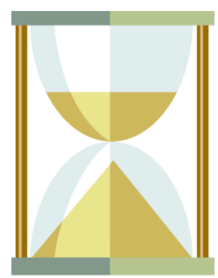
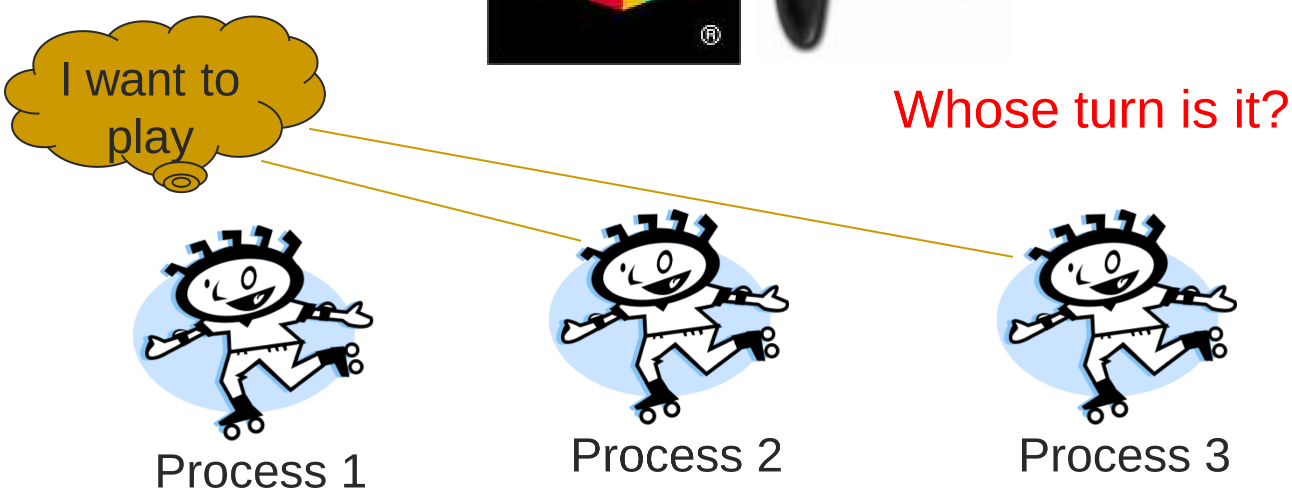
Process Scheduling



