoy Effect



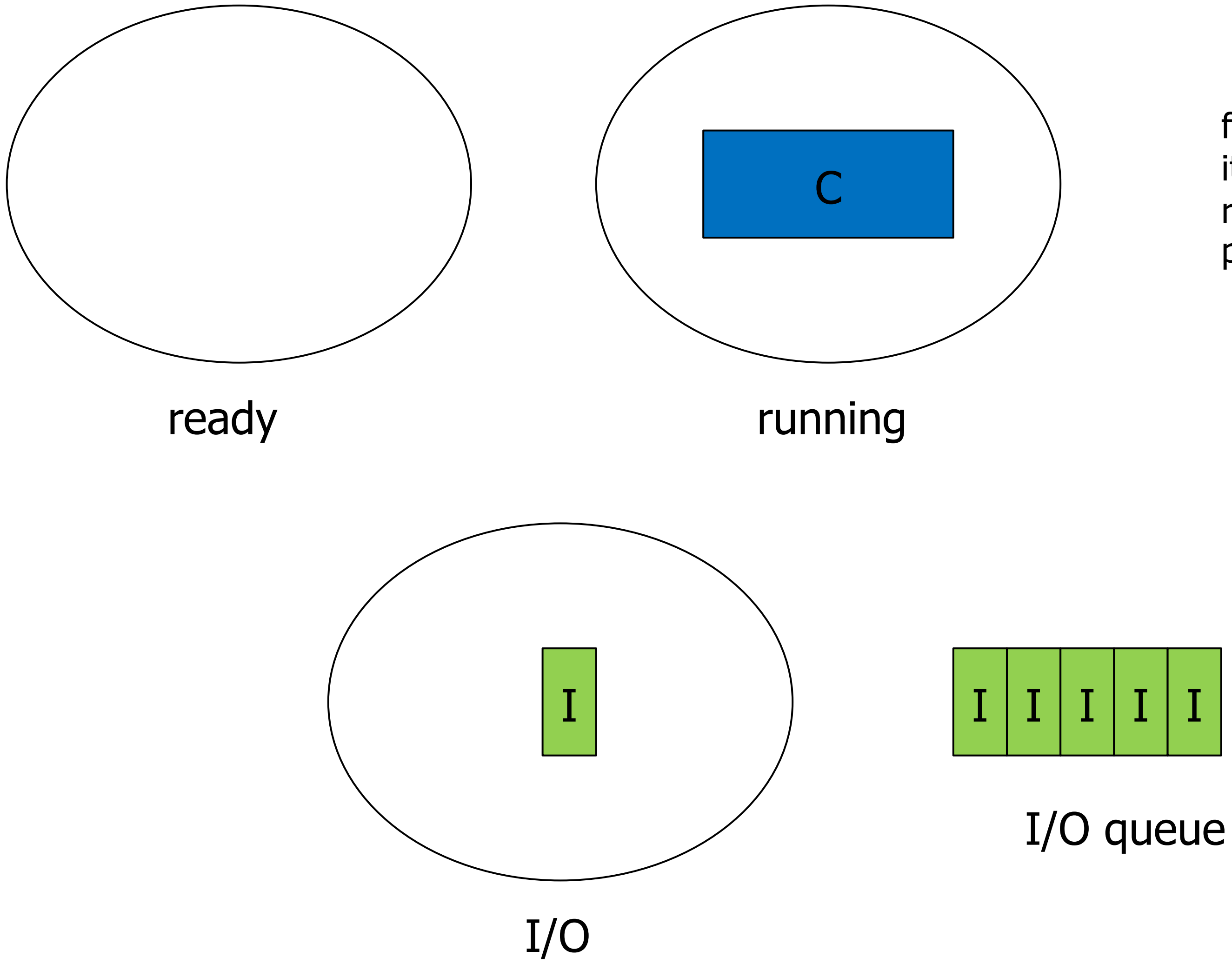
process C makes an I/O request; the first of the six I processes gets the CPU

