allocs A B C

| A | B | C |
| --- | --- | --- |
| 0 | 0 | 0 |

3 available chopsticks (to be avail takes into account of resources that will be returned eventually) → threads who can finish

| | A | B | C | | |
| --- | --- | --- | --- | --- | --- |
| $A^{+1}$ | 1 (1) | 0 (2) | 0 (2) | avail = 2, | to be avail = 3 |
| $C^{+1}$ | 1 (1) | 0 (2) | 1 (1) | avail = 1, | to be avail = 2, 3 |
| $B^{+1}$ | 1 (1) | 1 (1) | 1 (1) | avail = 0, | to be avail = 0 (no one can finish) |

∴ unsafe.

→ why do we need to know the max?

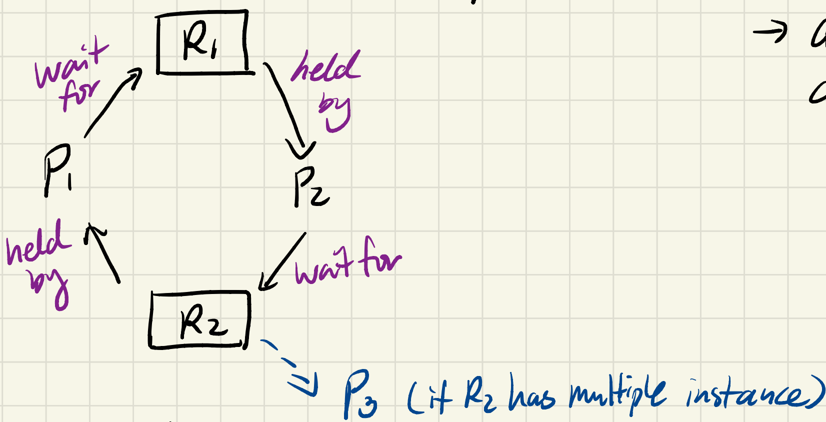
→ can we determine safe vs. unsafe w/out knowing the max?

10/27/23

→ avoidance

→ detection

→ Resource Allocation Graph




if there's a cycle:

single instance resource ⇒ deadlock

multi instance resource ⇒ potential deadlock

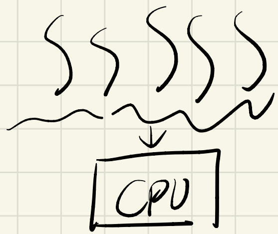
→ recover from deadlock

→ abort a process/task in the cycle. (why would this be ok?)

→ if app. supports abort & retry

→ typically impl. by app keeping updates locally so abort wouldn't leave things in a bad state

Scheduling : How to share the CPU?




a thread can do multiple tasks (read, encrypt, write)

wait time + task execution

scheduling + context switch

# Definitions

- Task/Job
  - User request: e.g., mouse click, web request, shell command, …
- Latency/response time (called turnaround time in OSTEP)
  - How long does a task take to complete?
- Throughput
  - How many tasks can be done per unit of time?
- Overhead
  - How much extra work is done by the scheduler?
- Fairness
  - How equal is the performance received by different users?
- Strategy-proof
  - Can a user manipulate the system to gain more than their fair share?
- Predictability
  - How consistent is a user's performance over time?

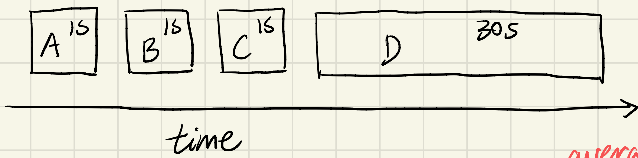
# Scheduling Policies

## ① First in First out (FIFO)

→ scheduling tasks in the order they arrive, each task runs to completion

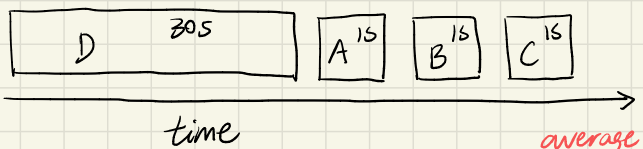
→ wait in line in grocery store

assume that jobs arrive at approximately the same time in the following order




$T_{latency} = T_{completion} - T_{arrival}$ → both execution & wait time are included.

average latency $= \frac{(1+2+3+33)}{4}$




average latency $= \frac{(30+31+32+33)}{4}$

] depends on the order of arrival

Pros: simple, minimum switching btwn threads

Cons: varying latency

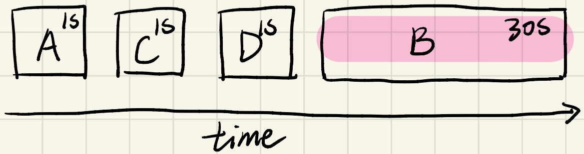
## ② Shortest Job First (SJF)

→ also called Shortest Remaining Time First (SRTF)

→ complete the short task first, if shorter task arrives, preempt the current task, switch to the shorter task.

How would we know if a task is long or short?

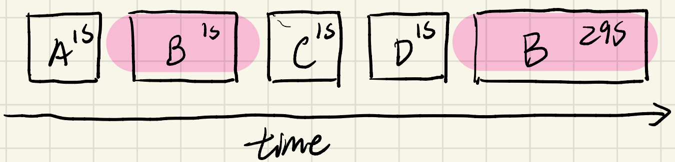
assume that jobs arrive at approximately the same time in alphabetical order




(though B arrived second, it's at the end of the queue)

→ if more small jobs keep arriving B can be starved (never get a chance to run)

assume that jobs arrive at different times in alphabetical order




→ B gets preempted when a smaller task arrives

Pros: optimal average latency

Cons: Starvation, can result in more context switches if we keep preempting longer tasks

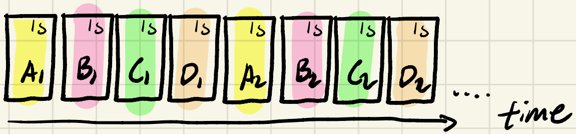
## ③ Round Robin (RR)

time slice / time quantum

→ FIFO but with fixed time for each task

→ no starvation!

assume that jobs arrive at approximately the same time in alphabetical order




(subscript = # of times scheduled)

impact on average latency

[if each task takes 2 seconds to finish w/ 1s time slice]

$$SJF = \frac{(2+4+6+8)}{4} \qquad RR = \frac{(5+6+7+8)}{4}$$

How to decide on the time quantum?

→ too large? similar to FIFO

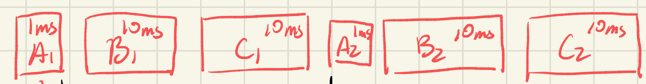
→ too small? lots of context switch overhead

→ typically 10–100ms

assume 10ms time quantum

A: I/O bound (runs for 1 ms, blocks for 3 ms, runs for 1 ms)

B, C: CPU bound (needs 20 ms total to finish the task)




blocks before time quantum expires

A waits a long time before getting scheduled again, "fixed time" impacts I/O bound & CPU bound jobs differently

☆ bad for I/O task response time

## ④ Multilevel Feedback Queue (MLFQ)

→ RR but multiple queues with increasing time quantum

→ improve latency for I/O bound (interactive) jobs

☆ one size doesn't fit all, so have multiple time quantum!