Process Scheduling

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

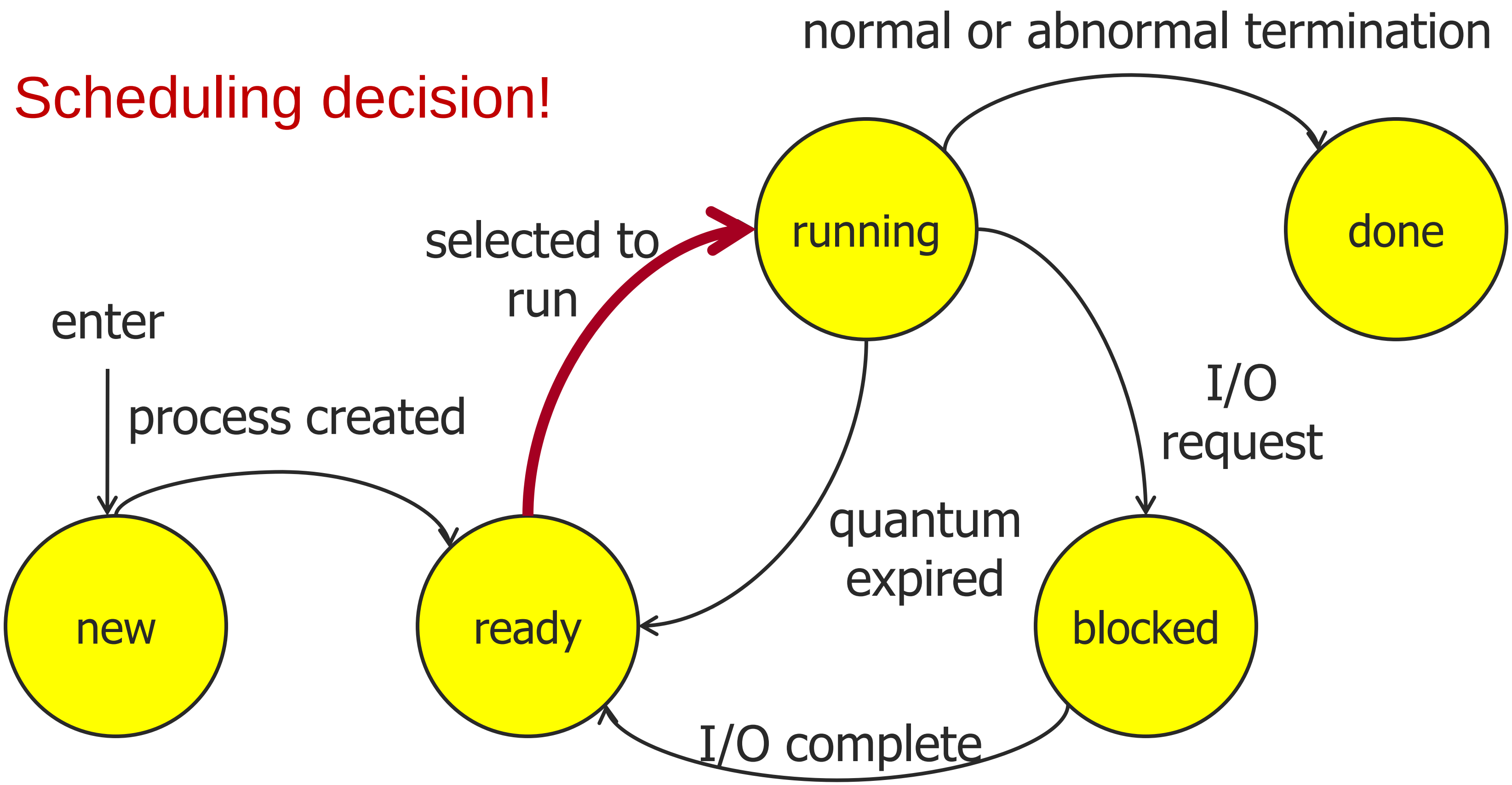
CPU



In this lecture

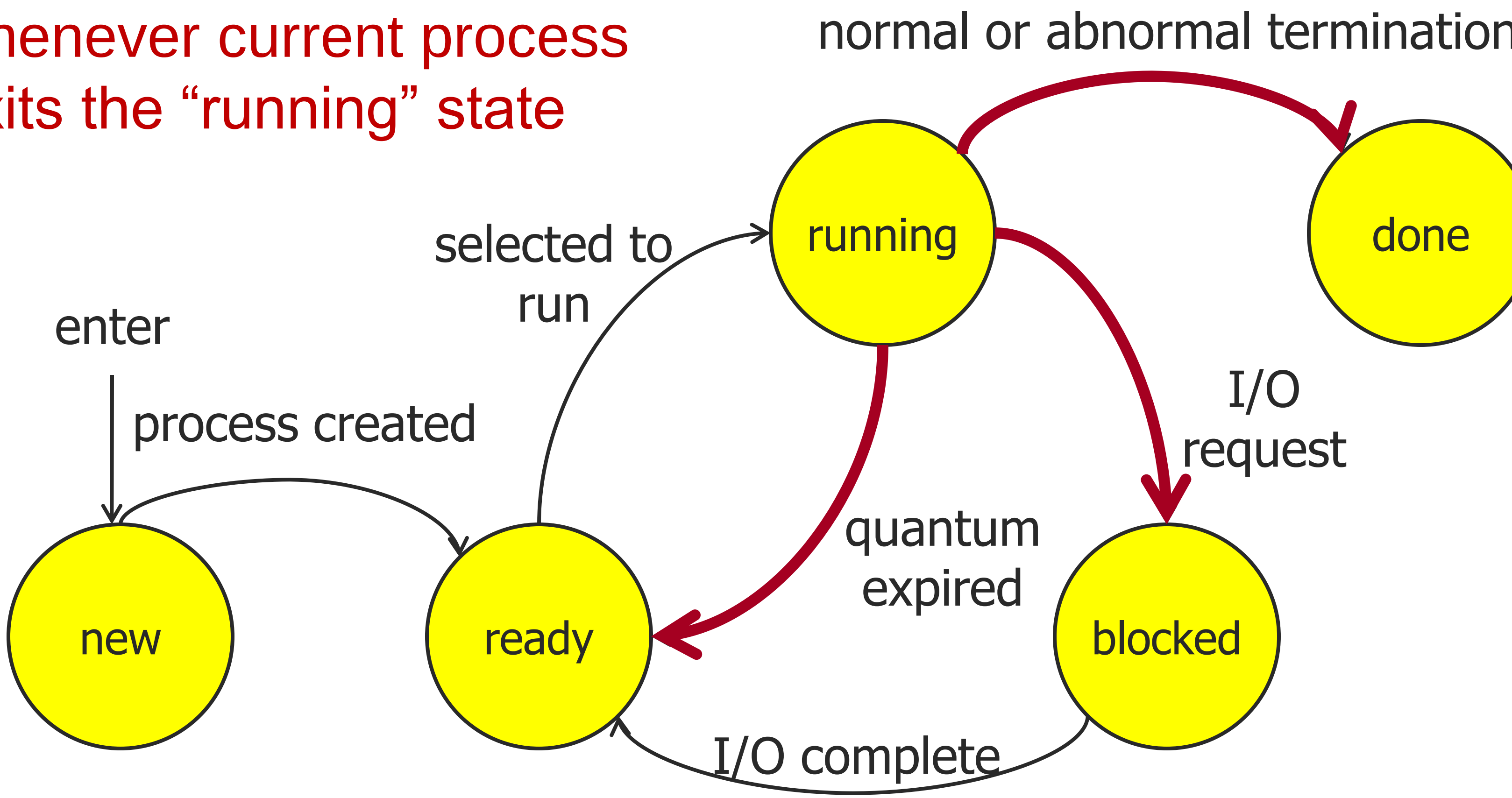
- Context: The scheduling problem
- Objectives
- Algorithms
- Conclusion