Convoy Effect



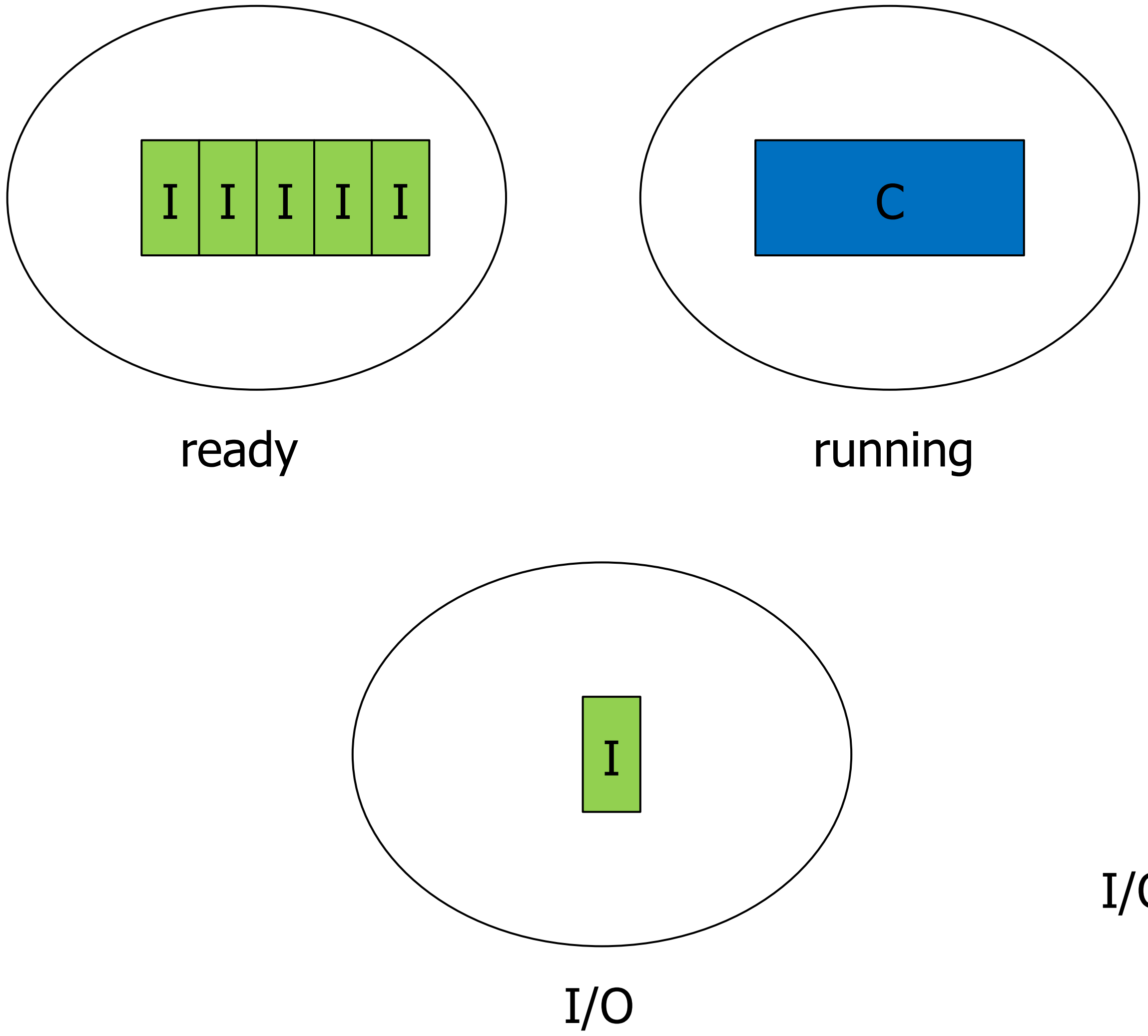
each of the six I processes needs only a little CPU before making an I/O request; when this happens, they have to wait for C to finish its I/O

Convoy Effect



finally, C finishes its I/O; it goes to ready and then running state; the six I processes do their I/O

Convoy Effect



each of the six I processes finishes its I/O and then goes back to the ready state to wait until C is done with the CPU

so most of the time, the I processes are just sitting around

I/O queue

Shortest-Job First (SJF)

More accurately stated, this is "shortest-next-CPU-burst-first", but SJF is the conventional name.

