Chapter 7: Deadlock Avoidance, Banker's Algorithm

CSCI 3753 Operating Systems
William Mortl, MS and Rick Han, PhD

Recap

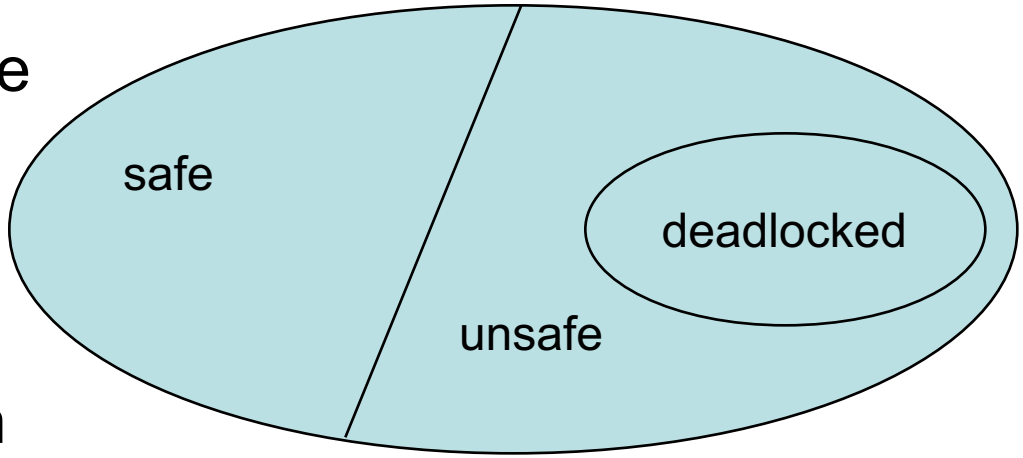
- Modeling Deadlock
- Deadlock Prevention: prevent at least 1 of 4 necessary conditions for deadlock from holding true
  1. Mutual exclusion
  2. Hold and wait
     - e.g. Release all before holding all
  3. No preemption
     - e.g. Preempt acquirer or holder
  4. Circular dependency
     - e.g. Order all resources

Recap

- Deadlock Avoidance: analyze the system state to see if there is a way to avoid deadlock
  - Given total # resources, current allocation of resources, and max demands for resources
  - find a *safe sequence* of releases and request grants
  - If even in the worst case the system can still escape deadlock, then the system is in a safe state free from deadlock

Deadlock Avoidance

- A safe state provides a safe “escape” sequence
- 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
  - ideally, check with each request for resources whether the system is still safe



Deadlock Avoidance

- Example 1 (initial state):

processes	max needs	allocated (total=12)
P0	10	5
P1	4	2
P2	9	2

- sequence <P1, P0, P2> is safe

  – P1 requests its max (has 2, so needs 2 more), holds 4,

  – then P1 releases all 4

Available = 3

Available = 1

Available = 5

Deadlock Avoidance

- Example 1 (initial state):

processes	max needs	allocated (total=12)
P0	10	5
P1	4	2
P2	9	2

- sequence <P1, P0, P2> is safe
  - P0 requests its max (has 5, so needs 5 more), holds 10
  - then P0 releases all 10

Available = 5

Available = 0

Available = 10

Deadlock Avoidance

- Example 1 (initial state):

processes	max needs	allocated (total=12)
P0	10	5
P1	4	2
P2	9	2

- sequence <P1, P0, P2> is safe
  - P2 requests its max (has 2, so needs 7 more), holds 9
  - then P2 releases all 9

Available = 10

Available = 3

Available = 12

Deadlock Avoidance

- Example 2 (initial state):

processes	max needs	allocated (total=12)
P0	10	5
P1	4	2
P2	9	**3**

- at time t1, P2 requests and is given 1 more instance of the resource
- Available = 2 instances
- Is the system in a safe state?
- Initially, P0 and P2's max needs cannot be met