Where scheduling fits

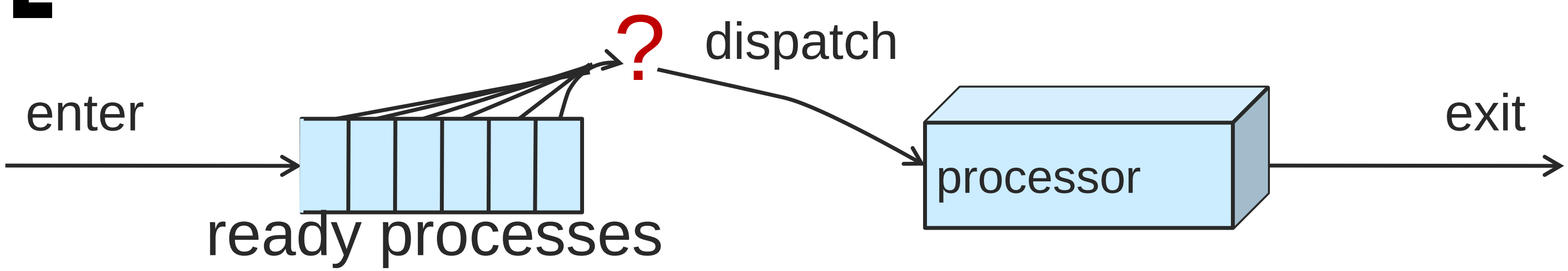


Where scheduling fits

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

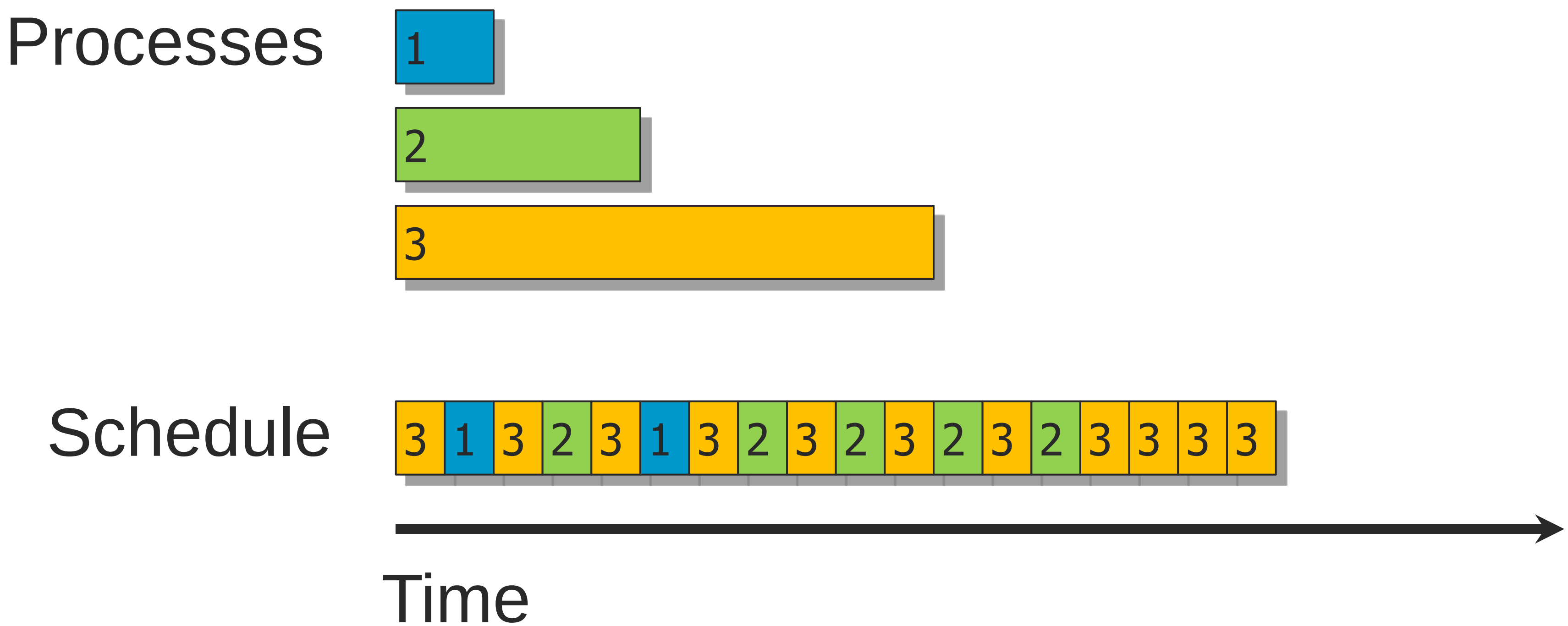


The basic scheduling decision



- Given a set of ready processes
  - Which one should I run next?
  - How long should it run?
  - ...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

