IT308: Operating Systems

Synchronization: Deadlock characterization

Deadlock (Definition)

- A situation in which two or more threads or processes are blocked and cannot proceed
- "blocked" either on a resource request that can't be granted, or waiting for an event that won't occur

Resource deadlock: canonical example

Thread 1:

```
lock(&A);
lock(&B);
```

Thread 2:

```
lock(&B);
lock(&A);
```

Resource deadlock: canonical example

Thread 1:

lock(&A);
lock(&B);

Thread 2:

lock(&B);
lock(&A);

One possible execution timeline:

```
lock(&A); // Thread 1 acquires lock A
<context switch to Thread 2>
```

Resource deadlock: canonical example

Thread 1:

lock(&A);
lock(&B);

Thread 2:

lock(&B);
lock(&A);

One possible execution timeline:

```
lock(&A); // Thread 1 acquires lock A
<context switch to Thread 2>

lock(&B); // Thread 2 acquires lock B
lock(&A); // Thread 2 blocks because lock A is taken
<context switch to Thread 1>
```

Resource deadlock: canonical example

Thread 1:

```
lock(&A);
lock(&B);
```

Thread 2:

```
lock(&B);
lock(&A);
```

One possible execution timeline:

```
lock(&A); // Thread 1 acquires lock A
<context switch to Thread 2>

lock(&B); // Thread 2 acquires lock B
lock(&A); // Thread 2 blocks because lock A is taken
<context switch to Thread 1>

lock(&B); // Thread 1 blocks because lock B is taken
```

Resource deadlock: canonical example

Thread 1:	Thread 2:
lock(&A);	lock(&B);
lock(&B);	lock(&A);

One possible execution timeline:

```
lock(&A); // Thread 1 acquires lock A
<context switch to Thread 2>

lock(&B); // Thread 2 acquires lock B
lock(&A); // Thread 2 blocks because lock A is taken
<context switch to Thread 1>

lock(&B); // Thread 1 blocks because lock B is taken
```

Both threads are waiting for each other: they are deadlocked

System Modeling

- Set of resource types: $R_1, R_2, \ldots, R_m$
  - There are multiple resource of each type: e.g., 3 NICs, 4 disks
- Set of processes (or threads): $P_1, P_2, \ldots, P_n$

System Modeling

- Set of resource types: $R_1, R_2, \ldots, R_m$
    - There are multiple resource of each type: e.g., 3 NICs, 4 disks
- Set of processes (or threads): $P_1, P_2, \ldots, P_n$
- Each process can:
    - Request a resource of a given type and block/wait until one resource instance of that type becomes available
    - Use a resource
    - Release a resource

System Modeling

- Set of resource types: R_1, R_2, . . . , R_m
  - There are multiple resource of each type: e.g., 3 NICs, 4 disks
- Set of processes (or threads): P_1, P_2, . . . , P_n
- Each process can:
  - Request a resource of a given type and block/wait until one resource instance of that type becomes available
  - Use a resource
  - Release a resource
- In the previous slide we have two processes, P_1 and P_2 (2 threads), two resource types R_1 (one lock, which corresponds to some resource), and R_2 (another lock, which corresponds to another resource)

Necessary conditions for deadlock

- That is, what conditions need to be true (of some system) so that deadlock is possible?
  - Aka Coffman conditions
  - 3 necessary conditions ...

Necessary conditions for deadlock

- That is, what conditions need to be true (of some system) so that deadlock is possible?
  - Aka Coffman conditions
  - 3 necessary conditions ...
- Not the same as causing deadlock!

Necessary condition 1: Mutual Exclusion

- At least one resource is non-shareable; in other words at most one process at a time can use it
- In our example: the locks are mutually exclusive

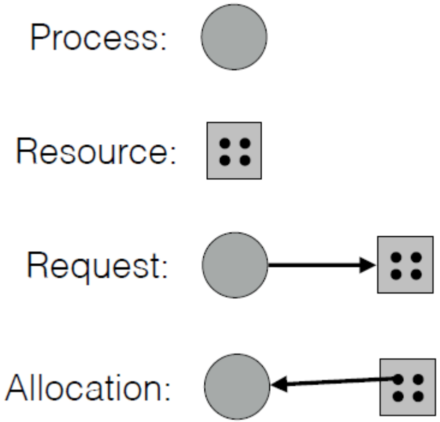
Necessary condition 2: No preemption

- Resources cannot be forcibly removed from processes that are holding them
- In our example: only the thread holding a lock can release it

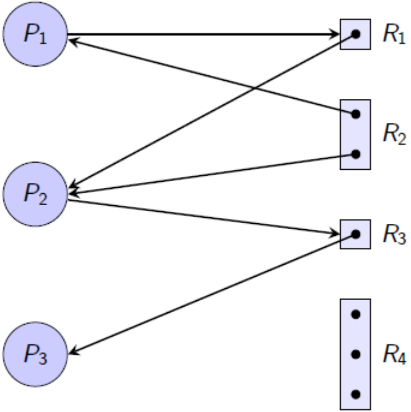
Necessary condition 3: Circular wait

- There exists a set $\{P_0, P_1, \ldots, P_p\}$ of waiting processes such that $(\forall i \in \{0, 1, \ldots, p-1\})$ P_i is waiting for a resource held by P_{i+1} and P_p is waiting for a resource held by P_0
- In other words, there is a circular chain of processes such that each process holds one or more resources that are being requested by the next process in the chain
- In our example: thread 1 has lock A and needs lock B, and thread 2 has lock B and needs lock A

