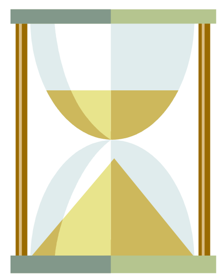
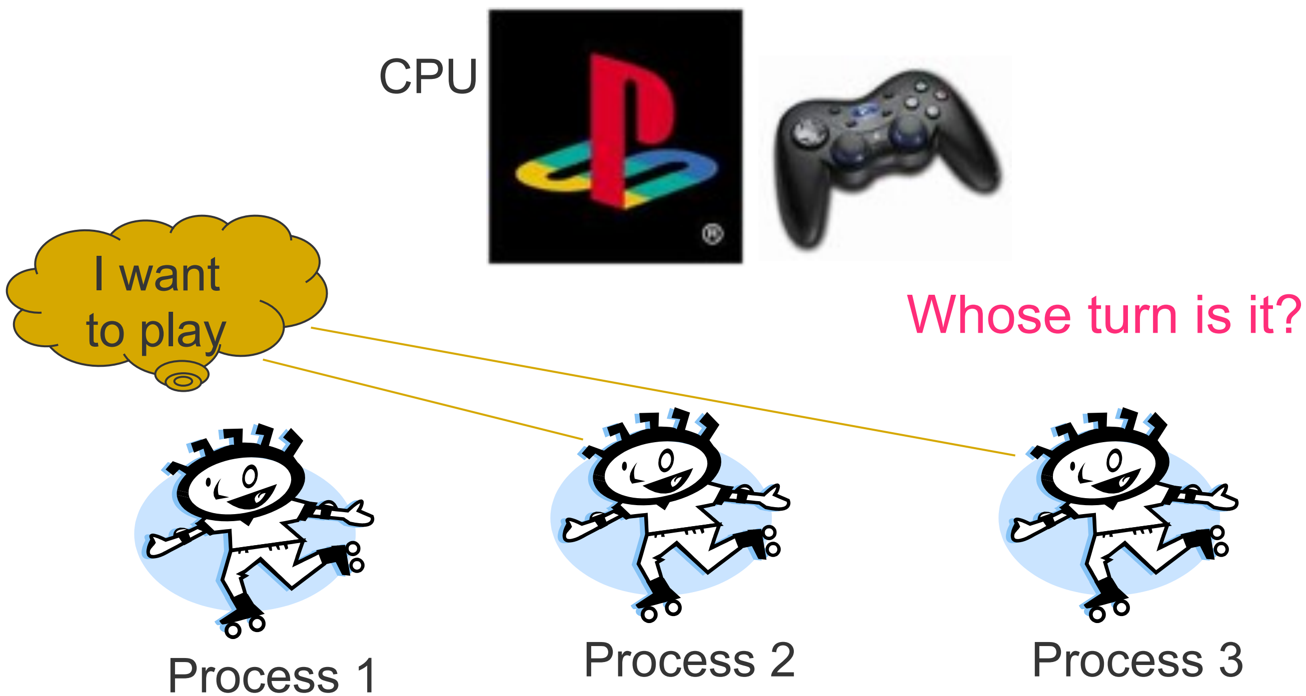
Process Scheduling



Process Scheduling

- Deciding which process/thread should occupy the resource (CPU, disk, etc)

