CS3841 – Deadlock

Situation:

- There are lots of resources on the system that we'd like to share access:
  - Memory
  - CPU
  - I/O Devices
  - Files
- The operating system provides several mechanisms mutually exclusive access to these resources
  - Semaphores
  - Mutexes
  - Condition Variables

Deadlock

- Problem
  - What if a process tries to get exclusive access to a resource held by another process and vice versa?

```
void* thread1(void*) {
        pthread_mutex_lock(&lock1);
        // Do work, wait
        pthread_mutex_lock(&lock2);
        // Do work, wait
        pthread_mutex_unlock(&lock2);
        pthread_mutex_unlock(&lock1);
        return NULL;
}
```

```
void* thread2(void*) {
        pthread_mutex_lock(&lock2);
        // Do work, wait
        pthread_mutex_lock(&lock1);
        // Do work, wait
        pthread_mutex_unlock(&lock1);
        pthread_mutex_unlock(&lock2);
        return NULL;
}
```

- Deadlock exists among a set of processes if every process is waiting for an event that can be caused only by another process in the set

Conditions for Deadlock

All the following must happen for deadlock to occur:

- Mutual exclusion
  - Only one process may use a resource at a time
  - No process may access a resource unit that has been allocated to another process
- Hold and wait
  - A process may hold allocated resources while awaiting assignment of other resources
- No preemption
  - No resource can be forcibly removed from a process holding it
- Circular wait
  - A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain

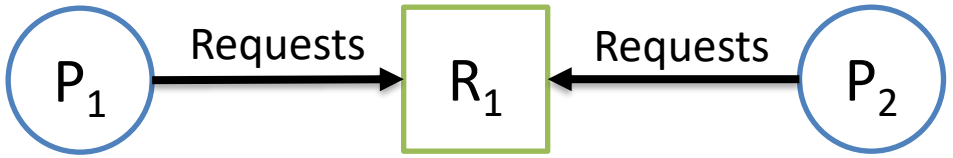
Deadlock Detection

- Resource Allocation Graph
  - Processes (P) and Resources (R)
  - An arrow pointing from P to R indicates a request for the resource

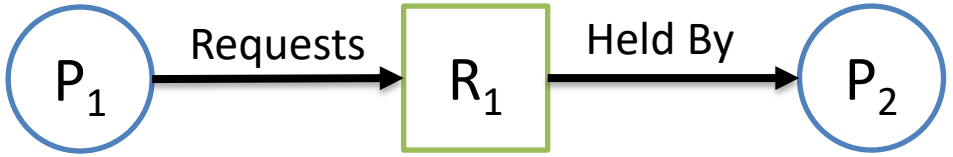
  - An arrow pointing from R to P indicates the resources is held

  - Create a graphs for all resources held and requested by all processes
  - Deadlock ***may*** exist if there is a cycle

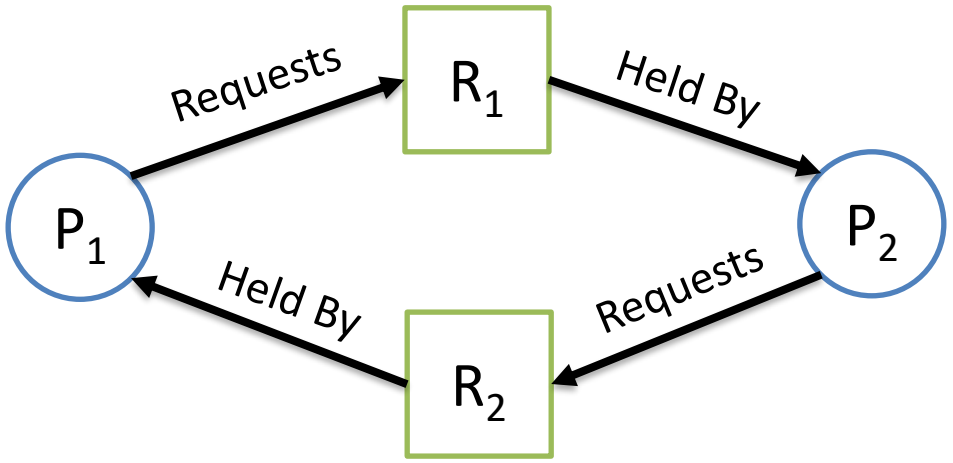
Resource Allocation Graphs



Deadlock? No



Deadlock? No



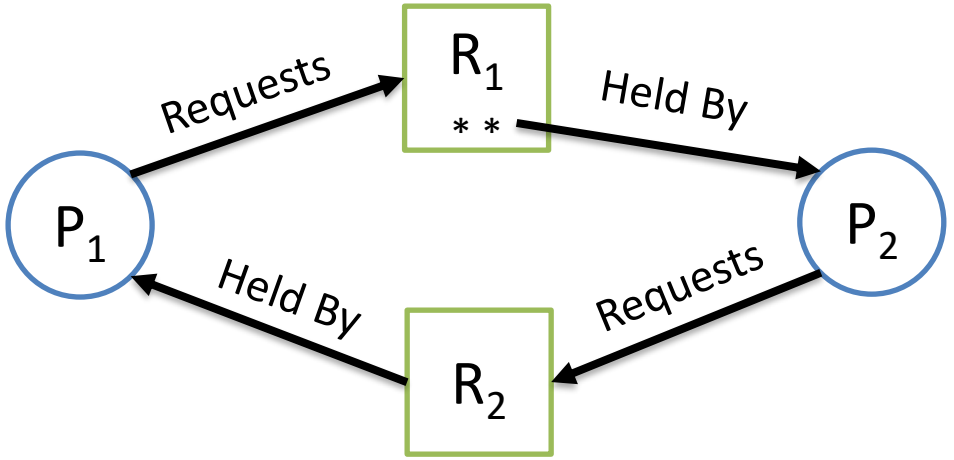
Deadlock? Yes

P_1 is requests R_1 which is held by P_2 which requests R_2 which is held by P_1

Resource Allocation Graphs

- What if you have more than one instance of a resource?
- Does a cycle detect deadlock in that case?
- Consider a semaphore where the value > 1