Resource-allocation graph



Example Graph



Cycle and deadlock

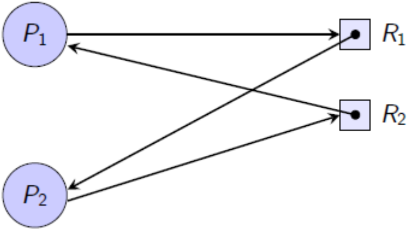
- Theorem [Holt]:
  If the resource allocation graph contains no (directed) cycle, then there is no deadlock in the system
    - If cycles do exist then a deadlock is possible

Cycle and deadlock

- Theorem [Holt]:
  If the resource allocation graph contains no (directed) cycle, then there is no deadlock in the system
  - If cycles do exist then a deadlock is possible
- If there is only one resource instance (black dot) per resource type then we have a stronger theorem:
  The existence of a cycle is a necessary and sufficient condition for the existence of a deadlock

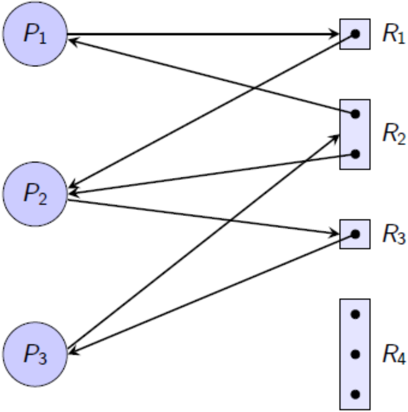
Cycle and deadlock: our 2-lock example

Clearly, there is a cycle



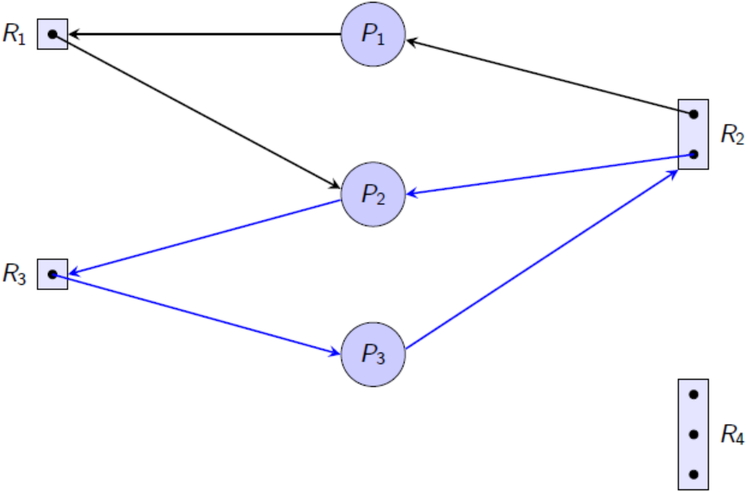
Another Example

Can you see the cycle(s)?



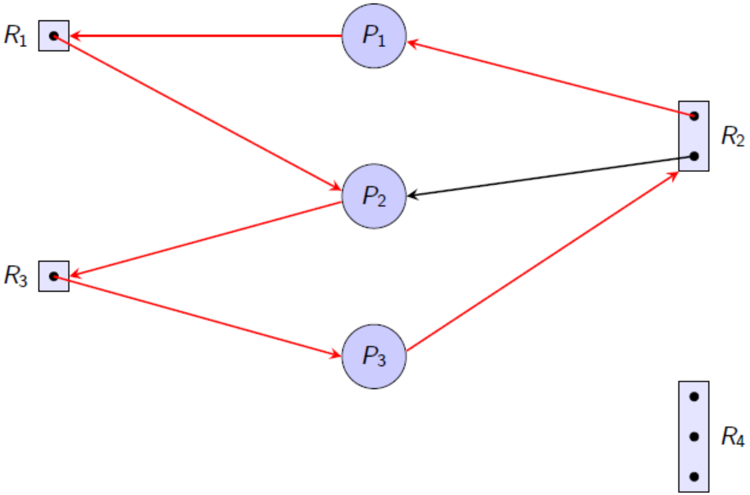
Moving vertices around

Can you see the cycle(s) now?



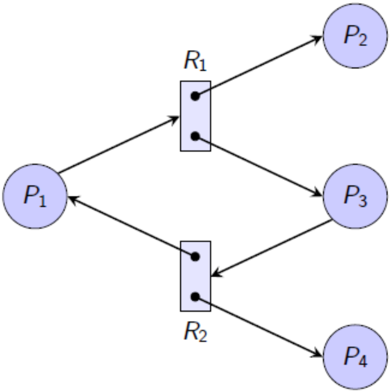
Moving vertices around

Can you see the cycle(s) now?