Deadlock Avoidance

- Example 2 (initial state):

processes	max needs	allocated (total=12)
P0	10	5
P1	4	2
P2	9	**3**

- Best case is for P1 to request its maximum (currently holds 2, and needs 2 more) of 4, then release all 4
  - Available now = 4 vs prev available = 2
  - Satisfying P1 and releasing its resources gives P0 and P2 best chance to meet their max needs

Deadlock Avoidance

- Example 2 (initial state):

processes	max needs	allocated (total=12)
P0	10	5
P1	4	2
P2	9	**3**

- Unfortunately, Available = 4 is not enough to satisfy either P0 or P2
  - P0 has 5 and needs 5 more to meet its max
  - P2 has 3 and needs 6 more to meet its max
  - *If both P0 and P2 request their maxes, they block forever = deadlock!*

Deadlock Avoidance

- Example 2 (initial state):

processes	max needs	allocated (total=12)
P0	10	5
P1	4	2
P2	9	**3**

- The system is deemed unsafe
  - The mistake was granting P2 an extra resource at time t1
  - Forcing P2 to wait for P0 or P1 to release their resources would have avoided potential deadlock
- Though unsafe, the system is *not necessarily deadlocked yet*

Intuition: Deadlock Avoidance

- Example 2 shows a system moving from a safe to an unsafe state where there *could* be deadlock
  - No safe sequence of max requests could be found among all the possibilities
  - We found a case where there would be deadlock

Intuition: Deadlock Avoidance

- Example 1 shows that even in worst case of max demands there is not deadlock
  - The system can find an ordered way of granting max requests such that deadlock is avoided
  - At any given stage, don't grant max requests to tasks whose max requests can't be satisfied
  - Do grant max requests to remaining tasks
    - After requesting and getting its max, each such task finishes and releases its resources, adding to the pool of available resources, etc.

Dijkstra's Banker's Algorithm

- *Generalizes* deadlock avoidance to multiple resources
  - Determines whether the system is in a safe state
- Before granting a request, run Banker's Algorithm pretending as if request was granted
  - Does the worst-case analysis find such a hypothetical system is in a safe state?
  - If so, grant request.
  - If not, delay requestor, and wait for more resources to be freed.

Dijkstra's Banker's Algorithm

- As before, try to find a safe sequence
  - each process declares its maximum demands
    - must be less than total resources in system
- Advantages:
  - Generalization to multiple types of resources
  - works for multiple instances of resources, unlike resource allocation graph
  - adapts at run time to individual requests
  - individual requests don't need to be known a priori, except for maximum demands, which is not completely unrealistic