High-level objectives

Objective	
Fairness	Equitable shares of resource
Priority	Allocate to most important first
Efficiency	Make best use of equipment
Encourage good behavior	Can’t take advantage of the system
Support heavy loads	Degrade gracefully
Adapting to different environments	Interactive, real-time, multi-media

Quantitative objectives

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

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

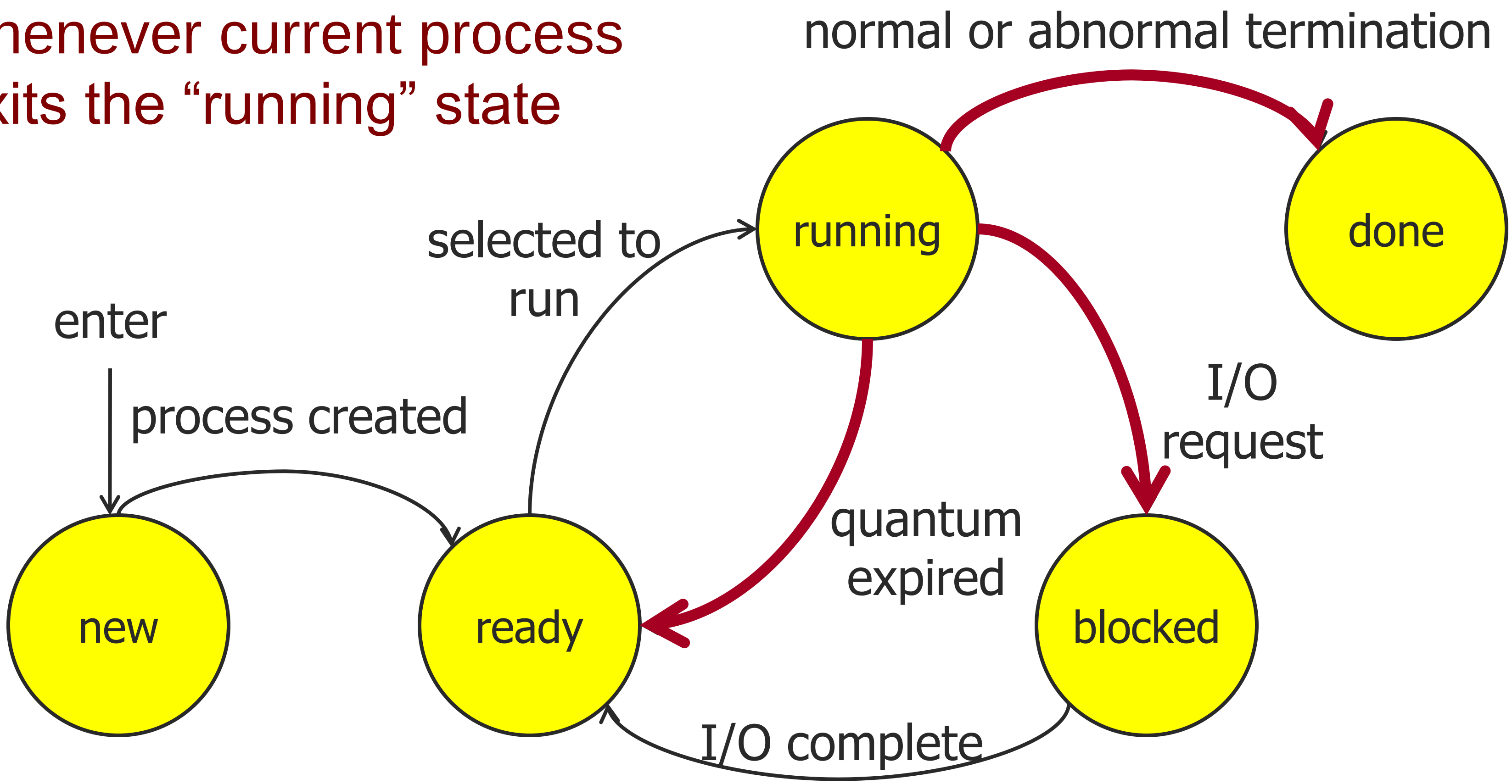
Scheduling Algorithms

- Non-preemptive: batch systems
  - Running process keeps CPU until it voluntarily gives it up
    - Process exits
    - Switches to blocked state
  - First come first serve (FCFS)
  - Shortest job first (SJF) (also preemptive version)
- Preemptive: interactive systems
  - Running process is forced to give up CPU
    - Via interrupts or signals (we'll see these later)
  - Round robin
  - Priority

These are some of the important ones to know, not a comprehensive list!