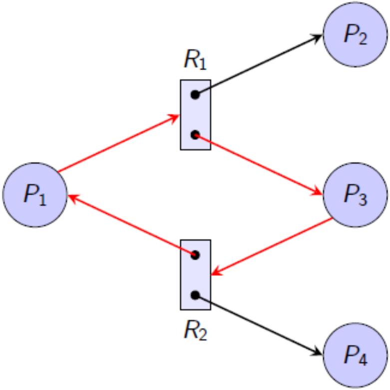
Example: Cycle and No Deadlock

There is a cycle . . .



Example: Cycle and No Deadlock

There is a cycle . . . but there is no deadlock

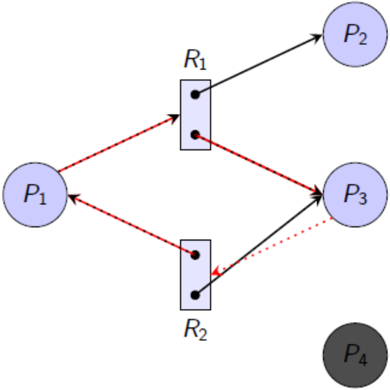


Example: Cycle and No Deadlock

When P_4 terminates it will release the instance of R_2 it locked, and that resource will be locked by P_3.

P_3 will then be able to complete.

(Another option is that P_2 completes first.)



Rule of Thumb

- A cycle in the resource allocation graph
  - Is a necessary condition for a deadlock
  - But not a sufficient condition

Deadlock Prevention

- Necessary condition 1 – Mutual Exclusion: "At least one resource is non-shareable"
  - If we make this condition not true, then we can't have a deadlock

Deadlock Prevention

- Necessary condition 1 – Mutual Exclusion: "At least one resource is non-shareable"
  - If we make this condition not true, then we can't have a deadlock
  - In other words: only use shareable resources
    - In our example, if we don't use locks, then we can't have deadlocks :)
  - The problem is that non-shareable resources are useful:
    - A critical section protected by locks, a file open for writing, etc.

Deadlock Prevention

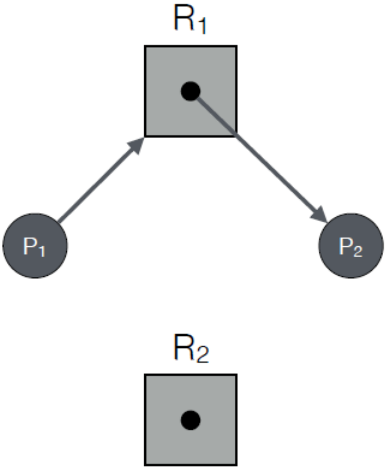
- Necessary condition 2 – No Preemption: “Resources cannot be forcibly removed”
  - If we make this condition not true, then we can't have a deadlock

Deadlock Prevention

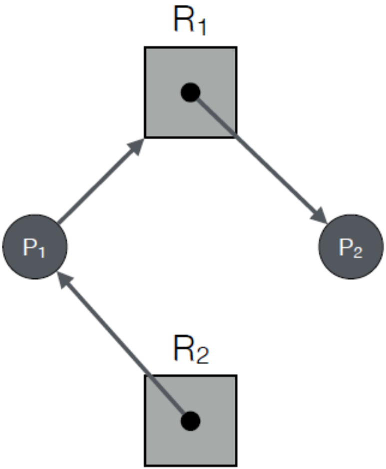
- Necessary condition 2 – No Preemption: "Resources cannot be forcibly removed"
  - If we make this condition not true, then we can't have a deadlock
  - But how do we even program in an environment in which you acquire resources but then lose them at any time?
  - And would it be safe?

Deadlock Prevention

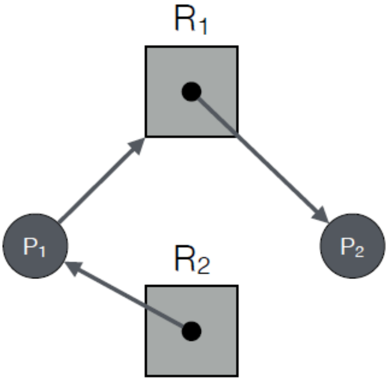
- Necessary condition 2 – No Preemption: “Resources cannot be forcibly removed”
    - If we make this condition not true, then we can’t have a deadlock
    - But how do we even program in an environment in which you acquire resources but then lose them at any time?
    - And would it be safe?
- In practice, we cannot eliminate mutual exclusion or no-premption
    - Circular Wait is where it’s at.



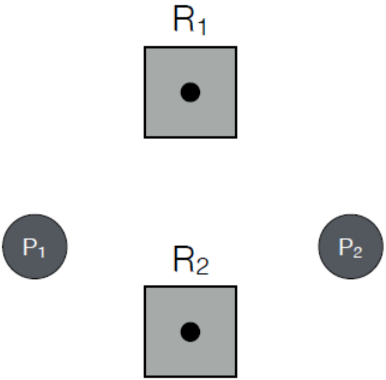
Possible to create cycle (with one edge)?



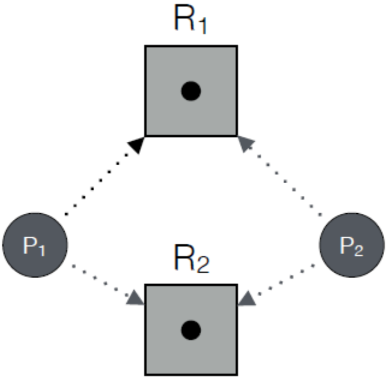
Possible to create cycle (with one edge)?