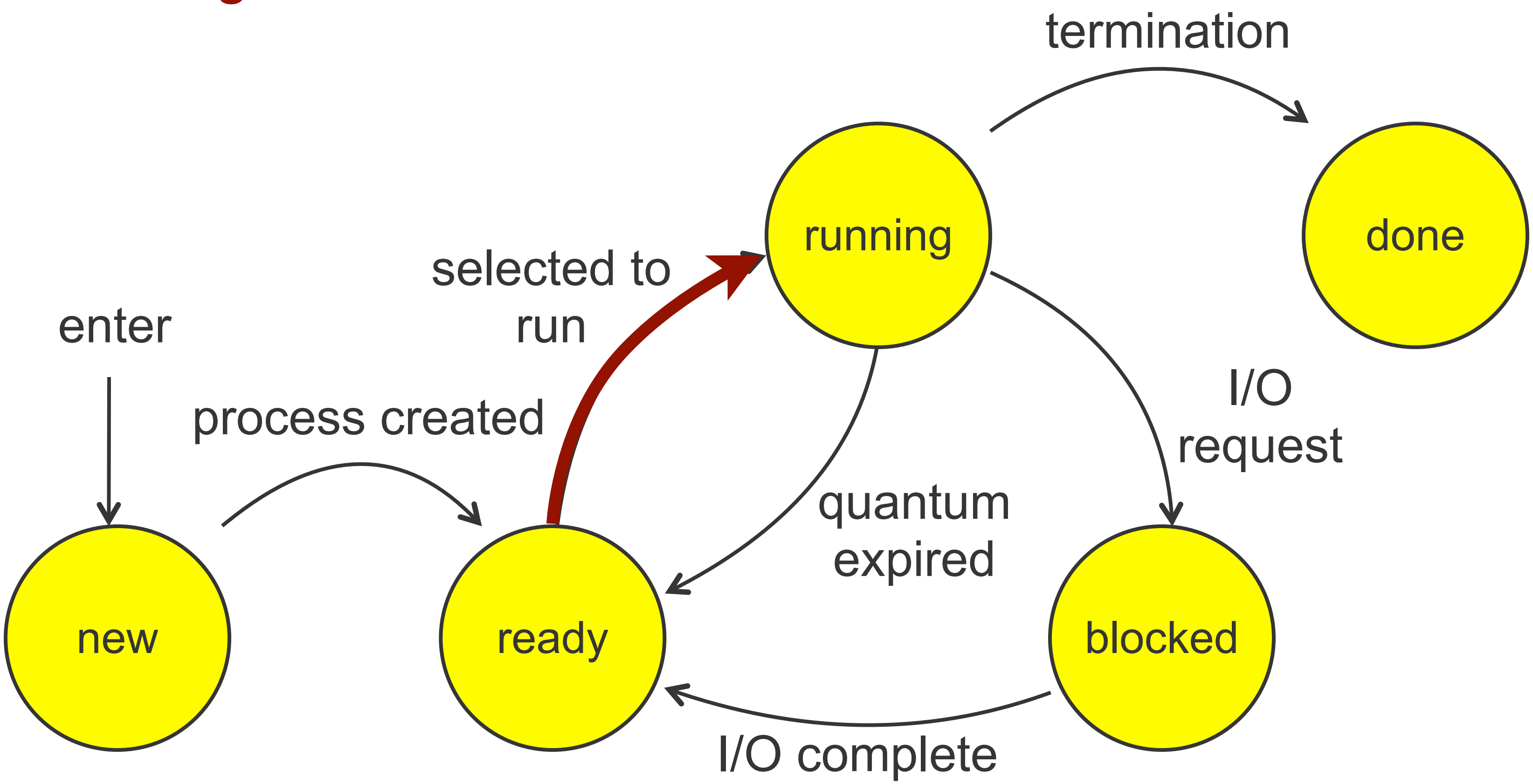


In this lecture

- Context: The scheduling problem
- Objectives
- Algorithms
- Conclusion

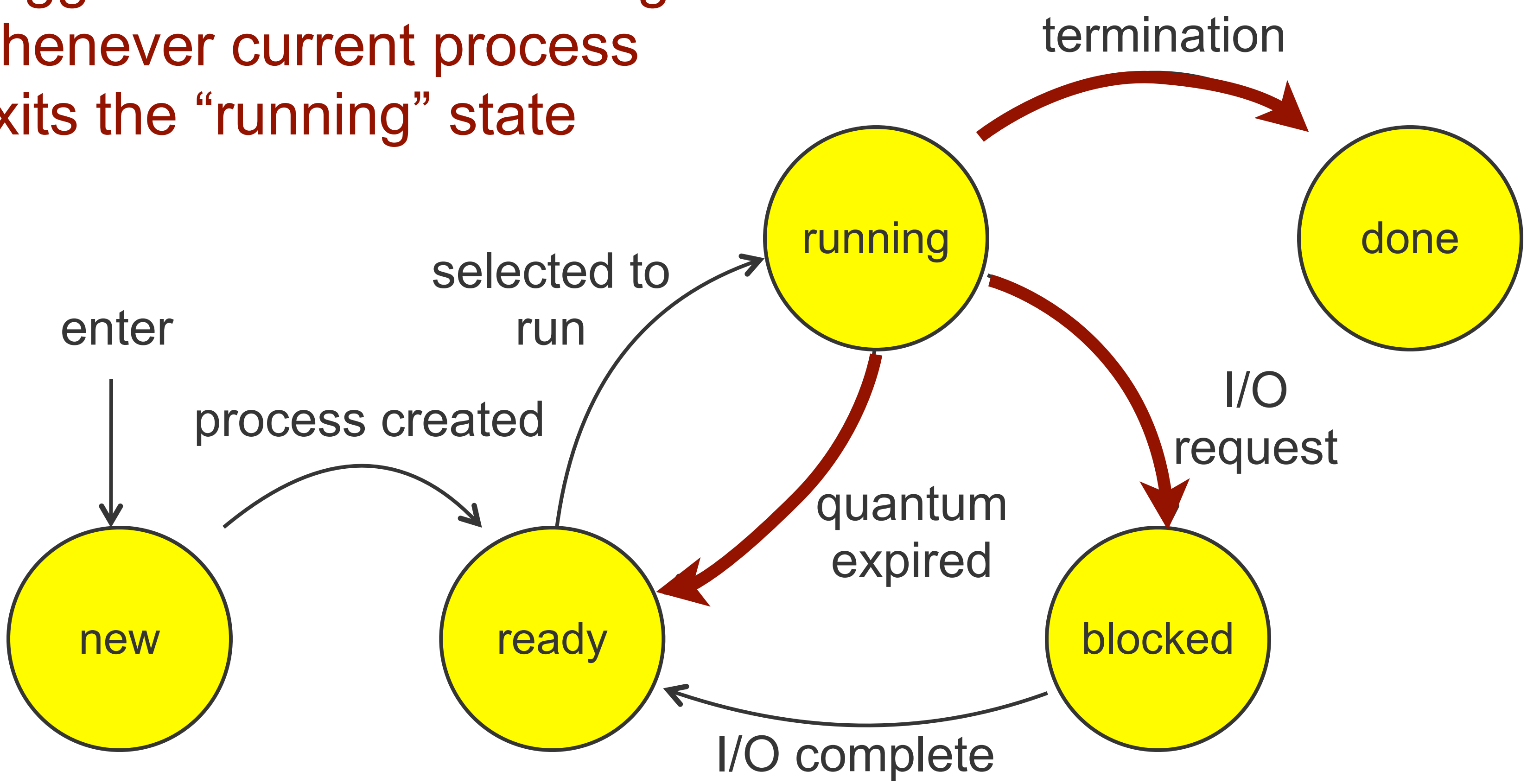
Where scheduling fits