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

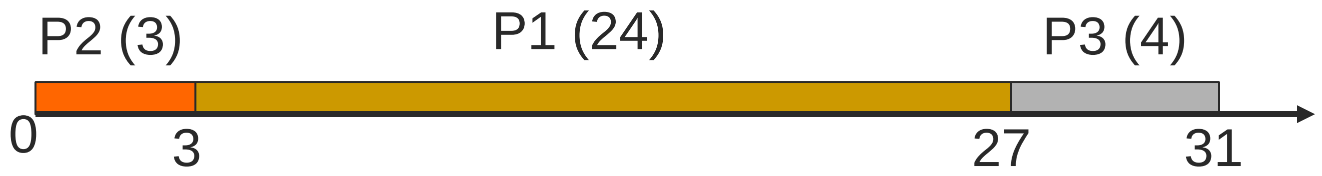
P1 waiting time:
P2 waiting time:
P3 waiting time:

The average waiting time:

FCFS Example

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

What if the arrival times of P1 and P2 are swapped?



P1 waiting time:
P2 waiting time:
P3 waiting time:

The average waiting time:

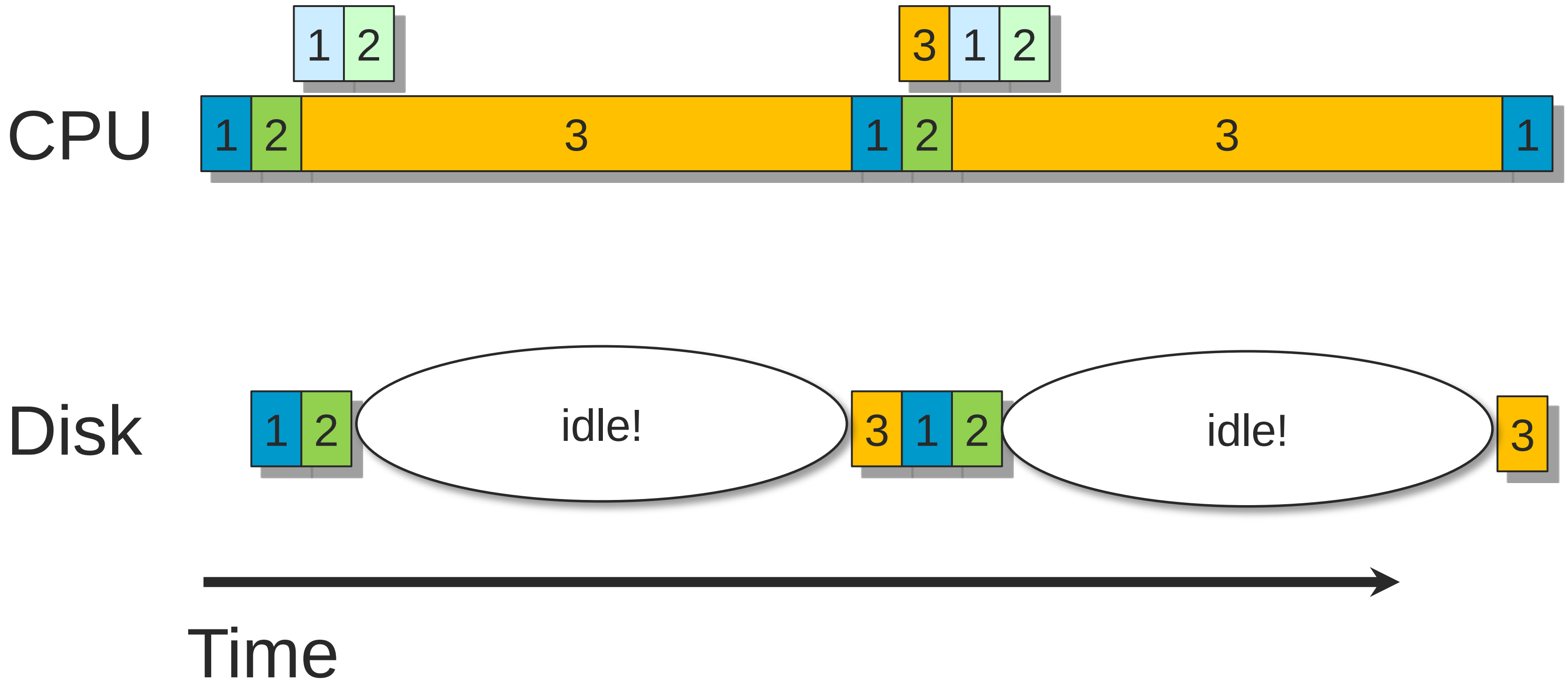
Problems with FCFS

- Non-preemptive
- Not optimal AWT
- Cannot utilize resources in parallel
  - Assume 1 process CPU bound and many I/O bound processes
- Result
  - Waiting time depends on arrival order
  - Potentially long wait for jobs that arrive later
  - Convoy effect, low CPU and I/O Device utilization