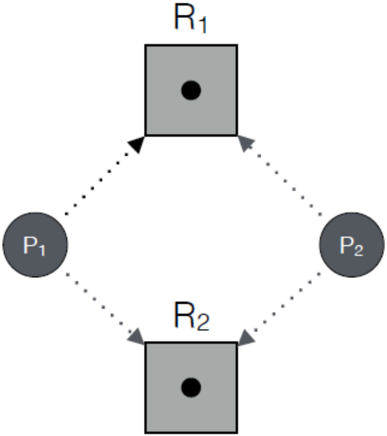
It's quite possible that P2 won't need R2, or, maybe P2 will release R1 before requesting R2, but we don't know if/when . . .



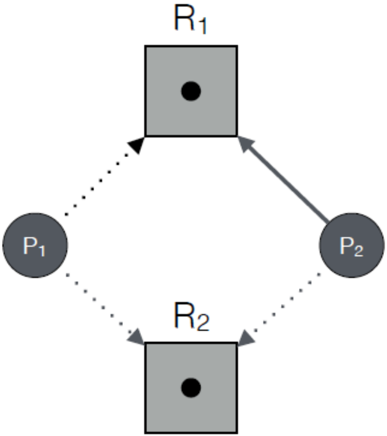
Preventing circular wait means avoiding a state where a cycle is an imminent possibility



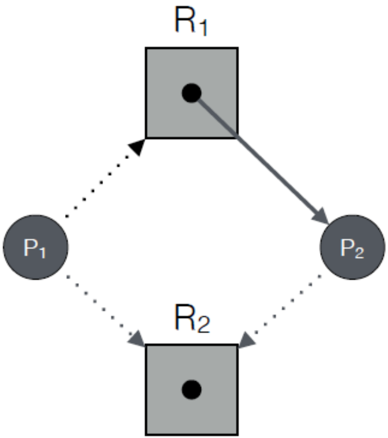
To predict deadlock, we can ask processes to "claim" all resources they need in advance



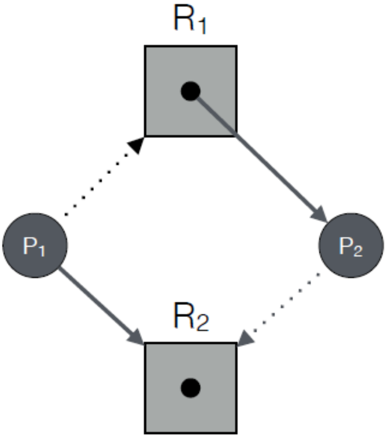
Graph with “claim edges”



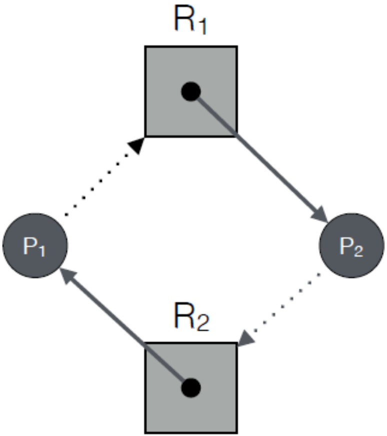
P2 requests R1



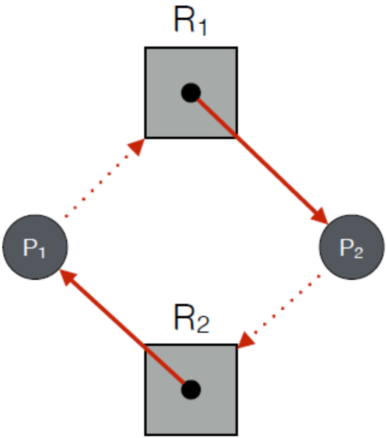
Convert to allocation edge; no cycle



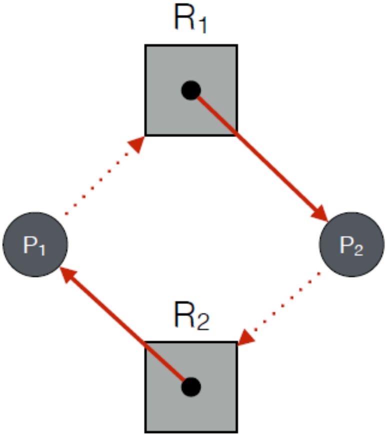
P1 requests R2



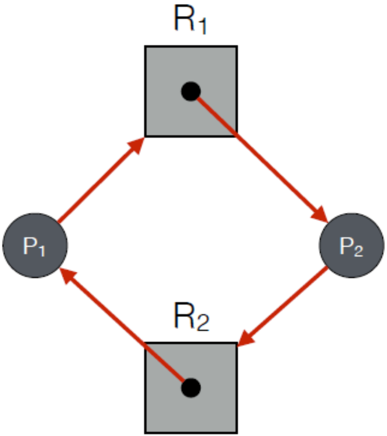
If we convert to an allocation edge . . .



