PRINCIPLES OF COMPUTER SYSTEMS DESIGN

CSE130

Winter 2020

Concurrency IV - Deadlocks

Baskin Engineering UC SANTA CRUZ

Notices

- **Assignment 1** due 23:59 **Monday January 27** **(24 Hour Extension)**
- **Assignment 2** due 23:59 **Sunday February 2**
- **Lab 2** due 23:59 **Sunday February 9**
- Section Times Reminder
  - Monday 1-2:30pm & 6:30-8pm
  - Tuesday 5-6:30pm
  - Wednesday 2-3:30pm
  - Friday 11am-12:30pm & 12:30-2pm
- Quiz - Possible Midterm Cancellation
  - On Canvas now
  - Take it, have your say!

Today's Lecture

- Introduction to Deadlocks
- The Dining Philosophers Problem
- Race Conditions
- Resource Allocation Graphs
- Deadlock Prevention, Avoidance, and Detection

- Introduction to Assignment 2

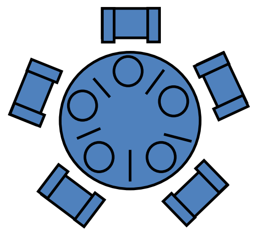
Deadlock Defined

"A situation from which it is impossible to proceed"

- Operating Systems must be designed to deal with deadlocks as resources are finite
- Not all OSs deal with deadlocks in the same way
- Best known illustration is "The Dining Philosophers Problem"

The Dining Philosophers Problem

- Five dining philosophers
- Five chopsticks
- Philosophers are either thinking, or eating
- To eat, each philosopher needs two chopsticks
- Chopsticks are put down while thinking



Codifying The Dining Philosophers Problem

- A simple solution is to represent each chopstick by a binary semaphore
- Then use an array of semaphores indexed by position
- A chopstick is picked up with:

```
P(chopsticks[pos])
```

- And set down with:

```
V(chopsticks[pos])
```


Simple Solution - Pseudo Code

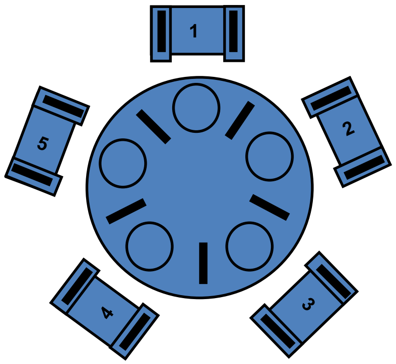
```
while (true) {
  P(chopsticks[left]);
  P(chopsticks[right]);
  // nom, nom, nom
  V(chopsticks[left]);
  V(chopsticks[right]);
  // ponder life, the universe, and everything
}
```

Problem?

Deadlocked Philosophers

- What if the philosophers all pick up their left chopstick at the same time?
  - No philosopher can grab their right chopstick (another's left chopstick) and will therefore wait forever and starve
- **This is *the* classic example of deadlock**

Not Deadlocked Philosophers



(see lecture video for animation)

Deadlocked Philosophers

(see lecture video for animation)

Race Conditions

- The dining philosophers are **in a race** to get the chopsticks
- **Race Condition**:

  "A flaw in a system or process whereby the output of the process is unexpectedly and critically dependent on the sequence or timing of other events."
- In the Pintos source code you'll see at least one comment observing that some function call sequences may be "racy"

Deadlock Prevention

- How can we stop the dining philosophers from becoming deadlocked and starving to death?
- **Possibilities**:
  - Limit the number of philosophers to 4 of 5 at the table at any one time
  - Pick up the chopsticks only if both are available
    - i.e. define a critical section
- **Best Solution**:
  - Alternate the order that chopsticks are picked up if the philosopher is seated at an odd or even seat

Self Study (totally not graded, but fun!)

- Code up The Dining Philosophers Problem using POSIX Semaphores
- See if you can get your implementation to deadlock
  - How frequently does it deadlock?
- Try the solutions from the previous page
  - Did they fix the problem?
  - How do you know they've fixed the problem?

Operating System Definitions

- **Deadlock**:
  "A set of processes is in a deadlock state when every thread or process in the set is waiting for an event that can only be caused by another thread or process in the set."
  - Threads competing for **critical sections**
  - Processes or threads competing for **resources**
    - Resource (CPU, memory, tape drives, printers, network cards…)
    - Some resources are **preemptable**, some are not
  - Resource access lifecycle:
    - **request** (wait), use, **release** (signal)
- **Starvation**:
  - A thread or process waiting forever

Deadlock => Starvation
Starvation ≠> Deadlock

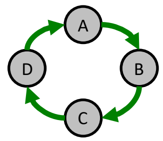
Deadlock?

```
MUTEX x = 1, y =1; // shared between threads

Thread 1        Thread 2
   P(x);           P(y);
   P(y);           P(x);
```