Scheduling decision

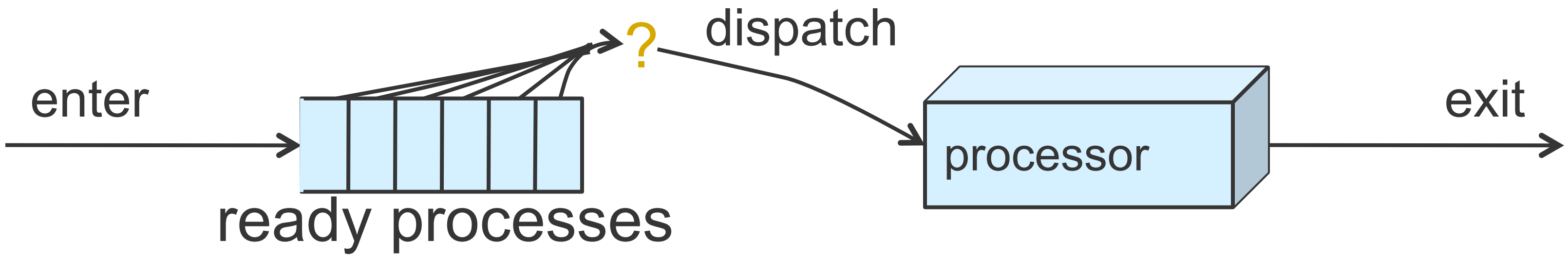


Where scheduling fits

Trigger to make scheduling decision: whenever current process exits the “running” state