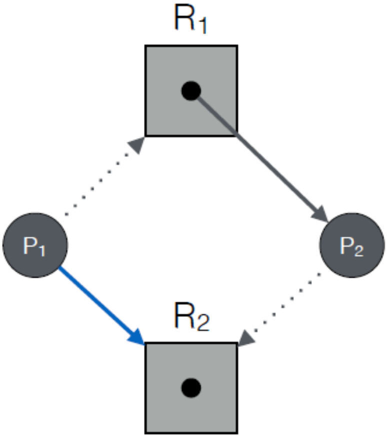
Cycle involving claim edges!



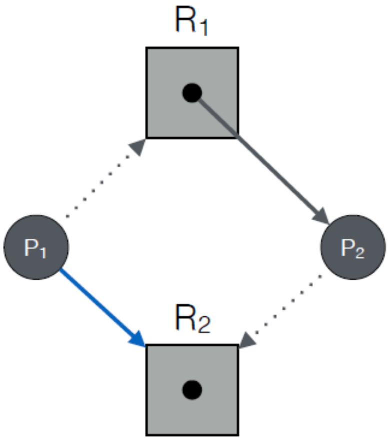
Means that if processes fulfill their claims, we can not avoid deadlock!



P1 $\rightarrow$ R1, P2 $\rightarrow$ R2



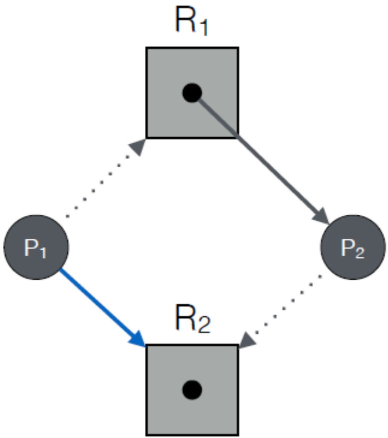
P1 $\rightarrow$ R2 should be blocked by the OS, even if it can be satisfied with available resources



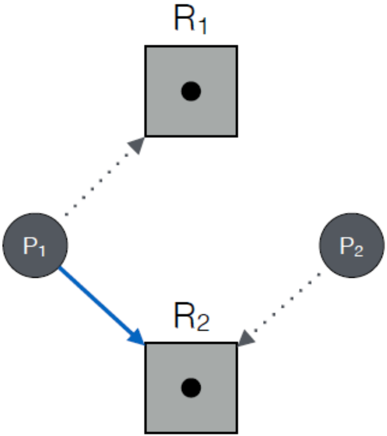
This is a “safe” state . . . i.e., no way a process can cause deadlock directly (i.e., without OS alloc)

Idea: if granting an incoming request would create a cycle in a graph with claim edges, deny that request (i.e., block the process)

- approve later when no cycle would occur

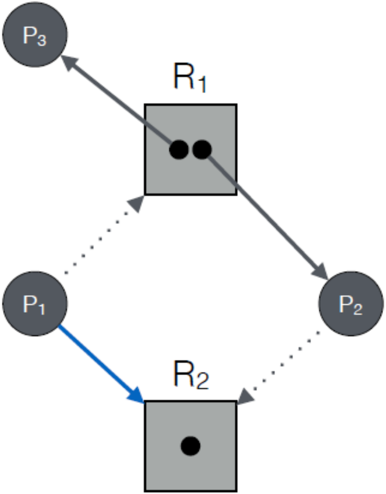


P2 releases R1



Now OK to approve P1 $\rightarrow$ R2 (unblock P1)

Problem: this approach may incorrectly predict imminent deadlock when resources with multiple instances are involved



Requires a more general definition of “safe state”

Deadlock Detection

- Goal: How can OS detect when there is a deadlock?
- OS should keep track of
  - Current resource allocation (who has what)
  - Current pending requests (who is waiting for what)
- This info is enough to check if there is a current deadlock (see next few slides)

Detecting Deadlocks

- Suppose there is only one instance of each resource
- Example 1: Is this a deadlock?
  - P1 has R2 and R3, and is requesting R1
  - P2 has R4 and is requesting R3
  - P3 has R1 and is requesting R4
- Example 2: Is this a deadlock?
  - P1 has R2, and is requesting R1 and R3
  - P2 has R4 and is requesting R3
  - P3 has R1 and is requesting R4

Detecting Deadlocks

- Solution: Build a graph, called Resource Allocation Graph (RAG)
  - There is a node for every process and a node for every resource
  - If process P currently has resource R, then put an edge from R to P
  - If process P is requesting resource R, then put an edge from P to R

Detecting Deadlocks

- Solution: Build a graph, called Resource Allocation Graph (RAG)
  - There is a node for every process and a node for every resource
  - If process P currently has resource R, then put an edge from R to P
  - If process P is requesting resource R, then put an edge from P to R
- There is a deadlock if and only if RAG has a cycle

Detecting deadlocks

- How to detect deadlocks when there are multiple instances of resources
- Example:
  - Suppose there are 2 instances of A and 3 of B
  - Process P currently has 1 instance of A, and is requesting 1 instance of A and 3 instances of B
  - Process Q currently has 1 instance of B, and is requesting 1 instance of A and 1 instance of B

Detecting deadlocks

- Suppose there are n processes P1, . . . , Pn and m resources R1, . . . , Rm
- To detect deadlocks, we can maintain the following data structures
  - Current allocation matrix C: C[i,j] is the number of instances of resource Rj currently held by process Pi

