allows $A, B, C$

0 1 0 0

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

A+i

(1) (2) (2)

1 0 0

avail = 2, to be avail = 3

C+i

(1) (2) (1)

1 0 1

avail = 1, to be avail = 2, 3

B+i

(1) (1) (1)

1 1 1

avail = 0, to be avail = 0 (no one can finish)

\& unsafe.

\rightarrow Why do we need to know the max?

\rightarrow can we determine safe vs. unsafe without knowing the max?
10/27/23

- avoidance
- detection

- Resource Allocation Graph

- 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

- if there's a cycle:
  - Single instance resource => deadlock
  - Multi instance resource => potential deadlock
Scheduling: How to share the CPU?

- Task/Job:
  - User request: e.g., mouse click, web request, shell command, ...
- Latency/response time:
  - 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:

\[
\begin{array}{c}
A^{15} & B^{15} & C^{15} & D^{205} \\
\hline
D^{205} & A^{15} & B^{15} & C^{15}
\end{array}
\]

\[
T_{\text{Latency}} = T_{\text{Completion}} - T_{\text{Arrival}}
\]

Pros: simple, minimum switching between threads
Cons: varying latency
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:

```
A 15  C 15  D 15  B 305
```

Assume that jobs arrive at different times in alphabetical order:

```
A 15  B 15  C 15  D 15  B 295
```

Pros: optimal average latency

Cons: starvation, can result in more context switches if we keep preempting larger tasks
Round Robin (RR)

→ FIFO but with **fixed time** for each task
→ no starvation!

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

Assume 10ms time quantum

A: I/O bound (runs for 1ms, blocks for 3ms, runs for 1ms)
B, C: CPU bound (needs 20ms to finish the task)

Impact on average latency

\[
S_{JF} = \frac{2 + 4 + 6 + 8}{4} \quad RR = \frac{5 + 6 + 7 + 8}{4}
\]

How to decide on the time quantum?

→ too large? similar to FIFO
→ too small? lots of context switch overhead
→ typically 10-100ms
4. Multilevel Feedback Queue (MLFQ)
   - RR but multiple queues with increasing time quantum
   - Improve latency for I/O bound (interactive) jobs

<table>
<thead>
<tr>
<th>Queue</th>
<th>Time Quantum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>1 ms</td>
</tr>
<tr>
<td>Q2</td>
<td>5 ms</td>
</tr>
<tr>
<td>Q3</td>
<td>10 ms</td>
</tr>
</tbody>
</table>

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