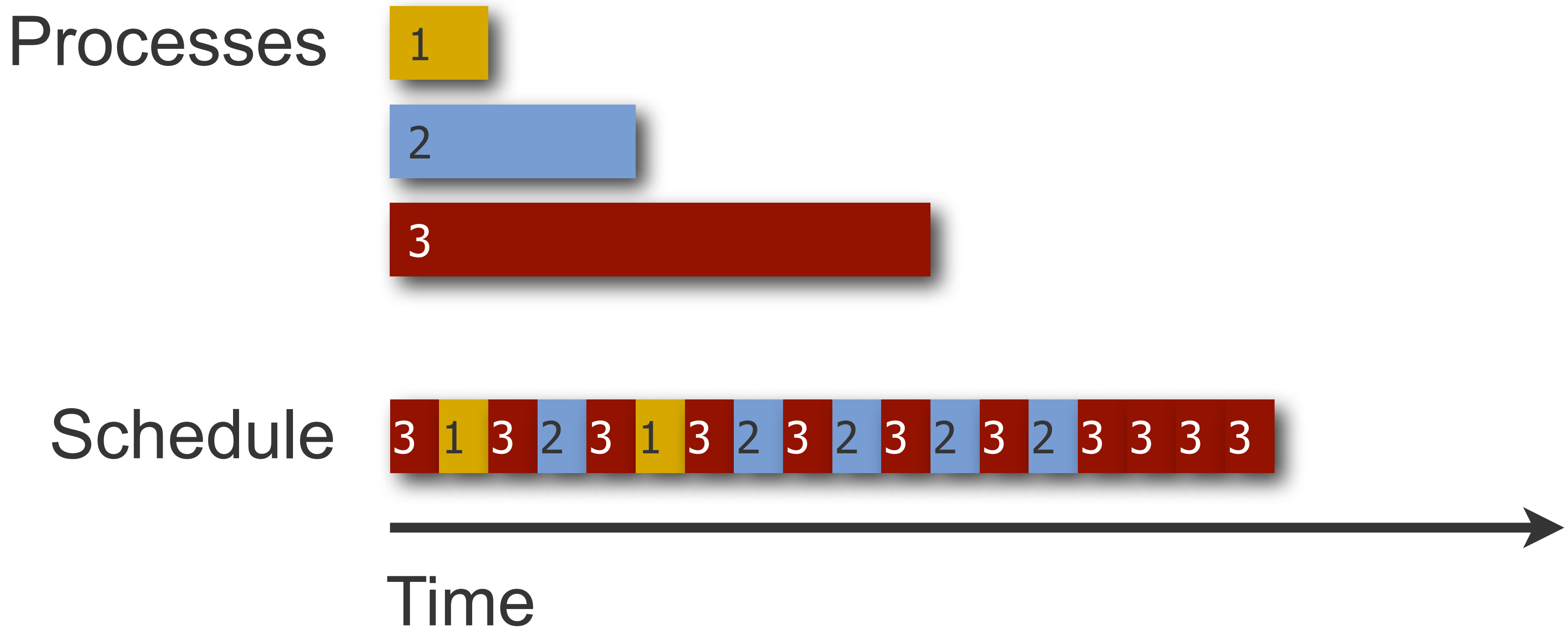


The basic scheduling decision



- Given ready processes, which one should I run next, and for how long?
  - ...for each resource (CPU, disk, ...)
- Same underlying concepts apply to scheduling processes or threads
  - or picking packets to send in routers!
  - or scheduling jobs in physical factories!

Example



Is this a good schedule?

Scheduling is not clear-cut

- Could I have done better? Depends!
  - Was some job very high priority?
  - Did I know when processes were arriving?
  - What's the context switch time?
  - What's my objective -- fairness, finish jobs quickly, meet deadlines for certain jobs, ...?
  - ...
- General-purpose OSes try to perform pretty well for the common case
  - Is this good enough to fly an airplane?
  - Special purpose (e.g., "real-time") scheduling exists

In this lecture

- Context: The scheduling problem
- Objectives
- Algorithms
- Conclusion

High-level objectives

- Fairness: equitable shares of CPU
- Priority: most important first
- Efficiency: make best use of equipment
- Encouraging good behavior: can’t take advantage of the system
- Support for heavy loads: degrade gracefully
- Adapting to different environments: interactive, real-time, multi-media

Quantitative objectives

- Fairness: processes get close to equal shares of the CPU
- Efficiency: keep resources as busy as possible
- Throughput: # of processes that complete per unit time
- Waiting Time: time a process spends waiting in kernel's ready queue

Quantitative objectives (cont'd)

- Turnaround Time: time from process start to its completion
- Response Time: amount of time from when a request was first submitted until first response is produced.
- Predictability and variance of any of the above objectives

Workloads

- I/O-Bound
  - Does too much I/O to keep CPU busy
  - E.g., interactive shell
- CPU-Bound
  - Does too much computation to keep I/O busy
  - E.g., a process sorting a million-entry array in RAM
- We should take advantage of these differences!
  - Scheduling should load balance between I/O-bound and CPU-bound processes
  - Ideal would be to run all equipment (CPU, devices) at 100% utilization

