DEADLOCKS

Ruben Acuña

Spring 2018

2 SYSTEM MODEL (7.1)

MODELING DEADLOCKS

- We will say that a system has n running processes P_0 … P_n, and m resource types R_0 … R_n. A resource may be unique, in which case some R_i has only one instance, otherwise we say there are o instances W_0 … W_o of that resource.
- Using a resource involves three steps: request, use, and release.

- What was the example of a concrete deadlock situation we saw earlier?
- How are the three steps above defined in that example?

4 DEADLOCK CHARACTERIZATION (7.2)

MUTEX SAMPLE

Is it possible for these threads to deadlock? How?

– and –

Is it possible for these threads to run without deadlocking? How?

```
/* thread one runs in this function */
void *do_work_one(void *param) {
        pthread_mutex_lock(&first_mutex);
        pthread_mutex_lock(&second_mutex);

        // Do some work

        pthread_mutex_unlock(&second_mutex);
        pthread_mutex_unlock(&first_mutex);
        pthread_exit(0);
}

/* thread two runs in this function */
void *do_work_two(void *param) {
        pthread_mutex_lock(&second_mutex);
        pthread_mutex_lock(&first_mutex);

        // Do some work

        pthread_mutex_unlock(&first_mutex);
        pthread_mutex_unlock(&second_mutex);
        pthread_exit(0);
}
```

NECESSARY CONDITIONS

For deadlock to occur, a system must demonstrate the following behaviors:

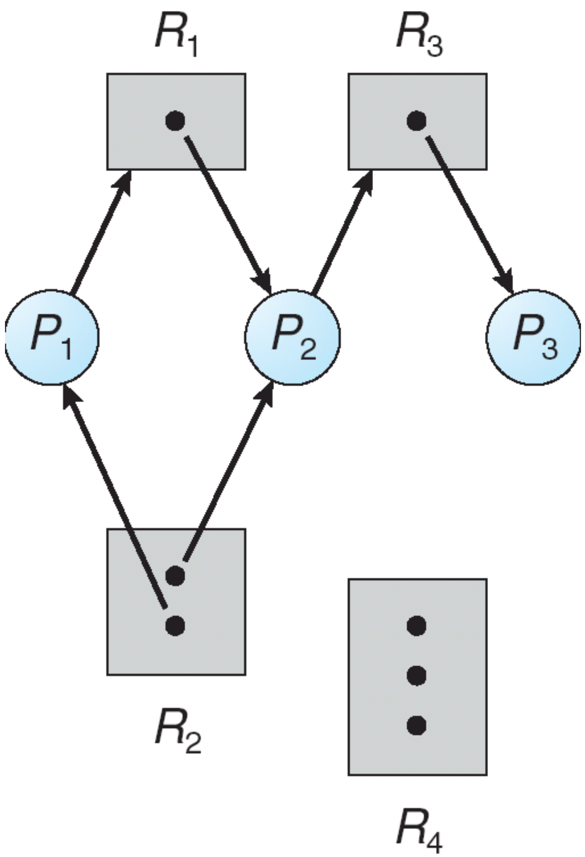
1. Mutual exclusion
2. Hold and wait
3. No preemption
4. Circular wait

How does each of these lead to a deadlock state?

If we can address any of these, then we should have a system that won't be held back by deadlocks.