Simple concept: give the CPU to the job that has the shortest next CPU burst time (use FCFS to break ties)

Process	**Next burst time duration (ms)**
P1	6
P2	8
P3	7
P4	3

here's the ready queue

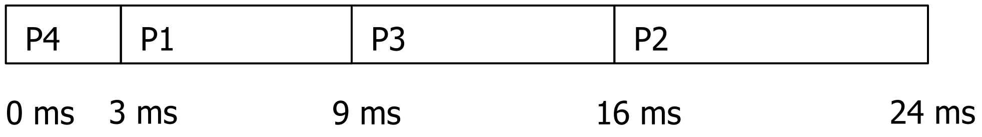
Shortest-Job First (SJF)

Example:

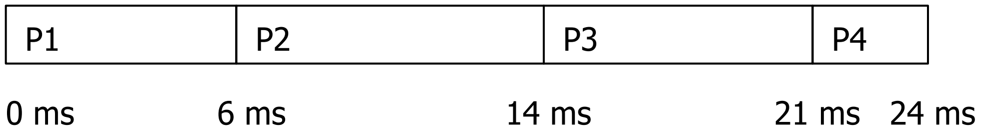
Process	Next burst time duration (ms)
P1	6
P2	8
P3	7
P4	3

observations?

SJF



FCFS



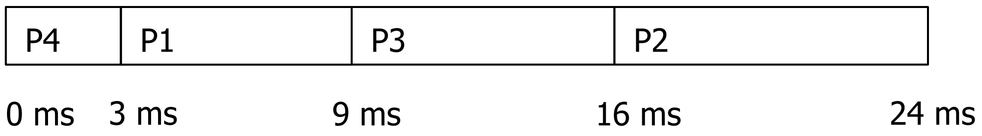
Shortest-Job First (SJF)

Example:

Process	Next burst time duration (ms)
P1	6
P2	8
P3	7
P4	3

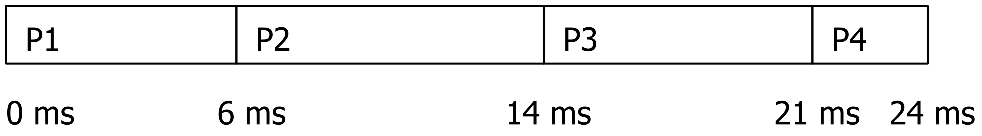
observations?

SJF



mean waiting time:
(0+3+9+16)/4 = 7 ms

FCFS



mean waiting time:
(0+6+14+21)/4 = 10.25 ms

Shortest-Job First (SJF)

SJF is provably optimal in terms of minimizing the mean waiting time.

So, this sounds even better.

What do you think about this scheme?

Shortest-Job First (SJF)

SJF minimizes waiting time, but what about fairness?

- lots of short-duration processes could crowd out a longer process

Shortest-Job First (SJF)

SJF minimizes waiting time, but what about fairness?

- lots of short-duration processes could crowd out a longer process
- also, how do we know how long each process will need for its next CPU burst?

Estimating Next Burst Time

The tricky thing for the dispatcher is knowing what the next burst time for each process actually is.

We can do something simplistic

- estimated time of next CPU burst is the same as the time of the previous CPU burst

Or, something more sophisticated:

- use an exponential moving average

Estimating Next Burst Time

Let t_n be the length of the n^{th} CPU burst (in the past)

Let τ_{n+1} be the predicted value of the next CPU burst (in the future)

take τ_0 to be some fixed value (a typical burst time)

Then:

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

Estimating Next Burst Time

We will use

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

In other words: our prediction for the next burst time for a particular process is a weighted average

- of the most recent burst time
- and the past history of burst times

Estimating Next Burst Time

We will use

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

The idea here is that we have a moving average of the most recent burst and the history of previous bursts

- α=1: only the most recent CPU burst is taken into account
- α=0: the most recent CPU burst is disregarded
- α=½: a nice compromise

Shortest-Job First (SJF)

Important consideration:

- whether the SJF scheduler should be preemptive or nonpreemptive

Shortest-Job First (SJF)

Suppose P_i is running and is expected to take an additional 10 ms, and that the ready queue is empty

- then suppose that several new processes arrive, and that the estimated burst time for each of them is $\ll$ 10 ms (say 2 ms)
- what should happen?
- one solution is to use a “shortest remaining time” regime, which requires preemption

Round-Robin Scheduling

Especially useful for time-shared systems

FCFS with preemption

Round-Robin Scheduling

Define a time quantum (aka time slice)

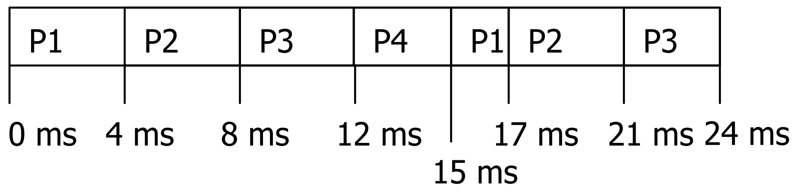
quantum is 10-100 ms: the CPY can do a lot of work in 10 ms!