Convoy effect – Low I/O

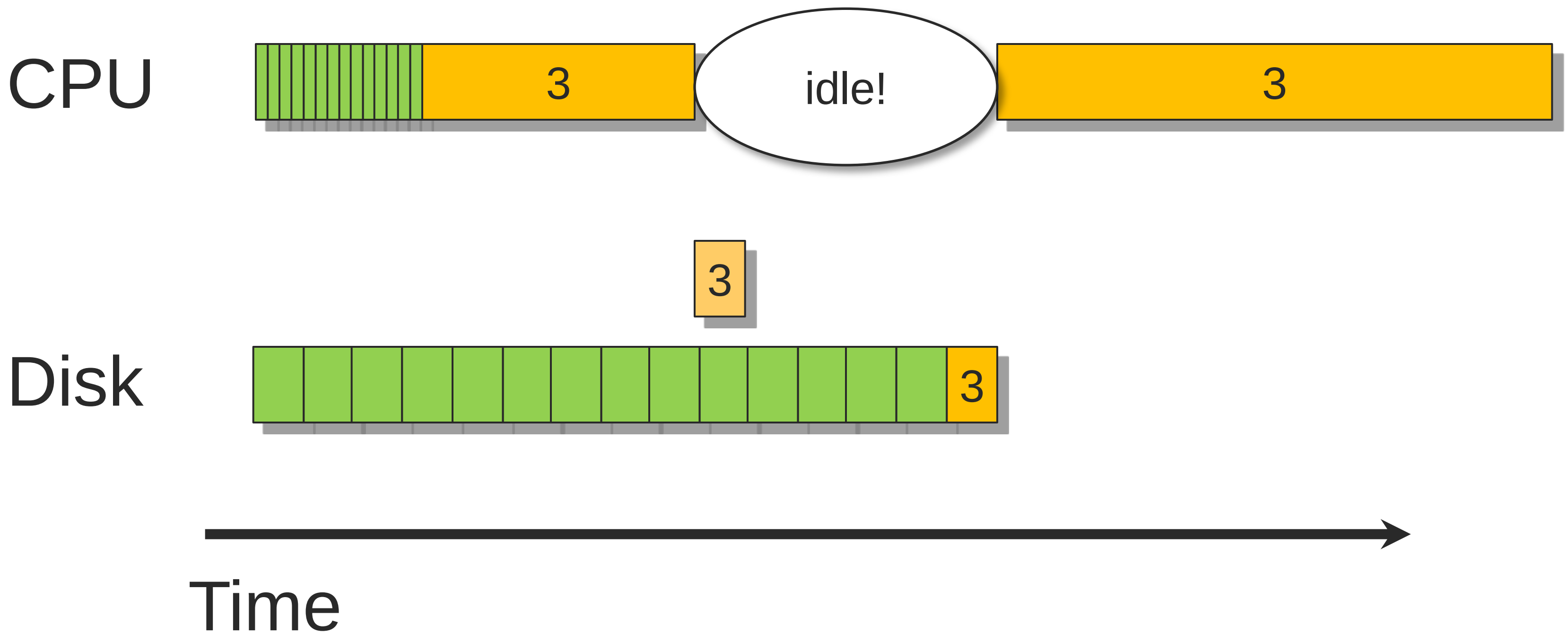
Jobs 1,2: a msec of CPU, lots of disk

Job 3: a sec of CPU, then a disk read



Convoy effect – Low CPU

Many jobs: a msec of CPU, lots of disk
Job 3: a sec of CPU, then a disk read

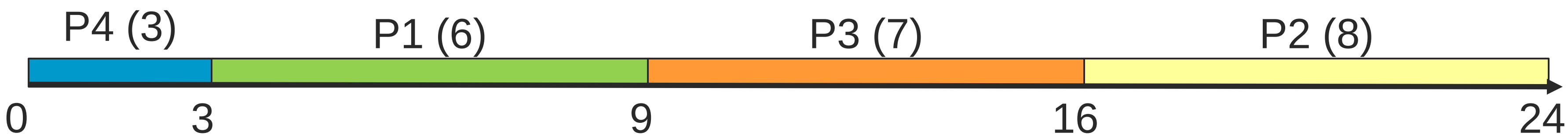


Shortest Job First (SJF)

- Job with shortest CPU time goes first
  - Often used in batch systems
- Two types
  - Non-preemptive
  - Preemptive