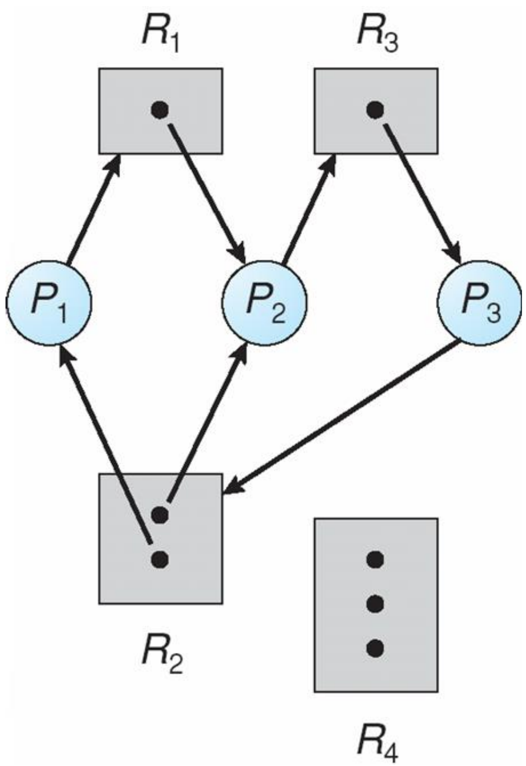
RESOURCE ALLOCATION GRAPHS

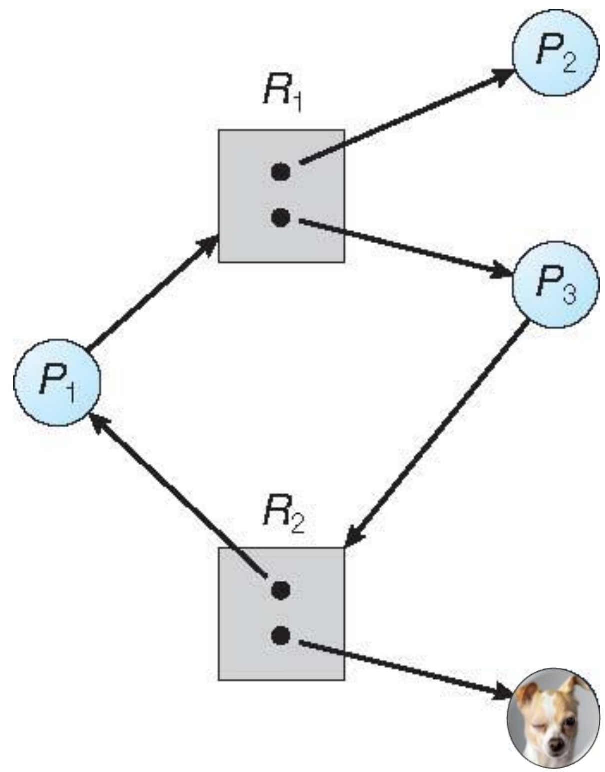
- Consider constructing a graph with nodes that correspond to elements of P and R.
- An edge from P_i to R_j represents that P_i is *requesting* an instance of R_j.
- An edge from $W_k \in R_j$ to P_i represents that resource instance is *assigned* to P_i.
- For visualization purposes we show processes as circles and resources as rectangles.
  - Each resource is a supernode that contains each instance of the resource type. Each dot within it represents a specific resource instance.



RESOURCE ALLOCATION GRAPHS

- A resource allocation graph can potentially tell us if there is a deadlock in a system.
- Does the left graph have a deadlock?
- Any general rule?





9

METHODS FOR HANDLING DEADLOCKS (7.3)

METHODS

- There are a few ways we can
  - *Prevention*: prevent one of the four necessary conditions from being possible.
  - *Avoidance*: check to see if a new process will create a deadlock and stop it from running.
  - *Recovery*: check if a set of processes is deadlocked and kill a process if needed.
  - *None*: assuming a reboot will eventually fix everything.

DEADLOCK PREVENTION (7.4)

PREVENTION METHODS

- Note that some of these methods may not apply to every process/resource combination. A practical goal is to prevent as many deadlocks as possible and then "do nothing" for the rest.

- Mutual Exclusion: instead, try to support sharable resources.

- Hold and Wait: prevent a process from acquiring a resource and then waiting for another to become available. Can require process to acquire entire sets of resources, instead of doing so piecewise.

PREVENTION METHODS

- No Preemption: instead, provide a method to allow resources to preempted by another process. Prevent sources from being about to hold on to some resources when they can't obtain all that they need to execute.

- Circular Wait: Can require all processes in the system to request resources in the same order. Hmm…

MUTEX SAMPLE PART 2

Any ideas how we can fix the example from earlier?

```
/* thread one runs in this function */
void *do_work_one(void *param) {
        pthread_mutex_lock(&first_mutex);
        pthread_mutex_lock(&second_mutex);

        // Do some work

        pthread_mutex_unlock(&second_mutex);
        pthread_mutex_unlock(&first_mutex);
        pthread_exit(0);
}

/* thread two runs in this function */
void *do_work_two(void *param) {
        pthread_mutex_lock(&second_mutex);
        pthread_mutex_lock(&first_mutex);

        // Do some work

        pthread_mutex_unlock(&first_mutex);
        pthread_mutex_unlock(&second_mutex);
        pthread_exit(0);
}
```

MUTEX SAMPLE 2

- Now, the last sample was problematic because we had two threads executing functions with different orders for resource acquisition.
- This sample is a single function, and it runs in two threads.
- Will everything be fine?