- keep all processes in a circular queue
- dispatch the process at the head to the CPU
- if that process exceeds the time quantum on the CPU:
  - then preempt it: move the active process to the back of the queue
  - and assign the next process to the CPU
- or, if the active process relinquishes the CPU, then put it at the back of the queue and dispatch the process at the head

Round Robin

Example, with q = 4:

Process	**burst time duration (ms)**
P1	6
P2	8
P3	7
P4	3

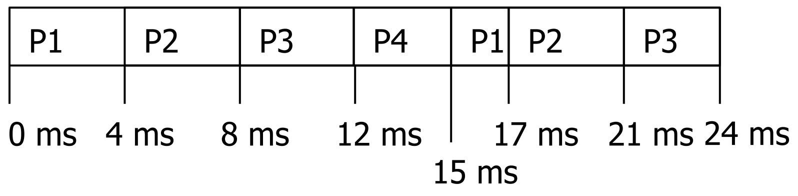


What's the average waiting time?

Round Robin

Example, with q = 4:

Process	burst time duration (ms)
P1	6
P2	8
P3	7
P4	3



What's the average waiting time?

- [(0 + (15-4)) + (4 + (17-8)) + (8 + (21-12)) + 12] / 4 = (11 + 13 + 17 + 12) / 4 = 13.25 ms

Round-Robin Scheduling

Considerations:

1. small time quantum $\Rightarrow$ frequent context switches

- context switch is essentially wasted CPU time
- so we want the time quantum to be relatively large compared to the context-switch time

2. large time quantum $\Rightarrow$ behavior approaches FCFS

- and as we've seen, waiting-time behavior is not predictable (or optimal) in FCFS

Priority Scheduling

Priority scheduling: give the CPU to the process in the ready queue that has the highest priority

SJF is a special case: in this case, the priority is the inverse of the estimated next burst time

- i.e., small value for next burst time $\Rightarrow$ high priority
- and large value for next burst time $\Rightarrow$ low priority

Priority Scheduling

But in general, the priority can be a value that is defined either internally or externally

- internal: based on memory requirements, I/O requirements, processing characteristics, etc.
- external: importance of the job, how much the user is paying, who the user is, etc.

Priority Scheduling

And we have the question again of whether the scheduling should be preemptive or nonpreemptive

- should a newly submitted high-priority process preempt a lower-priority process that is running?

Starvation (Indefinite Blocking)

Process that is in the ready queue, waiting for the CPU is said to be blocked

With preemptive scheduling, it’s possible that some processes can wait forever

- a constant stream of high-priority processes is submitted, continually preempting my low-priority job

How to avoid this?

Avoiding Starvation

One solution: hope that eventually the system will be lightly loaded (all the high-priority jobs will have run)