Detecting deadlocks

- Suppose there are n processes P1, . . . , Pn and m resources R1, . . . , Rm
- To detect deadlocks, we can maintain the following data structures
  - Current allocation matrix C: C[i,j] is the number of instances of resource Rj currently held by process Pi
  - Current request matrix R: R[i,j] is the number of instances of resource Rj currently being requested by process Pi

Detecting deadlocks

- Suppose there are n processes P1, . . . , Pn and m resources R1, . . . , Rm
- To detect deadlocks, we can maintain the following data structures
  - Current allocation matrix C: C[i,j] is the number of instances of resource Rj currently held by process Pi
  - Current request matrix R: R[i,j] is the number of instances of resource Rj currently being requested by process Pi
  - Availability vector A: A[j] is the number of instances of resources Rj currently free.

Deadlock detection

- Goal of the detection algorithm is to check if there is any sequence in which all current requests can be met
  - Note: If a process Pi's request can be met, then Pi can potentially run to completion, and release all the resources it currently holds. So for detection purpose, Pi's current allocation can be added to A

Example

L = {} /* List of processes that can be unblocked */

```
Allocation Matrix  C                Request Matrix R

    | R1   R2   R3                      | R1   R2   R3
---------------                    ---------------
P1  | 1    1    1                 P1  | 3    2    1
P2  | 2    1    2                 P2  | 2    2    1
P3  | 1    1    0                 P3  | 0    0    1
P4  | 1    1    1                 P4  | 1    1    1
```

Satisfiable request

```
A = (0, 0, 1) /* available resources */
```

Request by process i can be satisfied if the row R[i] is smaller than or equal to the vector A

After first iteration

$L = \{P3\}$

Allocation Matrix

	R1	R2	R3
P1	1	1	1
P2	2	1	2
P3			
P4	1	1	1

A = (1, 1, 1)

Request Matrix

	R1	R2	R3
P1	3	2	1
P2	2	2	1
P3			
P4	1	1	1

Satisfiable request

Note: P3's allocation has been added to A

After second iteration

L $= \{P3, P4\}$

Allocation Matrix

	R1	R2	R3
P1	1	1	1
P2	2	1	2
P3			
P4			

Request Matrix

	R1	R2	R3
P1	3	2	1
P2	2	2	1
P3			
P4			

Satisfiable request

A $= (2, 2, 2)$.

Algorithm

```
L = EmptyList; /* processes not deadlocked */
repeat
   s = length(L);
   for (i=1; i<=n; i++){

     if (!member(i,L) && R[i] <= A) {
        /* request of process i can be met */

        A = A + C[i];
       /* reclaim resources held by process i */

        insert(i,L);
     }
    }
until (s == length(L));
/* if L does not change, then done */

if (s<n) printf("Deadlock exists");
```

Note: Running time of this algorithm is $O(n^2 m)$, where m: length of a row

Last time

- Each process provides OS with information about its requests and releases for resources
- OS determines if there is any sequence in which all current requests can be met, avoiding deadlocks

Last time

- Each process provides OS with information about its requests and releases for resources
- OS determines if there is any sequence in which all current requests can be met, avoiding deadlocks
- Assumption: processes can complete with current allocation + all current requests

Last time

- Each process provides OS with information about its requests and releases for resources
- OS determines if there is any sequence in which all current requests can be met, avoiding deadlocks
- Assumption: processes can complete with current allocation + all current requests
- i.e., no future requests
  - Unrealistic!

Deadlock avoidance

- Knowing all future requests is difficult
- Estimating a maximum demand for resources for each process is easier to produce

Deadlock avoidance

- Knowing all future requests is difficult
- Estimating a maximum demand for resources for each process is easier to produce
- simple strategy: specify a maximum claim
- A resource allocation state is defined by
  - no. of available resources
  - no. of allocated resources to each process

Deadlock avoidance

- Knowing all future requests is difficult
- Estimating a maximum demand for resources for each process is easier to produce
- simple strategy: specify a maximum claim
- A resource allocation state is defined by
  - no. of available resources
  - no. of allocated resources to each process
  - maximum demands by each process

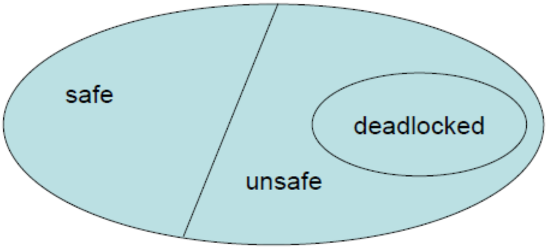
Deadlock avoidance

- A state is safe if there exists a safe sequence of processes $< P_1, \ldots, P_n >$ for the current resource allocation state
  - A sequence of processes is safe if for each P_i in the sequence, the resource requests that P_i can still make can be satisfied by:
    - currently available resources + all resources held by all previous processes in the sequence P_j, $j < i$
  - If resources needed by P_i are not available, P_i waits for all P_j to release their resources