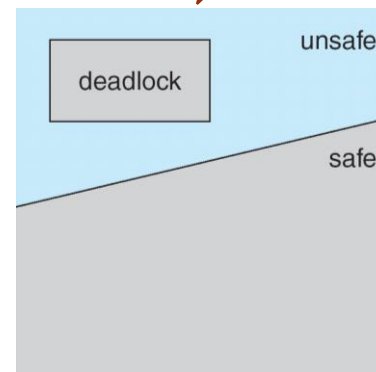
```
void transaction(Account from, Account to,
                 double amount) {
    mutex lock1, lock2;
    lock1 = get_lock(from);
        lock2 = get_lock(to);
            acquire(lock1);
            acquire(lock2);
            withdraw(from, amount);
            deposit(to, amount);
        release(lock2);
    release(lock1);
}
```

16

DEADLOCK AVOIDANCE (7.5)

This diagram is from the textbook. It has only been included so Ruben can rant about how poor a visualization it is.

SAFE VS UNSAFE SYSTEM STATES



- For this section, we will consider the case where a process can share information on how it will access resources. Here, that means maximum usage.
- If the process's usage would lead to deadlock, then we can avoid it systematically.
- Definition: a *safe* system is system with *safe sequence* $\langle P_1, P_2, \dots, P_n \rangle$, where P_i can be satisfied by available resources plus resources held by $P_j, s.t. j < i$.
- If a system is safe, then there is no deadlock. If it is unsafe, then a deadlock may occur.
- Consider the following two allocations: are the systems in a safe state? Assume 12 resources exist.

  - Sample 1

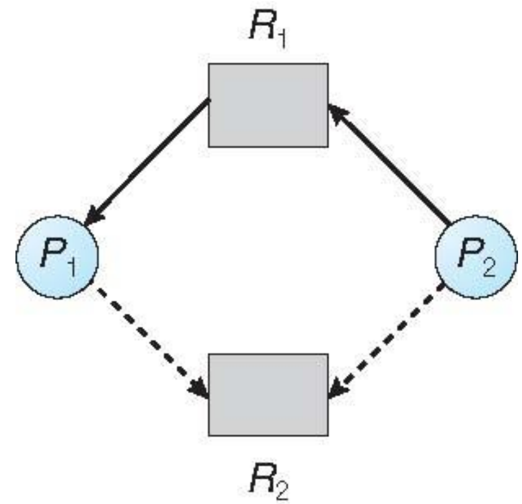
	Maximum Needs	Currently Holding
P0	10	5
P1	4	2
P2	9	2

  - Sample 2

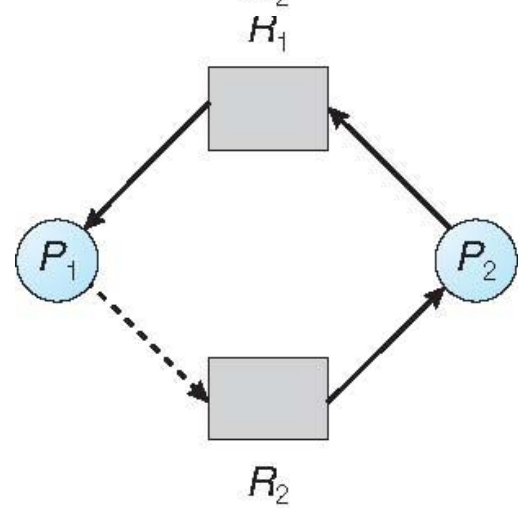
	Maximum Needs	Currently Holding
P0	10	5
P1	4	2
P2	9	4

RESOURCE ALLOCATION GRAPH ALGORITHM

- Can extend the previous concept of a resource allocation graph with the idea of a claim edge.
  - We will show these as dashed edges.



- A claim edge, $P_i \rightarrow R_j$ represents that P_i may request R_j in the future.
- When the resource is made, the claim edge may be converted to an assignment edge.



- Claim edges give us a new way to prevent deadlocks since we can check if converting them to a assignment edge would introduce a cycle into the graph.