- There are two instances of R_1
  - R_1 is requested by P_1
  - One instance is held by P_2
  - There is a cycle but no deadlock



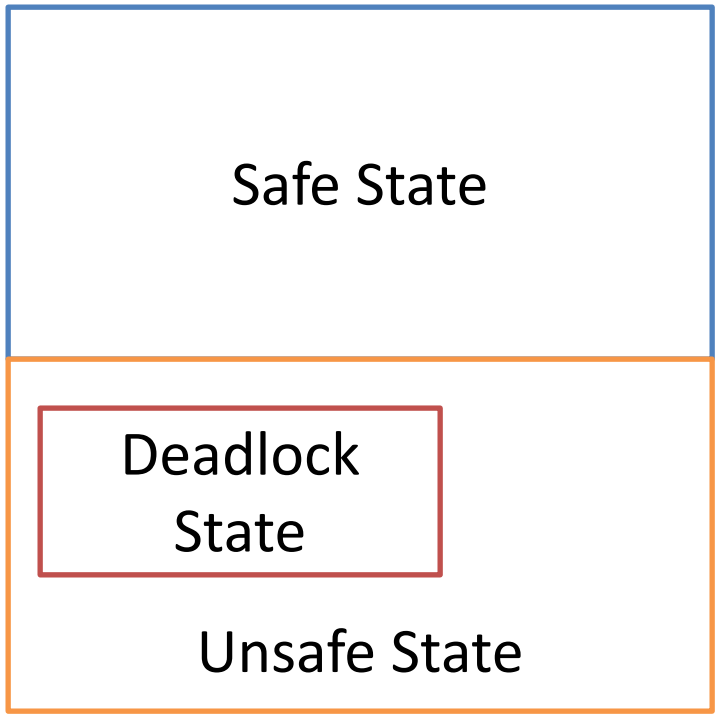
Deadlock Avoidance

- Conditions for deadlock
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait

- Eliminate any of the conditions and deadlock can’t happen

Deadlock Avoidance

- Don’t let a process request a resource if it could cause deadlock
- System state safety
  - Safe state – deadlock is not possible
  - Unsafe state – deadlock is possible (although might not be happening)
- Goal of deadlock avoidance - Never let the system get into an unsafe state



Banker's Algorithm

- Edsger Dijkstra – circa 1977
- Based on bank lenders – Don't loan money to someone if it would cause you to eventually deplete all your lending resources

- Main Idea
  - Processes will request and release resources throughout their lifetime
  - Assumes there is a maximum amount of resources that a process will ever request
  - Never allow the system to get into a situation where resources are depleted even if ALL processes request their max

Banker's Algorithm

Safe state algorithm

- For each process keep track of a flag indicating if they finished
- For each process check to see if they will finish if their max resources are requested
- If a process finishes add their allocations back what's available and mark the finished flag
- Keep iterating until there are no changes to the finished flags
- State is safe if all finished flags are true

Banker's Algorithm - Example

- A system has the following instances of resources -> 10 of A, 5 of B, and 7 of C
- The following shows the current allocations and max possible requests for 5 processes:

Allocated

	A	B	C
P_0	0	1	0
P_1	2	0	0
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2

Max

	A	B	C
P_0	7	5	3
P_1	3	2	2
P_2	9	0	2
P_3	2	2	2
P_4	4	3	3

Total

A	B	C
10	5	7

- Is the state of the system safe?

Banker's Algorithm - Example

- A system has the following instances of resources -> 10 of A, 5 of B, and 2 of C

Allocated

	A	B	C
P_0	0	1	0
P_1	2	0	0
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2

Max

	A	B	C
P_0	7	5	3
P_1	3	2	2
P_2	9	0	2
P_3	2	2	2
P_4	4	3	3

Potential Need

	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

Remaining

A	B	C
3	3	2

- Is the state of the system safe?

Banker's Algorithm - Example

Allocated

	A	B	C
P_0	0	1	0
P_1	2	0	0
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2

Max

	A	B	C
P_0	7	5	3
P_1	3	2	2
P_2	9	0	2
P_3	2	2	2
P_4	4	3	3

Potential Need

	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

Remaining

A	B	C
3	3	2

- Process P_0 requests 1 instance of A and 2 instances of C
- Should the request be allowed?
- Check – attempt the allocation and check if the resulting state is safe

Banker’s Algorithm - Disadvantages

- Not always possible to know the max resources a process will request
- Assumes processes will eventually release held resources
- Assumes a static number of processes

Deadlock Recovery

- What do you do when you have deadlock?
- Three main approaches
  - Terminate the processes
    - Terminate all
    - Terminate one by one - priority based termination
    - Problem: how to do you pick a process (or set of processes) to terminate
  - Preempt the resources
    - Rollback execution to a safe state
    - Problems
      - How do you find a safe state?
      - Starvation - how do you ensure progress?
  - Do nothing