Lecture 10: Deadlock

- Conditions for deadlock
- Deadlock prevention
- Deadlock detection
- Deadlock avoidance
“Dining philosophers”

Deadlock: no philosophers can make progress because all chopsticks taken.

```
philosopher
while true:
  - grab left chopstick
  - grab right chopstick
  - put sticks down
  - eat
  - think
```

```
context switch
```

```
0
```

```
```

```
0
```

```
```

```
0
```

```
```

```
0
```

```
```

```
0
```

```
```

```
0
```

```
```

```
0
```

```
```

```
0
```

```
```

```
0
```

```
```

```
0
```

```
```

```
0
```

```
```

```
0
```

```
```

```
0
```

```
```

```
0
```

```
```

```
0
```

```
```

```
0
```

```
```

```
0
```

```
```

```
0
```

```
```

```
0
```

```
```

```
0
```

```
```

```
0
```

```
```

```
0
```

```
```

```
0
```
4 necessary & sufficient conditions for deadlock

1. Mutual exclusion: Can't share resource (e.g., bunch of readers can share R1 but can't deadlock)
2. No preemption: threads can't voluntarily give up resources (e.g., CPU can be taken away)
3. Hold and wait: thread is able to hold one resource & wait for another
4. Circular wait: in graph of processes & resources, will edges representing who holds what resource, who is waiting, need a cycle.

Resource allocation graph

RAG

- threads
- resources

T → R : T is trying to acquire R
R → T : R is held by T.

During phi.1:
- Phil 1
- Chop stick 1
- Phi.2
- Chop stick 2

Phi.2 has Chop stick 1, trying to get CS 2.
1. Preventing mutual exclusion
   - Make resources shareable; virtualization.
   e.g., CPU = virtualizable

2. No preemption
   - And the ability for one thread to take resources from another.
     "Safely" taking away a lock leads to lost update.

"Roll back" the preempted to state before resource was acquired (i.e.,)

Example: thread wants to increment x
1. acquire lock
2. read x
3. locally increment value
4. write x
5. release lock.

- To preempt a thread here.

- Rolling back can lead to "livelock":
  - Threads seem to make progress, have to continually roll back, never actually finish.

- Example of optimistic concurrency: proceed with hope of no conflict, fix if it occurs.

- Can't roll back after doing 1/0.
3. Fixing hold and wait

- Require threads to acquire all resources at once.
- Doesn't work if first resource is needed to determine what next resource is.

ex:

```
T1
lock file 1
read filename 2 from file 1
lock file 2
read file 2
release locks.
```

```
lock file 1
read filename 2 from file 1
release file 1
lock file 2
read file 2
release file 2
```

broken reads stale file

```
not holding & waiting at same time
```

```
yes, release file 1 & acquire file 1 & file 2
reread file 1 to ensure file 2 unchanged
unchanged, read file 2, release.
```
4. Breaking circular wait
- if locks are ordered, have to acquire them in increasing order, can't have a cycle.

philosopher:
- grab lower-numbered chopstick
- grab higher-numbered chopstick
- eat()
- release
- sleep()

Can always run process holding highest resource to completion.
Deadlock detection

- Maintain representation of RAG.
- Periodically (or when resources requested/required) check for cycles.
  If found: break it by picking a process to kill / roll back.

- Efficient if no deadlock
- Easy to implement.

Safe, but requires app-specific code.
Banker's algorithm (deadlock avoidance)
- Processes pre-declare what resources they might acquire. (credit limit)
- When resource is requested, Banker blocks until enough resources are available to guarantee completion w/o deadlock. (stress test)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P2</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- A current allocation is safe if there is some order to run proc's guarantees all complete, even if they all acquire max resources.

Ex: |   | A | B | C | D |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Free</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>P1</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P2</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- P1 can complete because it has all its resources.
- P2 can then run, because its resources are available.
- P3 has enough resources available to finish.

\[
\begin{array}{c|cccc}
\text{P1} & A & B & C & D \\
\hline
\text{P1} & 5 & 1 & 1 & 1 \\
\text{P2} & 1 & 1 & 1 & 1 \\
\text{P3} & 2 & 1 & 1 & 1 \\
\end{array}
\]

Free: 1 0 1 0

\(\downarrow\ (\text{run P1})\)

\[
\begin{array}{c|cccc}
\text{P1} & A & B & C & D \\
\hline
\text{P1} & 5 & 1 & 1 & 1 \\
\text{P2} & 1 & 1 & 1 & 1 \\
\text{P3} & 2 & 1 & 1 & 1 \\
\end{array}
\]

Free: 1 0 1 0

\(\downarrow\ (\text{run P2})\)

\[
\begin{array}{c|cccc}
\text{P1} & A & B & C & D \\
\hline
\text{P1} & 5 & 1 & 1 & 1 \\
\text{P2} & 1 & 1 & 1 & 1 \\
\text{P3} & 2 & 1 & 1 & 1 \\
\end{array}
\]

Free: 5 1 1 1

\(\downarrow\ (\text{run P3})\)

all processes done