In this lecture

- Context: The scheduling problem
- Objectives
- Algorithms
- Conclusion

Scheduling Algorithms

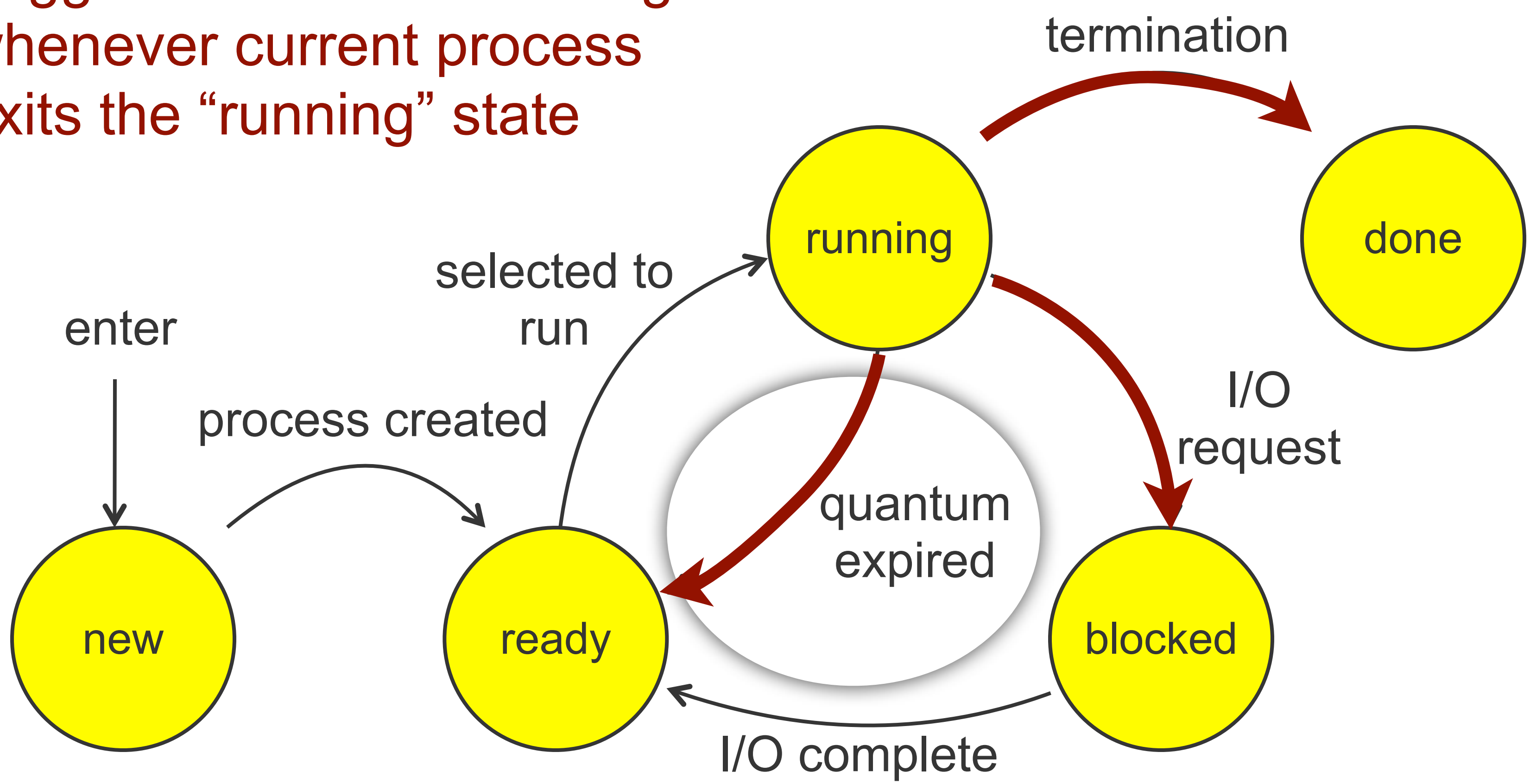
- Non-preemptive: batch systems
  - First come first serve (FCFS)
  - Shortest job first (SJF) *(also preemptive version)*
- Preemptive: interactive systems
  - Round robin
  - Priority
- These are some of the important ones to know, not a comprehensive list!

Preemption

- **Non-preemptive scheduling:**
  - The running process keeps the CPU until it voluntarily gives up the CPU
    - process exits
    - switches to blocked state
    - 1 and 4 only (no 3)
- **Preemptive scheduling**:
  - Running process is forced to give up CPU
  - Via interrupts or signals (we'll see these later)
    - What are interrupts?

Which transitions are preemptive?

Trigger to make scheduling decision: whenever current process exits the "running" state



First Come First Serve (FCFS)

- Process that requests the CPU first is allocated the CPU first.
- Also called FIFO
- Non-preemptive; used in batch systems
- Implementation
  - FIFO queues
  - A new process enters the tail of the queue
  - The scheduler selects next process to run from the head of the queue.

FCFS Example

Process	Duration	Order	Arrival Time
P1	24	1	0
P2	3	2	3
P3	4	3	7

The final schedule:

P1 waiting time: 0
P2 waiting time: 24-3=21
P3 waiting time: 27-7=20

The average waiting time:
(0+21+20)/3 = 13.67

What if arrival times of P1 and P2 are swapped?