Dijkstra's Banker's Algorithm

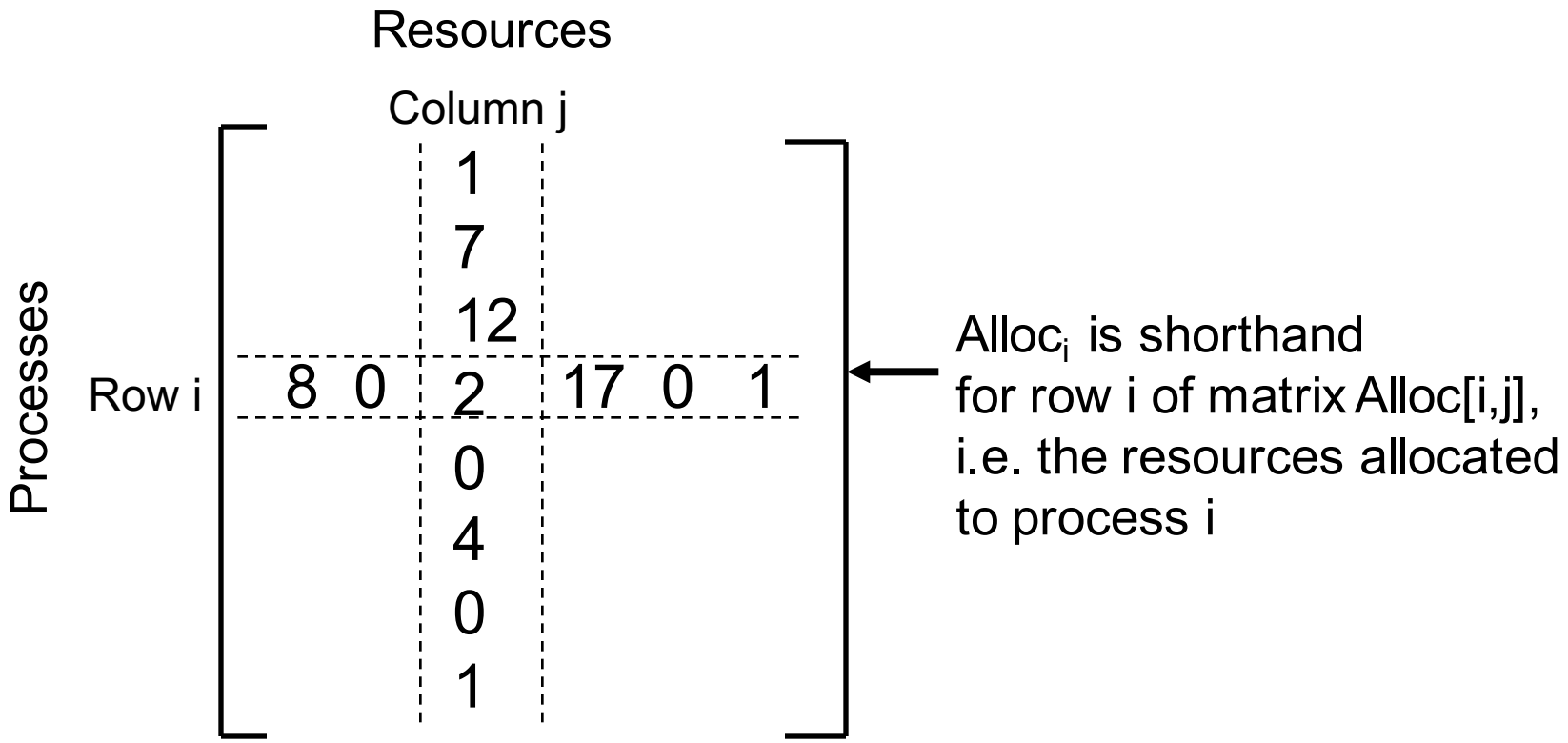
- Disadvantages:
  - Must know in advance the maximum resource usage for each process
  - Overly conservative in avoiding deadlock in the worst case

Banker's Algorithm Definitions

- 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 R_j 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 R_j
- Allocated resources in a matrix **Alloc[i,j]**
  - Alloc[i,j] = k means that process i is currently allocated k instances of resource R_j
- Needed resources in a matrix **Need[i,j]** = **Max[i,j]** - **Alloc[i,j]**

Banker's Algorithm Definitions

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



Banker's Algorithm Definitions

- Some terminology:
  - let X and Y be two vectors. Then we say X ≤ Y if and only if X[i] ≤ 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 ≤ V1, but V3 ≰ V1, i.e. V3 is not less than or equal to V1

Banker's Algorithm

- Is the system in a safe state? Find a safe sequence:
  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
     - $Need_i \leq Work$

     If no such i exists, go to step 4.
  3. Work = Work + $Alloc_i$

     Finish[i] = true

     Go to step 2.
  4. If Finish[i]==true for all i, then the system is in a safe state

Intuition: if all prior processes give up all their resources, is there enough to meet the max needs of next process in the sequence?

Banker's Algorithm Example

- Example 3:
  - 3 resources (A,B,C) with total instances available (10,5,7)
  - 5 processes
  - At time t0, the allocated resources **Alloc[i,j]**, max needs **Max[i,j]**, and available resources **Avail[j]**, are:

Alloc[i,j]

	A	B	C
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2

Max[i,j]

A	B	C
7	5	3
3	2	2
9	0	2
2	2	2
4	3	3

Avail[j]

A	B	C
3	3	2

Need[i,j]

A	B	C
7	4	3
1	2	2
6	0	0
0	1	1
4	3	1

Banker's Algorithm Example

- Is the system in a safe state? Is there a safe sequence?
  - Yes, the claim is that <*P1, P3, P4, P2, P0*> is a safe sequence
  - Find a process *i* such that $Need_i \leq$ Work (=Avail)
    - Which processes satisfy this condition?