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

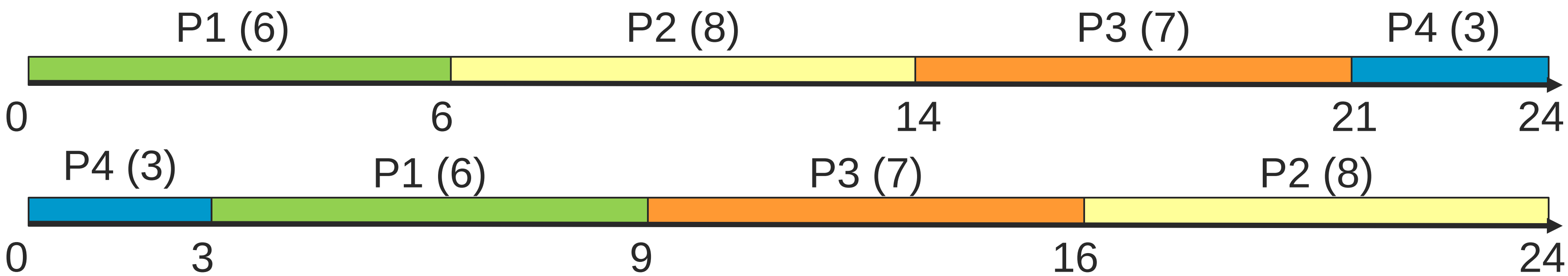


P1 waiting time:
P2 waiting time:
P3 waiting time:
P4 waiting time:

Total waiting time =
Average waiting time =

Compare 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:
P2 waiting time:
P3 waiting time:
P4 waiting time:

Total waiting time =
Average waiting time =

Non-preemptive SJF

- Advantages
  - Provably optimal for minimizing average wait time
    - Moving shorter job before longer job improves waiting time of short job more than it harms waiting time of long job
  - Helps keep I/O devices busy
- Disadvantages
  - Not practical: Cannot predict future CPU burst time
    - OS solution: Use past behavior to predict future behavior
  - Starvation: Long jobs may never be scheduled

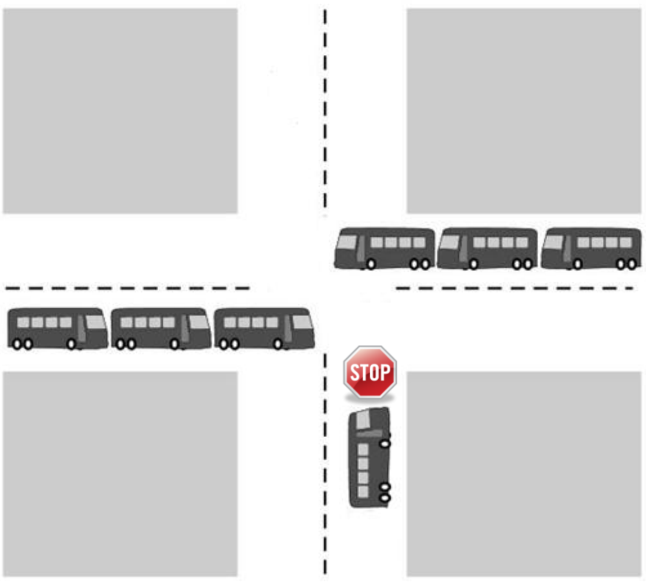
Preemptive SJF

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

Preemptive SJF

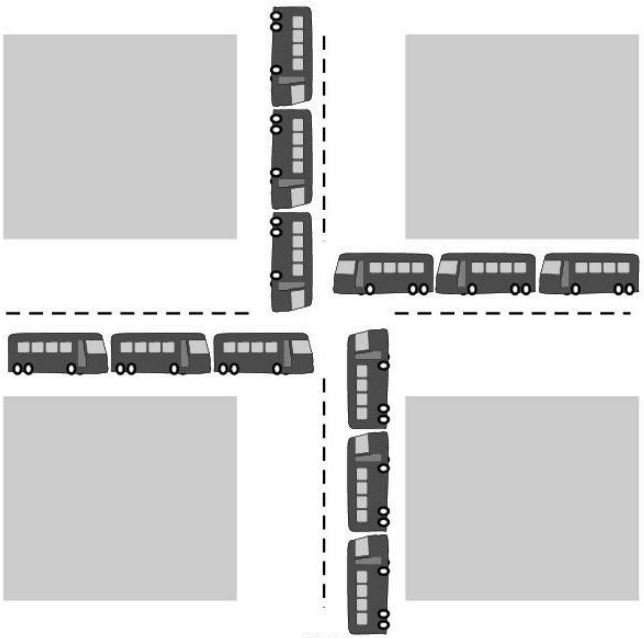
- Starvation again
  - A long job keeps getting preempted by shorter ones
  - Example
    - Process A with CPU time of 1 hour arrives at time 0
    - Every 1 minute, a short process with CPU time of 2 minutes arrives
    - What happens to A?
      - A never gets to run
- What's the difference between starvation and deadlock?