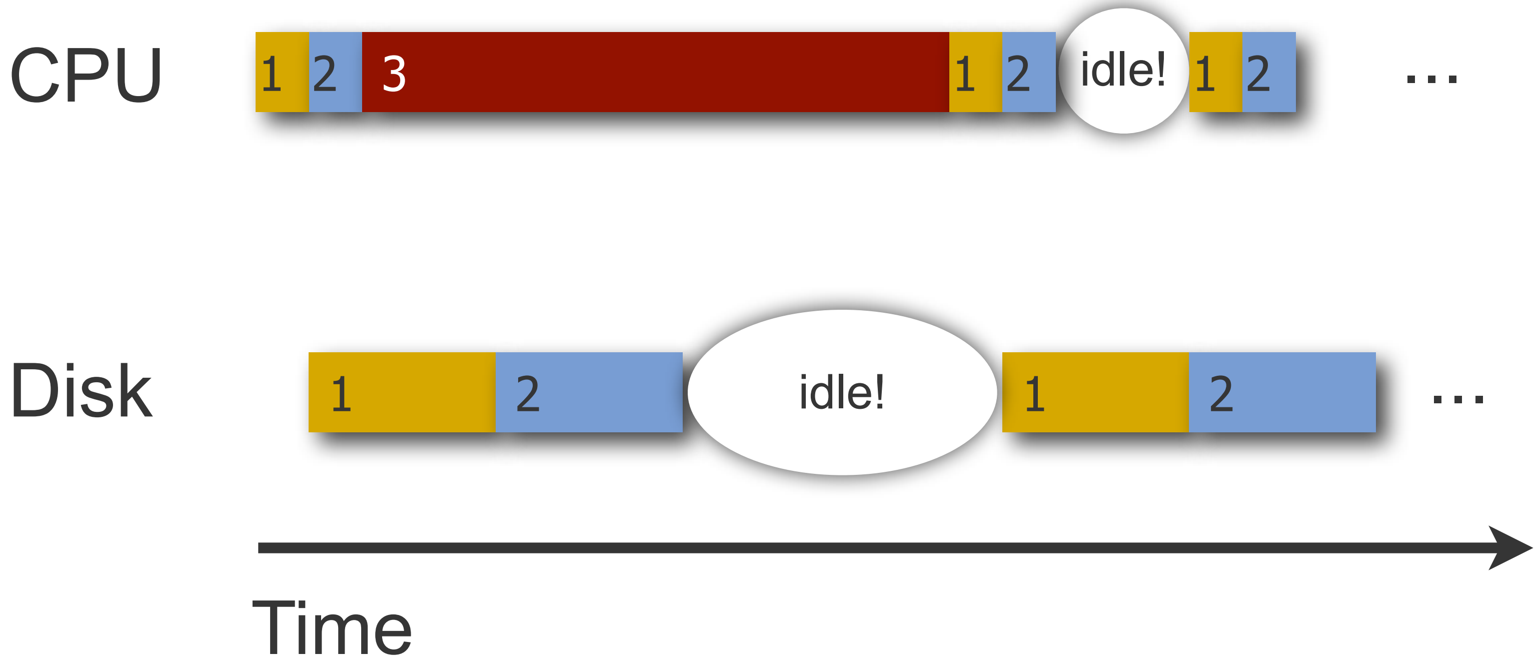
AWT (average waiting time) = (0+0+20)/3=6.67

Problems with FCFS

- Non-preemptive
- Not optimal AWT
- Cannot utilize resources in parallel:
  - Assume 1 process CPU bounded and many I/O bounded processes
  - Result: convoy effect, low CPU and I/O Device utilization

Convoy effect

Jobs 1,2 alternate: a bit of CPU, lots of disk.
Job 3 just wants a whopping chunk of CPU.

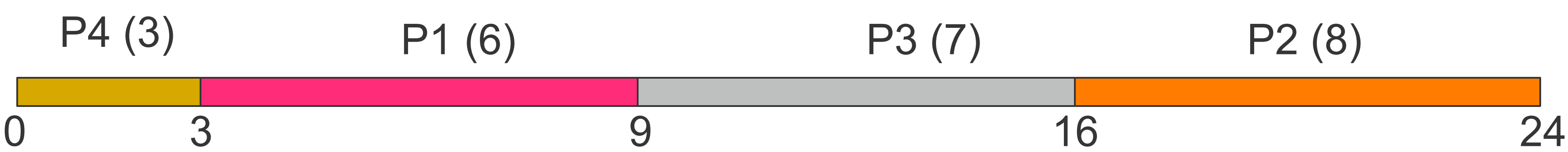


Shortest Job First (SJF)

- Job with shortest computation time goes first
- Scheduling often used in batch systems
- Two types:
  - Non-preemptive
  - Preemptive
- Optimal average waiting time if all jobs are available simultaneously
  - Why?

Non-preemptive SJF: Example

Process	Duration	Order	Arrival Time
P1	6	1	0
P2	8	2	0
P3	7	3	0
P4	3	4	0



P4 waiting time: 0
P1 waiting time: 3
P3 waiting time: 9
P2 waiting time: 16

Total waiting time = 0+3+9+16 = 28
Average waiting time = 28/4 = 7

Comparing to FCFS

Process	Duration	Order	Arrival Time
P1	6	1	0
P2	8	2	0
P3	7	3	0
P4	3	4	0

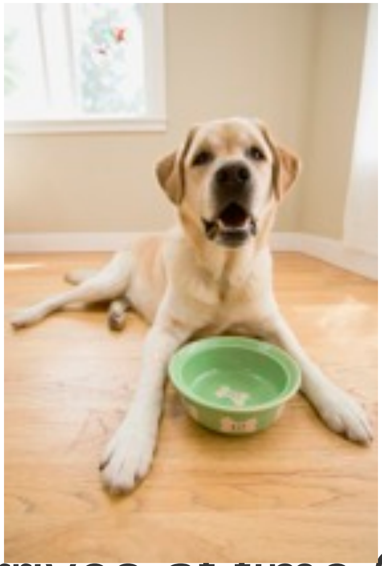
P1 waiting time: 0
P2 waiting time: 6
P3 waiting time: 14
P4 waiting time: 21