Alloc[i,j]

	A	B	C
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2

Max[i,j]

A	B	C
7	5	3
3	2	2
9	0	2
2	2	2
4	3	3

Avail[j]

A	B	C
3	3	2

Need[i,j]

A	B	C
7	4	3
1	2	2
6	0	0
0	1	1
4	3	1

Banker's Algorithm Example

- claim is that *<P1, P3, P4, P2, P0>* is safe
  - P1 and P3 have $Need_i \leq Work$ (=Avail)
  - Choose P1: $Need_1$ = <1,2,2> ≤ Available = <3,3,2>
  - P1 takes all that it needs, then releases all held resources, which equals its maximum demands
  - Work = Work + $Alloc_1$ = <3,3,2> + <2,0,0> = <5,3,2>

Alloc[i,j]

	A	B	C
P0	0	1	0
P1	**2**	**0**	**0**
P2	3	0	2
P3	2	1	1
P4	0	0	2

Max[i,j]

A	B	C
7	5	3
3	2	2
9	0	2
2	2	2
4	3	3

Avail[j]

A	B	C
3	3	2

Need[i,j]

A	B	C
7	4	3
1	**2**	**2**
6	0	0
0	1	1
4	3	1

Banker's Algorithm Example

- claim is that *<P1, P3, P4, P2, P0>* is safe
  - Find process i with $Need_i \leq Work = <5,3,2>$
  - Choose P3: $Need_3 = <0,1,1> \leq Available = <5,3,2>$
  - P3 takes its max, releases all
  - $Work = Work + Alloc_3 = <5,3,2> + <2,1,1> = <7,4,3>$

Alloc[i,j]

	A	B	C
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	**2**	**1**	**1**
P4	0	0	2

Max[i,j]

A	B	C
7	5	3
3	2	2
9	0	2
2	2	2
4	3	3

Avail[j]