- and then our sad little low-priority process will get the CPU

Better solution: aging

- increase the priority of a job as it waits in the ready queue

Multilevel Queue Scheduling

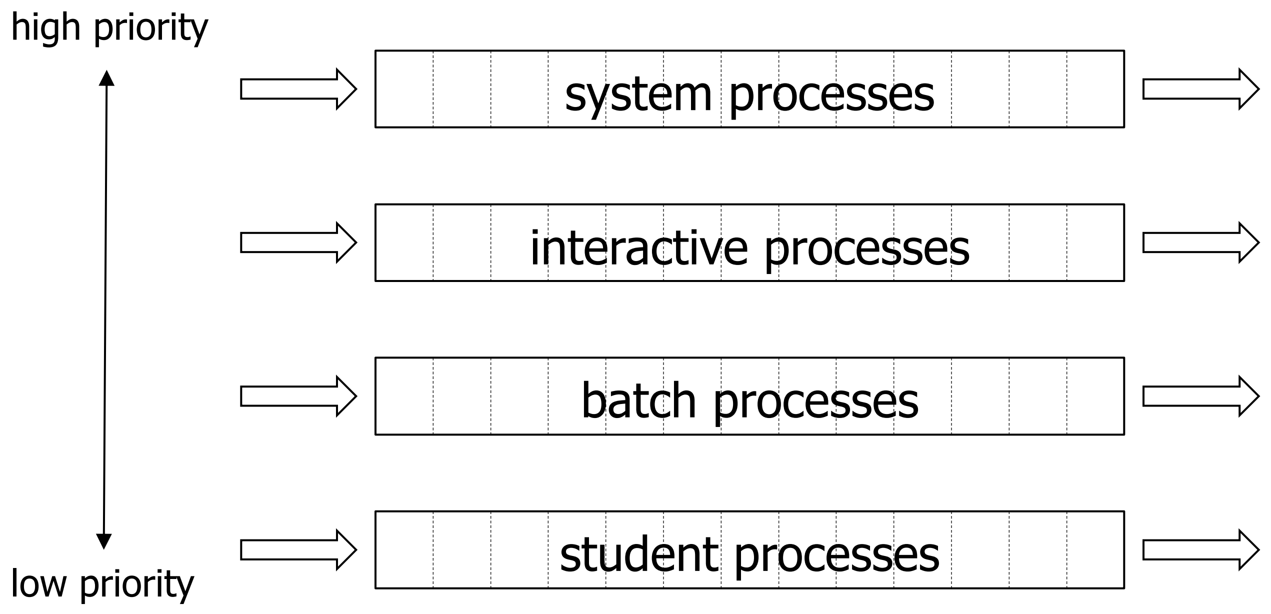
Partition the processes into classes, and maintain a separate queue—and scheduler—for each queue

Classes of processes might include

- system processes
- interactive processes
- batch processes
- student processes

Multilevel Queue Scheduling

Different queues have different schedulers



Different queues can then have different schedulers

- can do this strictly in priority order
- or timeslice the queues themselves

Multilevel Feedback Queue Scheduling

Allow processes to migrate between queues

- a process that uses too much CPU might be downgraded to a lower-priority queue
- I/O-bound processes can be put in higher-priority queues
- processes that age can be upgraded to a higher-priority queue

Multilevel Feedback Queue Scheduling

This scheme (MFQS) allows us to build the most general CPU scheduler

- but it’s also the most complex
- there are a lot of parameters to set and adjust

Thread Scheduling

In fact, on operating systems that support threads, it is kernel threads that the system schedules

user-level threads: live in a single process

- the CPU scheduler does not see these; they are managed by the threading library

kernel-level threads: these are scheduled by the system

OK not to know details of Sections 5.4.1, 5.4.2

Multiprocessor Scheduling

Consider homogeneous systems: each processor is like the others

- now we can do some load-sharing
- but the scheduling of processes to >1 processor can become complex

Multiprocessor Scheduling

Terminology:

much more common (Linux, macOS, Windows)

- symmetric multiprocessing (SMP): each processor is an equal candidate for a particular process
- asymmetric multiprocessing: one processor is the “master” and does all of the system scheduling; the other processors execute user code—used in systems that are more specialized; also on embedded systems

Multiprocessor Scheduling: Considerations

1. Processor affinity

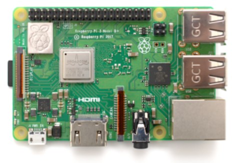
- as a process executes, the values from memory that it needs gradually fill the processor cache; this speeds up execution
- if we move a process from one processor to a different processor, then we must invalidate the cache for the first processor and let the process refill the cache for its new processor
- this is time consuming!

Multiprocessor Scheduling: Considerations

2. Load balancing

- it's in our interest to keep the workload across all of the processors approximately equal
- the system might employ push migration or pull migration to keep itself balanced
- what is the downside to process migration? The cache problem from the previous slide.

Multicore Processors



CPU architecture has moved strongly towards multicore processors

In a sense, we can treat each core as a CPU and assign threads to individual cores

- it’s actually a little more complicated than this, since cores share a cache
- in principle, the OS typically assigns two or more threads to each core, and the core interleaves the execution of the threads, using a very fast context-switching mechanism in hardware

Real-Time CPU Scheduling

Goal: minimize latency associated with switching from one process to another

Another goal might be: provide a bounded response time

- guarantee that it will take no more than a specified time to schedule and start a critical process

Real-Time CPU Scheduling

Soft real-time system:

- give high-priority (critical) tasks preference over lower-priority tasks

Hard real-time system:

- guarantee that a critical task will get attention before its deadline

Section 5.6 has a lot of further discussion, but it's enough to understand the material in Section 5.6.1

OS Examples

Linux: has two priority classes

- priority-based scheduling for real-time tasks
- Completely Fair Scheduler (CFS) for other tasks

CFS (Completely Fair Scheduler)

- essentially a priority scheduler
- tasks that use the CPU more have their priority reduced dynamically
- thus, I/O-bound tasks have their priority increased
- like a round-robin schedule except that the time quantum is dynamically adjusted

Algorithm Evaluation

How can we evaluate different scheduling algorithms?