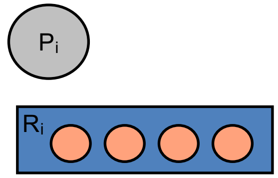

Necessary Conditions for Deadlock

- **Mutual exclusion**
  - At least one resource does not support simultaneous use
- **No preemption**
  - A resource cannot be seized from the holder
- **Hold and wait**
  - At least one process or thread must hold a resource, and be waiting on another
- **Circular wait**
  - e.g. A waits on B waits on C waits on D waits on A



Resource Allocation Graphs

- Useful in examining deadlocks
- Set of processes [P_1, P_2, ...]
- Set of resources [R_1, R_2, ...]
  - Can be more than one instance of some resources
  - Resources are identical and interchangeable



Resource Allocation Graphs

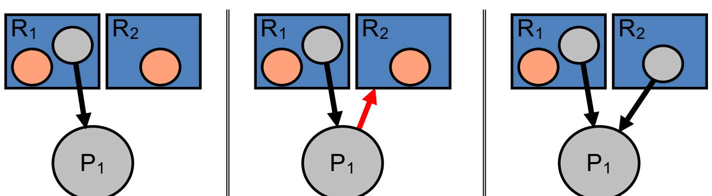
Protocol:

1. Request a resource
2. Acquire an instance of a resource (may have to wait to get it)
3. Use the resource
4. Release the resource

- Resource may be a lock, a buffer, a device, etc.
- "**Acquire**" and "**Release**" are system calls
  - Processes may need to wait on a resource
    - Typically processes wait for an event associated with the release of the resource

Constructing Resource Allocation Graphs

- Draw:
  - Edges from P_i to R_j on **request** (P_i, R_j) **"wants"** →
  - Edges from R_j to P_i on **allocation** (R_j, P_i) **"has"** →



Initial State
(P_1 has R_1)

P_1 wants R_2

P_1 has R_2
(and still has R_1)

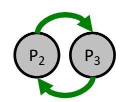
Deadlock #1

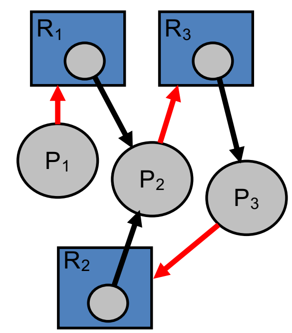
If:
- P_1 wants R_1
- P_2 wants R_3
- P_3 wants R_2

And:
- P_2 has R_1
- P_2 has R_2
- P_3 has R_3

Then:
- P_1 waiting for R_1
- P_2 waiting for R_3
- P_3 waiting for R_2

Deadlock? **YES!**





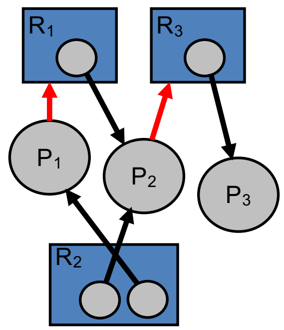
Circular Wait between P_2 and P_3 as only one instance of R_2

Deadlock #2

P_1 waiting for R_1
P_2 waiting for R_3
P_3 not waiting for anything
=> P_3 runnable

Deadlock? **NO!**

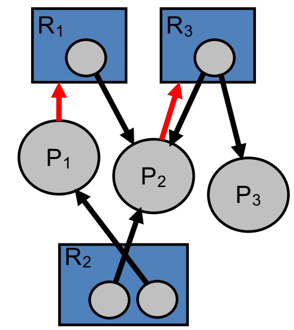
But there may be a deadlock a little later…
…it depends of what happens next



Deadlock #2 - Possibility 1

P_3 releases R_3
P_2 acquires R_3
=> P_2 runnable

Deadlock? **NO!**



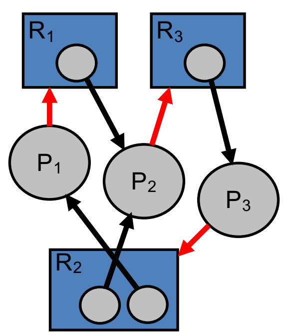
(see lecture video for animation)

Deadlock #2 - Possibility 2

P_3 requests R_2

Deadlock? **YES!**

Circular Waits between P_1, P_2, P_3 and P_3, P_2 even though two instances of R_2



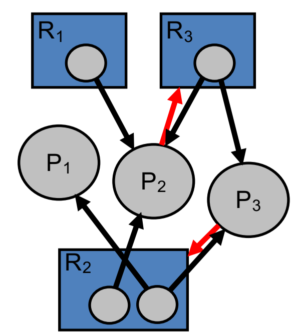
(see lecture video for animation)

Is Circular Wait Sufficient for Deadlock?