The total time is the same.
The total waiting time is not the same.
AWT = (0+6+14+21)/4 = 10.25
(compare to SJF's AWT = 7)

Preemptive SJF

- Shortest job runs first.
- A job that arrives and is shorter than the running job will preempt it

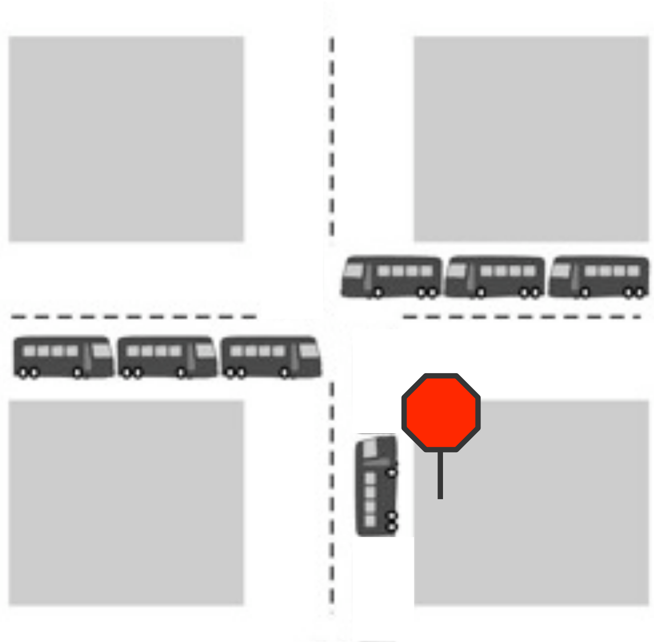
A Problem with Preemptive SJF



- Starvation
  - A job may keep getting preempted by shorter ones
  - Example
    - Process A with elapse time of 1 hour arrives at time 0
    - But every 1 minute, a short process with elapse time of 2 minutes arrives
    - Result of SJF: A never gets to run
- What's the difference between starvation and deadlock?

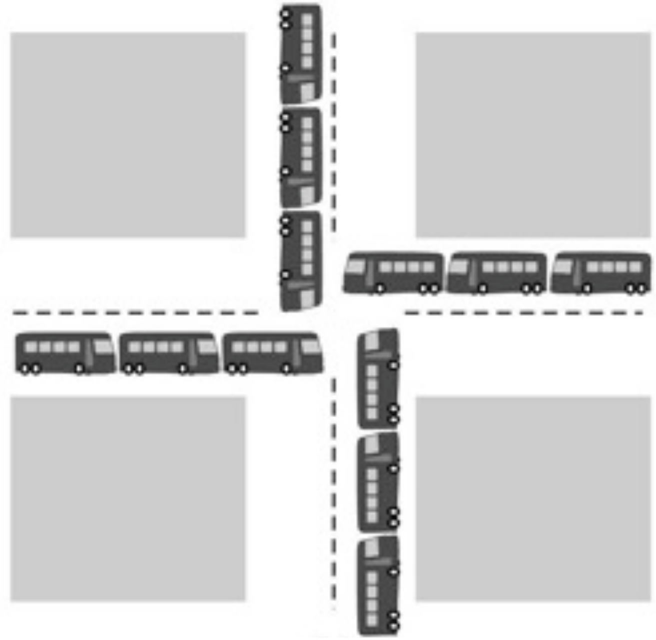
Starvation vs. deadlock

Starvation



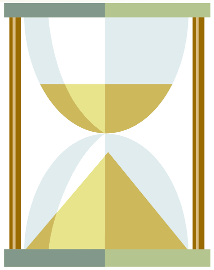
Unlucky job unlikely to make progress

Deadlock



No hope of progress for anyone involved

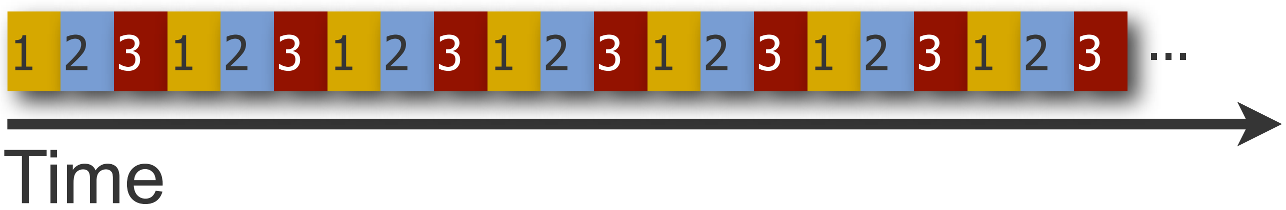
Interactive Scheduling



- Usually preemptive
  - Time is sliced into quanta, i.e., time intervals
  - Scheduling decision is also made at the beginning of each quantum
- Performance Metrics
  - Average response time
  - Fairness (or proportional resource allocation)
- Representative algorithms:
  - Round-robin
  - Priority scheduling