BANKER'S ALGORITHM

- This is an algorithm that can analyze systems with multiple resource instances and if a process will create a deadlock.
- Basic idea: make a process define its resource resources before committing to its execution, and see if allocating those resources would give an unsafe state.
- We must represent *available* resources, *maximum* resources required by processes, resources currently *allocated* to processes, and remaining resources *needed*.
- Let n be the number of processes and m be the number of resource types.
- For a matrix, we will use subscript to refer to a row vector of it.

$$Max = \begin{matrix} a_{00} & \dots & a_{0m} \\ \dots & & \dots \\ a_{n0} & \dots & a_{nm} \end{matrix}$$

$$Allocation = \begin{matrix} a_{00} & \dots & a_{0m} \\ \dots & & \dots \\ a_{n0} & \dots & a_{nm} \end{matrix}$$

$$Available = \begin{matrix} a_0 & \dots & a_m \end{matrix}$$

$$Need = \begin{matrix} a_{00} & \dots & a_{0m} \\ \dots & & \dots \\ a_{n0} & \dots & a_{nm} \end{matrix}$$

$$Need = Max - Allocation$$

SAFETY ALGORITHM

- Part one, we want to check if a system is in a safe state.

```
#step 1
Work = Available
Finish = [False] * n


#step 2
i = index such that Finish[i] = False and
                    Need_i ≤ Work
while i valid:
    #step 3
    Work = Work + Allocation_i
    Finish[i] = True

    #step 2
    i = index such that Finish[i] = False and
                        Need_i ≤ Work

#step 4
if forall i, Finish[i] = True:
  system is in safe state
```

RESOURCE REQUEST ALGORITHM

- Next, we want to check if a request can be safely granted by the system.
  - $Request_i = a_0 \quad \ldots \quad a_m$
- We need to represent which resources are being requested by each process, call it *i,* to complete.

```
validate_request(Request_i):
    if Request_i ≤ Need_i:
        if Request_i ≤ Available:
            Available = Available-Request_i
            Allocation_i = Allocation_i + Request_i
            Need_i = Need_i-Request_i
            if resulting state is safe: #safety algorithm
                finalize resource allocation
            else:
                #P_i must wait for Request_i
                restore old resource allocation state
    else:
        raise error #why?
```

SAFETY ALGORITHM EXAMPLE 1

	Allocation	Request
	ABC	ABC
P0	010	753
P1	200	322
P2	302	902
P3	211	222
P4	002	433

	Need
	ABC
P0	
P1	
P2	
P3	
P4	

Available
ABC
332

SAFETY ALGORITHM EXAMPLE 2

	Allocation	Request
	ABC	ABC
P0	010	753
P1	200	322
P2	302	902
P3	211	222
P4	002	433

	Need
	ABC
P0	743
P1	122
P2	600
P3	011
P4	431

Available
ABC
032

24

DEADLOCK DETECTION (7.6)

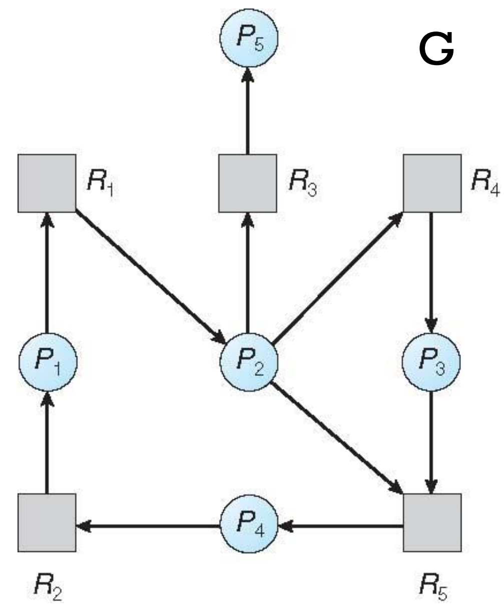
DETECTION

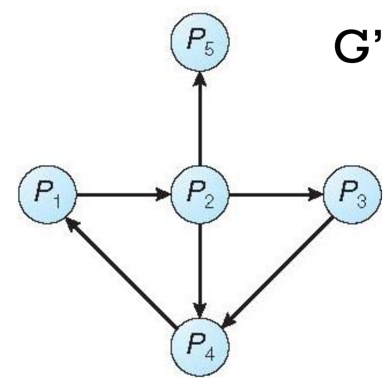
- In some systems, avoiding deadlocks may be expensive, in which case we could instead try to address them after they have occurred. This has two parts:
  - Detection
    - We'll discuss this for single and multiple resource instance systems.
  - Recovery

- Need to think about how to use a detection-algorithm. They have some cost so we want to minimize how often it is run.
  - How often should we look for deadlocks?