Deadlock avoidance

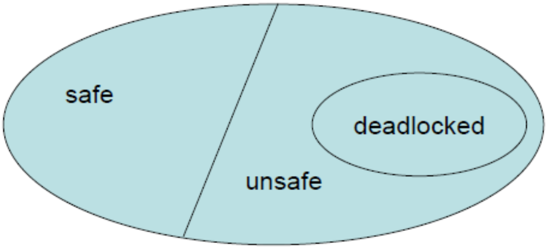
- Intuition for a safe state: given that the system is in a certain state, we want to find at least one “way out of trouble”
- i.e. find a sequence of processes that, even when they demand their maximum resources, won’t deadlock the system

Deadlock avoidance



- A deadlocked state is unsafe
- An unsafe state is not necessarily deadlocked

Deadlock avoidance



- A deadlocked state is unsafe
- An unsafe state is not necessarily deadlocked
- A system may transition from a safe to an unsafe state if a request for resources is granted

Deadlock avoidance

- Example 1:
  - 12 instances of a resource
  - At time t_0, P_0 holds 5, P_1 holds 2, P_2 holds 2
  - Available $=$ 3 free instances

processes	max needs	allocated
P0	10	5
P1	4	2
P2	9	2

Deadlock avoidance

- Example 1 (cont)
  - Claim: the sequence $< P_1, P_0, P_2 >$ is safe.
    - P_1 requests its maximum (currently has 2, so needs 2 more) and holds 4, then there is only 1 free resource

Deadlock avoidance

- Example 1 (cont)
  - Claim: the sequence $< P_1, P_0, P_2 >$ is safe.
    - P_1 requests its maximum (currently has 2, so needs 2 more) and holds 4, then there is only 1 free resource
    - Then P_1 then releases all of its held resources, so that there are 5 free

Deadlock avoidance

- Example 1 (cont)
  - Claim: the sequence $< P_1, P_0, P_2 >$ is safe.
    - P_1 requests its maximum (currently has 2, so needs 2 more) and holds 4, then there is only 1 free resource
    - Then P_1 then releases all of its held resources, so that there are 5 free
    - Next, suppose P_0 requests its maximum (currently has 5, so needs 5 more) and holds 10, so that there are 0 free

Deadlock avoidance

- Example 1 (cont)
  - Claim: the sequence $< P_1, P_0, P_2 >$ is safe.
    - P_1 requests its maximum (currently has 2, so needs 2 more) and holds 4, then there is only 1 free resource
    - Then P_1 then releases all of its held resources, so that there are 5 free
    - Next, suppose P_0 requests its maximum (currently has 5, so needs 5 more) and holds 10, so that there are 0 free
    - Then P_0 releases all of its held resources, so that there are 10 free

- Example 1 (cont)
  - Claim: the sequence $< P_1, P_0, P_2 >$ is safe.
    - P_1 requests its maximum (currently has 2, so needs 2 more) and holds 4, then there is only 1 free resource
    - Then P_1 then releases all of its held resources, so that there are 5 free
    - Next, suppose P_0 requests its maximum (currently has 5, so needs 5 more) and holds 10, so that there are 0 free
    - Then P_0 releases all of its held resources, so that there are 10 free
    - Next P_2 requests its max of 9, leaving 3 free and then releases them all

Deadlock avoidance

- Example 1 (cont)
  - Claim: the sequence $< P_1, P_0, P_2 >$ is safe.
    - P_1 requests its maximum (currently has 2, so needs 2 more) and holds 4, then there is only 1 free resource
    - Then P_1 then releases all of its held resources, so that there are 5 free
    - Next, suppose P_0 requests its maximum (currently has 5, so needs 5 more) and holds 10, so that there are 0 free
    - Then P_0 releases all of its held resources, so that there are 10 free
    - Next P_2 requests its max of 9, leaving 3 free and then releases them all
  - Thus the sequence $< P_1, P_0, P_2 >$ is safe

- Example 2:
  - at time t_1, process P_2 requests and is given 1 more instance of the resource, then

processes	max needs	allocated
P0	10	5
P1	4	2
P2	9	3

  - Available = 2 free instances
  - Is the system in a safe state?

Deadlock avoidance

- Example 2 (cont)
  - P_1 can request its maximum (currently holds 2, and needs 2 more) of 4, so that there are 0 free

- Example 2 (cont)
  - P_1 can request its maximum (currently holds 2, and needs 2 more) of 4, so that there are 0 free
  - P_1 releases all its held resources, so that available $= 4$ free

Deadlock avoidance

- Example 2 (cont)
  - P_1 can request its maximum (currently holds 2, and needs 2 more) of 4, so that there are 0 free
  - P_1 releases all its held resources, so that available $=$ 4 free
  - Neither P_0 nor P_2 can request their maximum resources (P_0 needs 5, P_2 needs 6, and there are only 4 free)
    - Both would have to wait, so there could be deadlock

Deadlock avoidance

- Example 2 (cont)
  - P_1 can request its maximum (currently holds 2, and needs 2 more) of 4, so that there are 0 free
  - P_1 releases all its held resources, so that available $= 4$ free
  - Neither P_0 nor P_2 can request their maximum resources (P_0 needs 5, P_2 needs 6, and there are only 4 free)
    - Both would have to wait, so there could be deadlock
  - The system is deemed unsafe
    - The mistake was granting P_2 an extra resource at time t_1
    - Forcing P_2 to wait for P_0 or P_1 to release their resources would have avoided potential deadlock