Round-robin

- One of the oldest, simplest, most commonly used scheduling algorithms
- Select process/thread from ready queue in a round-robin fashion (i.e., take turns)

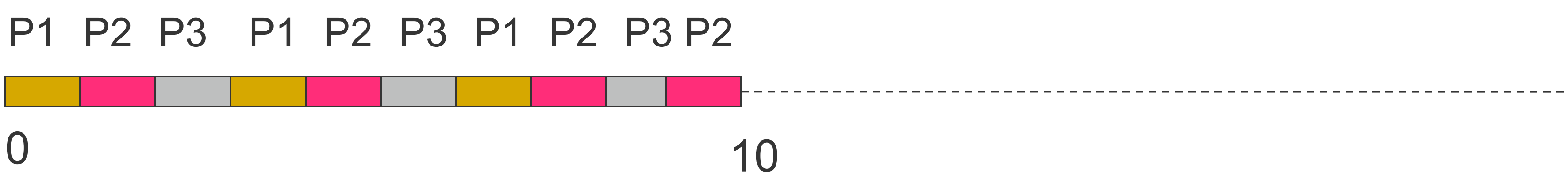


- Problems
  - Might want some jobs to have greater share
  - Context switch overhead

Round-robin: Example

Process	Duration	Order	Arrival Time
P1	3	1	0
P2	4	2	0
P3	3	3	0

Suppose time quantum is 1 unit and P1, P2 & P3 never block



P1 waiting time: 4
P2 waiting time: 6
P3 waiting time: 6

The average waiting time (AWT):
(4+6+6)/3 = 5.33

Choosing the time quantum

- Time quantum too large
  - FIFO behavior
  - Poor response time
- Time quantum too small
  - Too many context switches (overheads)
  - Inefficient CPU utilization
- How should we choose the time quantum?

Choosing the time quantum

Objective 1: Fast response time

Best case: quantum = 0,
response time = C

Objective 2: Efficiency

Best case: quantum = infinity,
Job completion time = J

General strategy: set quantum = small constant * C

E.g., quantum = 10C
So, response time ≤ 10C
Job completion time ≤ 1.1J

"Nearly" the best of both worlds

Choosing the time quantum

- Depends on priorities, architecture, etc.
- Typical quantum 10-100 ms
  - Large enough that overhead is small percentage
  - Small enough to give illusion of concurrency

Priority Scheduling

- Each job is assigned a priority.
- Select highest priority runnable job.
  - FCFS or Round Robin to break ties
- Rationale: higher priority jobs are more mission-critical
  - Example: DVD movie player vs. send email
- Problems:
  - May not give the best AWT
  - Starvation of lower priority processes

Priority Scheduling: Example

(Lower priority number is more preferable)

Process	Duration	Priority	Arrival Time
P1	6	4	0
P2	8	1	0
P3	7	3	0
P4	3	2	0

P2 waiting time: 0
P4 waiting time: 8
P3 waiting time: 11
P1 waiting time: 18

The average waiting time (AWT):
(0+8+11+18)/4 = 9.25
(worse than SJF)

Setting priorities

- In Unix, every process has a default priority
- User can also change a process priority
  - Command-line: nice, renice

```
NICE(1)                       User Commands                       NICE(1)

NAME
       nice - run a program with modified scheduling priority

SYNOPSIS
       nice [OPTION] [COMMAND [ARG]...]

DESCRIPTION
       Run  COMMAND  with  an adjusted scheduling priority.  With no COMMAND,
       print the current scheduling  priority.   ADJUST  is  10  by  default.
       Range goes from -20 (highest priority) to 19 (lowest).

       -n, --adjustment=ADJUST
              increment priority by ADJUST first

       --help display this help and exit
```

Setting priorities in C

GETPRIORITY(2) BSD System Calls Manual GETPRIORITY(2)

NAME

getpriority, **setpriority** -- get/set program scheduling priority

SYNOPSIS

```
#include <sys/resource.h>

int
getpriority(int which, id_t who);

int
setpriority(int which, id_t who, int prio);
```

DESCRIPTION

The scheduling priority of the process, process group, or user as indicated by which and who is obtained with the **getpriority**() call and set with the **setpriority**() call. Additionally, the current thread can be set to background state. Which is one of PRIO_PROCESS, PRIO_PGRP, ...

In this lecture

- Context: The scheduling problem
- Objectives
- Algorithms
- Conclusion

Issues to remember

- Why doesn't scheduling have one easy solution?
- What are the pros and cons of each scheduling policy?
- How does this matter when you're writing multiprocess/multithreaded code?
  - Can't make assumptions about when your process will be running relative to others!