P_2 and P_3 are in a circular wait
P_1 releases R_2
=> P_2 and P_3 no longer in a circular wait
P_3 acquires R_2
=> P_3 runnable

Deadlock? **NO!**

P_3 releases R_2 and R_3
P_2 acquires R_3
=> P_2 runnable



(see lecture video for animation)

Handling Deadlocks

- **Ignore** them (deadlocks are rare)
  - Cheap, can rely on manual detection
  - Avoids performance hits
- Make sure they don't happen:
  - **Prevention:** Limit ways to request resources
  - **Avoidance:** Allocate resources to stop them happening
- **Detect** them and recover

Deadlock Prevention

- Eliminate **mutual exclusion**:
  - Not generally possible
- Eliminate **no preemption:**
  - If we wait on a resource held by another waiting process, then preempt the resource and allocate to us
- Eliminate **hold and wait**:
  - If waiting on a resource, release all currently held resources
    - Resource utilisation may be poor
    - Starvation may occur with popular resources
- Eliminate **circular wait**:
  - Impose progressive ordering for resources requests $R_1, R_2, \ldots, R_n$
  - Multiple resources of the same type must be requested together

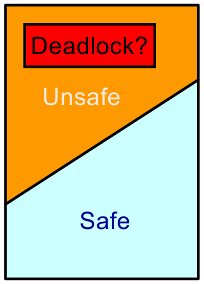
Deadlock Prevention

- Difficult to achieve without it leading to low resource utilization which leads to low throughput, i.e. poor performance
- Operating Systems generally do not try to prevent deadlocks

Deadlock Avoidance

"A resource allocation is safe if (and only if) there exists an execution sequence <P_a P_b .... P_n> such that all processes can run to completion"

- Takes into account resources the process might use
- All other states are unsafe
- From a safe state the system can allocate resources to avoid deadlock
- Requires a priori knowledge of maximum resources required by each process



Deadlock Avoidance - Example

12 instances of a single resource are available:

max = most instances of the resource a process $\mathbf{P_n}$ might ask for
alloc = instances of the resource currently allocated to $\mathbf{P_n}$
free = instances not currently allocated to any process
risk = instances $\mathbf{P_n}$ could ask for in next timeslot

	max	alloc	risk
P_0	10	5	5
P_1	4	2	2
P_2	9	2	7
	free	3	

<P_1, P_0, P_2> is **safe** ☺
When P_1 finishes, 5 free, P_0 can run
When P_0 finishes, 10 free, P_2 can run

	max	alloc	risk
P_0	10	5	5
P_1	4	2	2
P_2	9	**3**	**6**
	free	**2**	

All sequences are **unsafe** ☹
Deadlock if P_0 requests 5
Deadlock if P_2 requests 6
(only 4 free when P_1 finishes)

Making Sure Deadlocks Don't Happen

- "Unsafe" does not mean a deadlock *will* occur, rather there is a *possibility* of deadlock
- It's about guaranteeing deadlocks don't happen:
  - State max resource requirements up-front
  - Don't grant $\mathbf{R_i}$ to $\mathbf{P_j}$ unless the resulting state remains safe
    - Can allocate max if needed
- However, resource utilization may be negatively impacted
  - There's always an overhead to safety checking
- Various approaches:
  - Resource Allocation Graphs
  - Dijkstra's Bankers' Algorithm (self study, Google it!)

Deadlock Detection

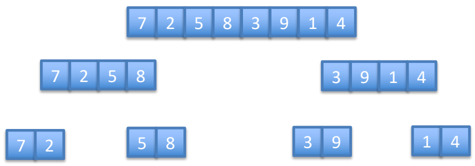
- If we don't want to, or can't avoid / prevent deadlocks, we can detect them and do something about it:
- **Break the circular wait**:
  - Option 1: Kill all deadlocked processes/threads
  - Option 2: Kill deadlocked processes/threads one-by-one
    - After each kill, see if the deadlock is broken
  - Remember, some processes should not be killed!
- **Preempt resources**:
  - Rollback to a checkpoint
  - Run the process in a different sequence

Self Study (who am I kidding, no one is actually going to do this)

- Modify your code for the Dining Philosophers Problem to detect deadlock and either **break the circular wait** or **preempt** the chopsticks and rollback

Assignment 2

- Multi Threaded Merge Sort



https://opendsa-server.cs.vt.edu/embed/mergesortAV

POSIX Thread Functions (not a complete list)

		Process "equivelant"
`pthread_create`	Create a new thread	`fork`
`pthread_join`	Join with terminated thread	`wait`
`pthread_exit`	Terminate calling thread	`exit`

http://man7.org/linux/man-pages/man2/shmget.2.html

Next Lecture

- CPU Scheduling I