Deadlock avoidance

- Policy: before granting a request, at each step, perform a worst-case analysis - is there a safe sequence, a way out?
  - If so, grant request.
  - If not, delay requestor, and wait for more resources to be freed.
- Banker's Algorithm takes this approach

Deadlock avoidance

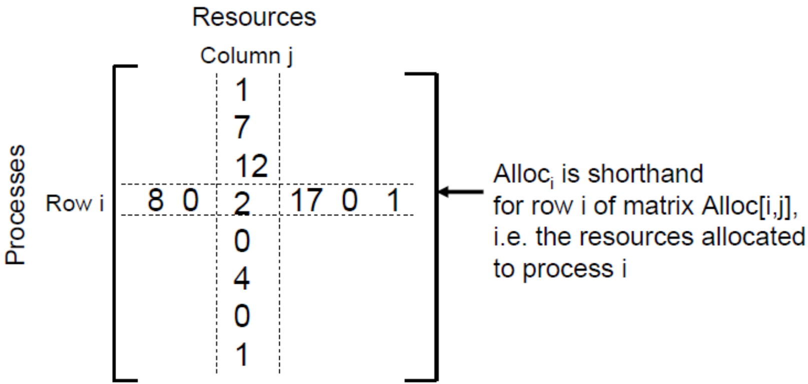
- Banker's Algorithm:
  - when there is a request, the system determines whether allocating resources for the request leaves the system in a safe state that avoids deadlock
    - if no, then wait for another process to release resources
  - each process declares its maximum demands
    - must be less than total resources in system
  - works for multiple instances of resources, unlike resource allocation graph

Deadlock avoidance

- Define quantities:
  - Available resources in a vector or list Available[j], j=1,...,m resource types
    - Available[j] = k means that there are k instances of resource Rj available
  - Maximum demands in a matrix or table Max[i,j], where i=1,...,n processes, and j=1,...,m resource types
    - Max[i,j] = k means that process i's maximum demands are for k instances of resource Rj
  - Allocated resources in a matrix Alloc[i,j]
    - Alloc[i,j] = k means that process i is currently allocated k instances of resource Rj
  - Needed resources in a matrix Need[i,j] = Max[i,j] -Alloc[i,j]

Deadlock avoidance

- An example of the Alloc[i,j] matrix:



Deadlock avoidance

- Some terminology:
  - let X and Y be two vectors. Then we say $X \leq Y$ if and only if $X[i] \leq Y[i]$ for all i.
  - Example:

$$V1 = \begin{bmatrix} 1 \\ 7 \\ 3 \\ 2 \end{bmatrix} \quad V2 = \begin{bmatrix} 0 \\ 3 \\ 2 \\ 1 \end{bmatrix} \quad V3 = \begin{bmatrix} 0 \\ 10 \\ 2 \\ 1 \end{bmatrix}$$

then $V2 \leq V1$, but $V3 \not\leq V1$, i.e. V3 is not less than or equal to V1

Deadlock avoidance

- Banker's (Safety) Algorithm: find a safe sequence, i.e. is the system in a safe state?
  1. Let Work and Finish be vectors length m and n respectively. Initialize Work = Available, and Finish[i]=false for i=0, . . . , n-1
  2. Find a process i such that both
     - Finish[i]==false, and
     - $\text{Need}_i \leq \text{Work}$

     If no such i exists, go to step 4.
  3. $\text{Work} = \text{Work} + \text{Alloc}_i$
     Finish[i] = true
     Go to step 2.
  4. If Finish[i]==true for all i, then the system is in a safe state

- Example 3:
  - 3 resources (A,B,C) with total instances available (10,5,7)
  - 5 processes
  - At time t_0, the allocated resources Alloc[i,j], Max needs Max[i,j], and Available resources Avail[j], are:

$$
\begin{array}{c}
 \\ P0 \\ P1 \\ P2 \\ P3 \\ P4
\end{array}
\begin{array}{c}
\text{Alloc[i,j]} \\
\begin{array}{ccc} A & B & C \end{array} \\
\left[\begin{array}{ccc} 0 & 1 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{array}\right]
\end{array}
\begin{array}{c}
\text{Max[i,j]} \\
\begin{array}{ccc} A & B & C \end{array} \\
\left[\begin{array}{ccc} 7 & 5 & 3 \\ 3 & 2 & 2 \\ 9 & 0 & 2 \\ 2 & 2 & 2 \\ 4 & 3 & 3 \end{array}\right]
\end{array}
\begin{array}{c}
\text{Avail[j]} \\
\begin{array}{ccc} A & B & C \end{array} \\
\left[\begin{array}{ccc} 3 & 3 & 2 \end{array}\right]
\end{array}
\begin{array}{c}
\text{Need[i,j]} \\
\begin{array}{ccc} A & B & C \end{array} \\
\left[\begin{array}{ccc} 7 & 4 & 3 \\ 1 & 2 & 2 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{array}\right]
\end{array}
$$

where Need[i,j] is computed given Alloc[i,j] and Max[i,j]

- Example 3 (cont):
  - Execute Banker's Algorithm – is the system in a safe state?
  - Yes, the sequence $< P1, P3, P4, P2, P0 >$ is safe
  - In-class Exercise