Starvation vs. Deadlock



(b)

Unlucky job unlikely to make progress



(b)

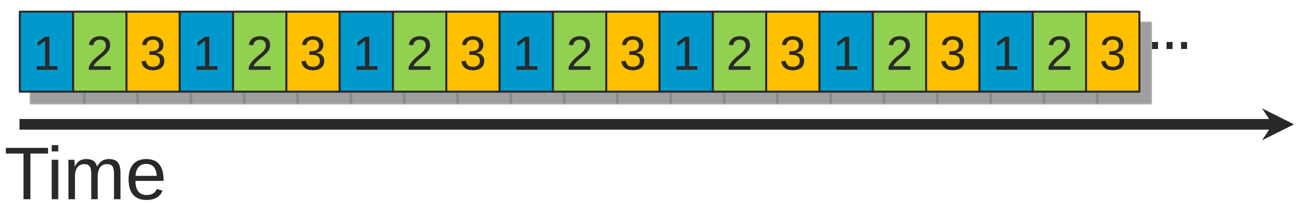
No hope of progress for anyone!

Interactive Scheduling

- Usually preemptive
  - Time is sliced into quanta, i.e., time intervals
  - Scheduling decisions are 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)

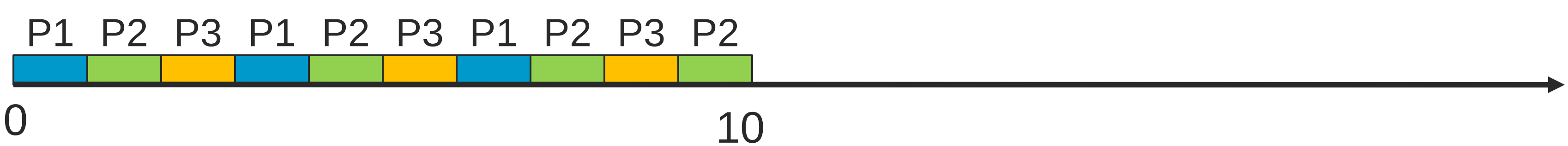


- Problems
  - Might want some jobs to have greater share
  - Context switch overhead

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:
P2 waiting time:
P3 waiting time:

The average waiting time (AWT):

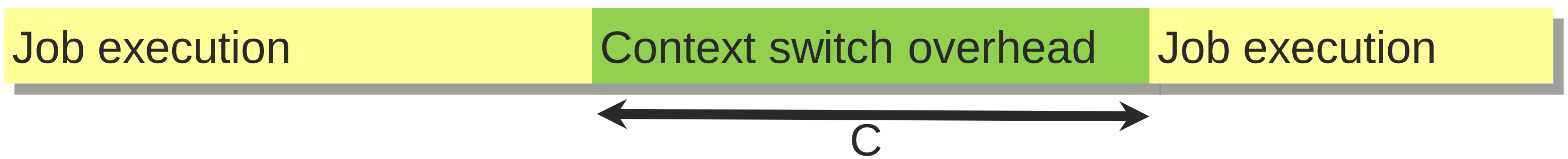
Round-robin

- Advantages
  - Jobs get fair share of CPU
  - Shortest jobs finish relatively quickly
- Disadvantages
  - Poor average waiting time with similar job lengths
    - Example: 10 jobs each requiring 10 time slices
    - RR: All complete after about 100 time slices
    - FCFS performs better!
  - Performance depends on length of time quantum

Choosing the time quantum

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

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

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

Priority Scheduling: Example

(Lower priority number is preferable)

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

P1 waiting time:
P2 waiting time:
P3 waiting time:
P4 waiting time:

The average waiting time (AWT):

Setting priorities: `nice`

`nice [OPTION] [COMMAND [ARG]...]`

- Run `COMMAND` with an adjusted niceness
- With no `COMMAND`, print the current niceness.
- Nicenesses range from -20 (most favorable scheduling) to 19 (least favorable).

- Options
  - `-n, --adjustment=N`
    - add integer N to the niceness (default 10)
  - `--help`
    - display this help and exit
  - `--version`
    - output version information and exit

Setting priorities in C

```
#include <sys/time.h>
#include <sys/resource.h>
int getpriority(int which, int who);
int setpriority(int which, int who, int prio);
```

- Access scheduling priority of process, process group, or user
- Returns:
  - `setpriority()` returns 0 if there is no error, or -1 if there is
  - `getpriority()` can return the value -1, so it is necessary to clear `errno` prior to the call, then check it afterwards to determine if a -1 is an error or a legitimate value
- Parameters:
  - `which`
    - `PRIO_PROCESS, PRIO_PGRP, or PRIO_USER`
  - `who`
    - A process identifier for PRIO_PROCESS, a process group identifier for PRIO_PGRP, or a user ID for PRIO_USER

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!