DETECTION WITH SINGLE RESOURCES

- We now introduce the idea of a *wait-for* graph.
- Previously we defined resource allocation graphs as the digraph: $G = (V, E) = (P \cup R \cup W, P \times R \cup W \times P)$.
- For some G, the wait-for graph is: $G' = (P, E')$, where $E' = \left(\left(P_i, P_j\right) \middle| \, (P_i, R_k) \in E \ \vee \ (R_k, P_j) \in E \right)$.
- What is the advantage of using this over a resource allocation graph?
- What about disadvantages?





DETECTION WITH RESOURCE INSTANCES

- Let *n* be the number of processes and *m* be the number of resource types.
- For a matrix, we will use subscript to refer to a row vector of it.
- We need to represent *available* resources, resources currently *allocated* to processes, and which resources are being *request* by each process to complete.

$$Available = \begin{array}{ccc} a_0 & \dots & a_m \end{array}$$

$$Allocation = \begin{array}{ccc} a_{00} & \dots & a_{0m} \\ \dots & & \dots \\ a_{n0} & \dots & a_{nm} \end{array}$$

$$Request = \begin{array}{ccc} a_{00} & \dots & a_{0m} \\ \dots & & \dots \\ a_{n0} & \dots & a_{nm} \end{array}$$

```
#step 1
Work = Available
Finish = [False] * n
for each Allocation_i = 0:
    Finish[i] = True

#step 2
i = index such that Finish[i] = False and
                    Request_i ≤ Work
while i valid:
    #step 3
    Work = Work + Allocation_i
    Finish[i] = True

    #step 2
    i = index such that Finish[i] = False and
                        Request_i ≤ Work

#step 4
if for some i, Finish[i] = False:
  system is deadlocked on process i
else
  system is not deadlocked
```

DEADLOCKS WITH INSTANCES: EXAMPLE 1

```
#step 1
Work = Available
Finish = [False] * n
for each Allocation_i = 0:
    Finish[i] = True

#step 2
i = index such that Finish[i] = False and
                    Request_i ≤ Work
while i valid:
    #step 3
    Work = Work + Allocation_i
    Finish[i] = True

    #step 2
    i = index such that Finish[i] = False
and
                        Request_i ≤ Work

#step 4
if for some i, Finish[i] = False:
  system is deadlocked on process i
else
  system is not deadlocked
```

	Allocation	Request
	AB	AB
P0	10	01
P1	01	10

Available
AB
00

DEADLOCKS WITH INSTANCES: EXAMPLE 2

```
#step 1
Work = Available
Finish = [False] * n
for each Allocation_i = 0:
    Finish[i] = True

#step 2
i = index such that Finish[i] = False and
                    Request_i ≤ Work
while i valid:
    #step 3
    Work = Work + Allocation_i
    Finish[i] = True

    #step 2
    i = index such that Finish[i] = False
and
                        Request_i ≤ Work

#step 4
if for some i, Finish[i] = False:
  system is deadlocked on process i
else
  system is not deadlocked
```

	Allocation	Request
	ABC	ABC
P0	010	000
P1	200	202
P2	303	000
P3	211	100
P4	002	002

Available
ABC
000

DEADLOCKS WITH INSTANCES: EXAMPLE 3

```
#step 1
Work = Available
Finish = [False] * n
for each Allocation_i = 0:
    Finish[i] = True

#step 2
i = index such that Finish[i] = False and
                    Request_i ≤ Work
while i valid:
    #step 3
    Work = Work + Allocation_i
    Finish[i] = True

    #step 2
    i = index such that Finish[i] = False
and
                        Request_i ≤ Work

#step 4
if for some i, Finish[i] = False:
  system is deadlocked on process i
else
  system is not deadlocked
```

	Allocation	Request
	ABC	ABC
P0	010	000
P1	200	202
P2	303	001
P3	211	100
P4	002	002

Available
ABC
000

31

RECOVERY FROM DEADLOCK (7.7)

RECOVERY

- Once we've found a deadlock, we need to deal with it. We can either try to fix it by killing process(es) or preempting resources.
  - Process Termination approaches:
    - Abort all deadlocked processes
    - Abort just one process
      - Which process should we abort?
  - Resource preemption considerations:
    - Selecting a victim
    - Rollback
    - Starvation