What is the first question to ask?

Algorithm Evaluation

How can we evaluate different scheduling algorithms?

What is the first question to ask?

- on what basis will we evaluate performance of the algorithm?
- response time? throughput?
- or perhaps we want to maximize CPU utilization

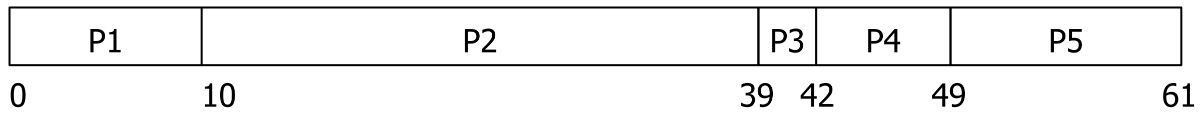
Deterministic Modeling

1. Construct a realistic workload
2. on paper, analyze the results of applying different algorithms to the workload
3. tally results

Deterministic Modeling

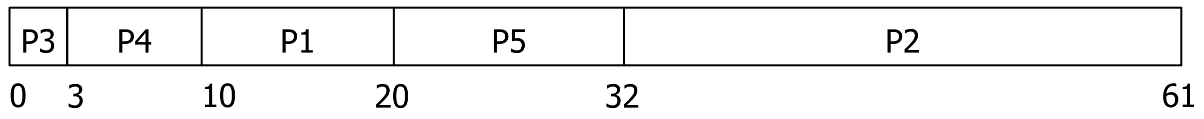
Process	Burst time (ms)
P1	10
P2	29
P3	3
P4	7
P5	12

FCFS



mean waiting time = ?

SJF

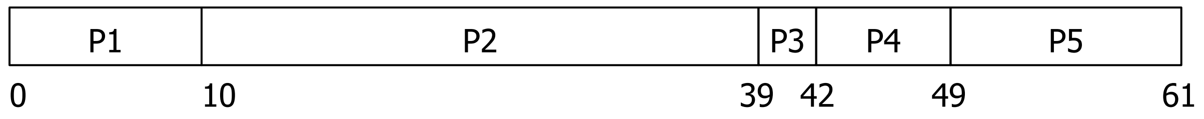


mean waiting time = ?

Deterministic Modeling

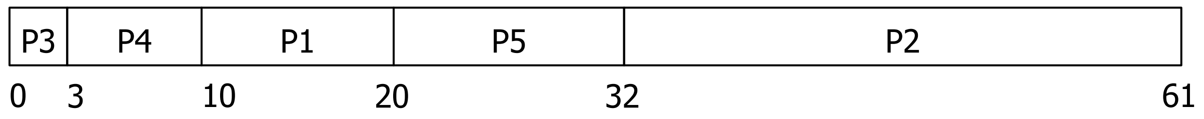
Process	Burst time (ms)
P1	10
P2	29
P3	3
P4	7
P5	12

FCFS



mean waiting time = (0+10+39+42+49) / 5 = 28 ms

SJF



mean waiting time = (0+3+10+20+32) / 5 = 13 ms

Deterministic Modeling

- gives us exact results
- but requires exact inputs
- well suited for demonstrating examples
- can help us gain insight and see trends

Queueing Models

Construct a mathematical model of a queue or of a series of queues

- queueing theory: prediction of queue behavior using statistical analysis
- specify a distribution for arrival times (i.e., when jobs are submitted) and for CPU burst times
- basic formula is Little's Formula: $n = \lambda W$, where n is average queue length, W is average waiting time, and λ is average arrival rate; this equation holds when the queue is in a steady state

Simulations

Construct a software model of the system

- generate random arrival times, CPU burst times, total per-process CPU time, etc.
- or, use actual trace data from an operating system
- feed this information into the simulator and see how different scheduling algorithms perform by gathering statistics of interest

assignment #4!

Implementation

The best system of all

- would allow us to analyze performance dynamically
- and adjust the behavior of the system as we collect real data about the use of the system