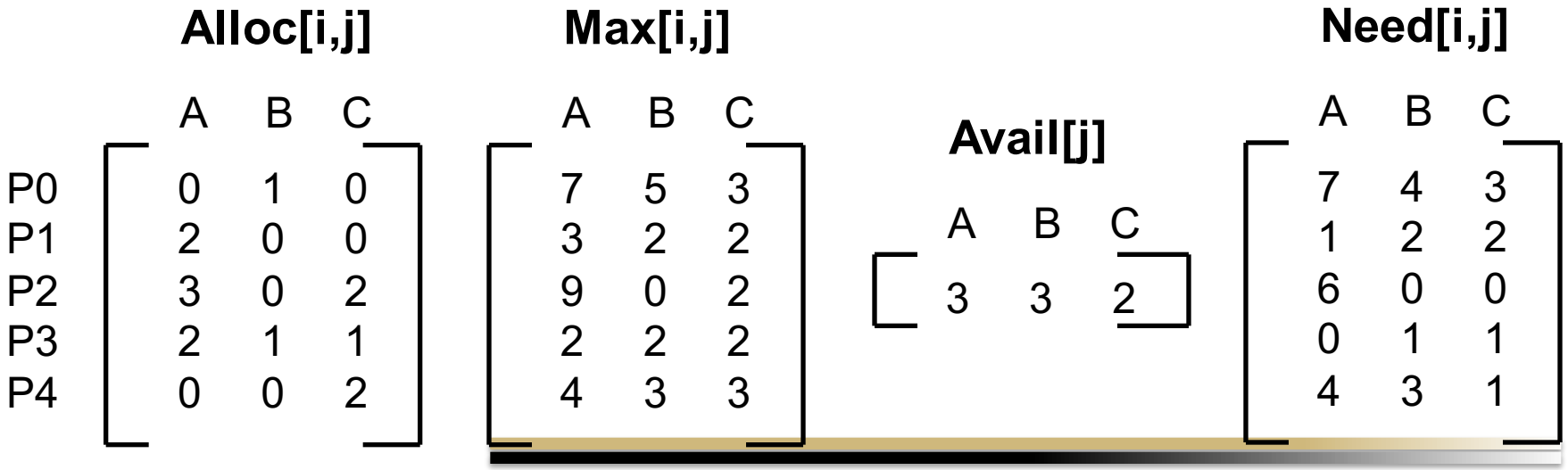
A	B	C
3	3	2

Need[i,j]

A	B	C
7	4	3
1	2	2
6	0	0
0	**1**	**1**
4	3	1

Banker's Algorithm Example

- claim is that <*P1, P3, P4, P2, P0*> is safe
  - Find process i with $Need_i \leq Work = <7,4,3>$
  - P0, P2 and P4 all satisfy this condition, so we can take them in any order.
  - Therefore, <P1, P3, P4, P2, P0> is safe!



Banker's Algorithm

- Is the system in a safe state? Find a safe sequence:
  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
     - $Need_i \leq Work$

     If no such i exists, go to step 4.
  3. Work = Work + $Alloc_i$

     Finish[i] = true

     Go to step 2.
  4. If Finish[i]==true for all i, then the system is in a safe state

Deadlock Avoidance

- Questions about Banker's Algorithm:
  - If it finds a safe sequence, are there other safe sequences, i.e. is the safe sequence that is found unique?
    - The safe sequence is not necessarily unique. There may be others.
    - Simple case: if P1 and P2 both can have their max needs satisfied by available resources, then <P1, P2> and <P2,P1> are both safe sequence
    - In Banker's example 3, another safe sequence was <P3,P1,P0,P2,P4>

Deadlock Avoidance

- Complexity of Banker's Algorithm is order $O(m*n^2)$
  - If there are n processes, then in the worst case the processes are ordered such that each iteration of the banker's algorithm must evaluate all remaining processes before the last one satisfies $Need_i \leq Work$
    - the 1^{st} time, n processes have to compare if their $Need_i \leq Work$
    - the 2^{nd} time, n-1 processes have to be compared
    - Thus, in the worst case, there are $n + (n-1) + (n-2) + \ldots + 2 + 1 = n(n+1)/2$ comparisons, which is proportional to n^2 complexity
    - Each vector comparison requires m individual comparisons, since there are m resource types, so total complexity is $O(m*n^2)$

Deadlock Avoidance

- Banker's Algorithm determines whether the system is in a safe state
- Suppose we have a *new* request, and the current system is in a safe state
  - Should we grant the new request?
  - Yes, if it leaves the system in a safe state.

Resource-Request Algorithm

Let $Request_i$ be a new request vector for resources for process P_i

1. If $Request_i \leq Need_i$, go to step 2. Else, the new request exceeds the maximum claim. Exit.
2. If $Request_i \leq Available$, go to step 3. Else, process P_i must wait because there aren't enough available resources
3. Temporarily modify Available[j], Need[i,j], and Alloc[i,j]
   - Avail -= $Request_i$
   - $Alloc_i$ += $Request_i$
   - $Need_i$ -= $Request_i$

   Execute the Banker's algorithm. If system is in a safe state, grant the request and update Avail, Need, and Alloc

Deadlock Avoidance

- Recall Example 3:
  - 3 resources (A,B,C) with total instances available (10,5,7)
  - 5 processes
  - At time t0, the allocated resources Alloc[i,j], Max needs Max[i,j], and Available resources Avail[j], are:

Alloc[i,j]

	A	B	C
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2

Max[i,j]

	A	B	C
P0	7	5	3
P1	3	2	2
P2	9	0	2
P3	2	2	2
P4	4	3	3

Avail[j]

A	B	C
3	3	2

Need[i,j]

	A	B	C
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

Banker's Algorithm found that this was in a safe state, with safe sequence <P1, P3, P4, P2, P0>

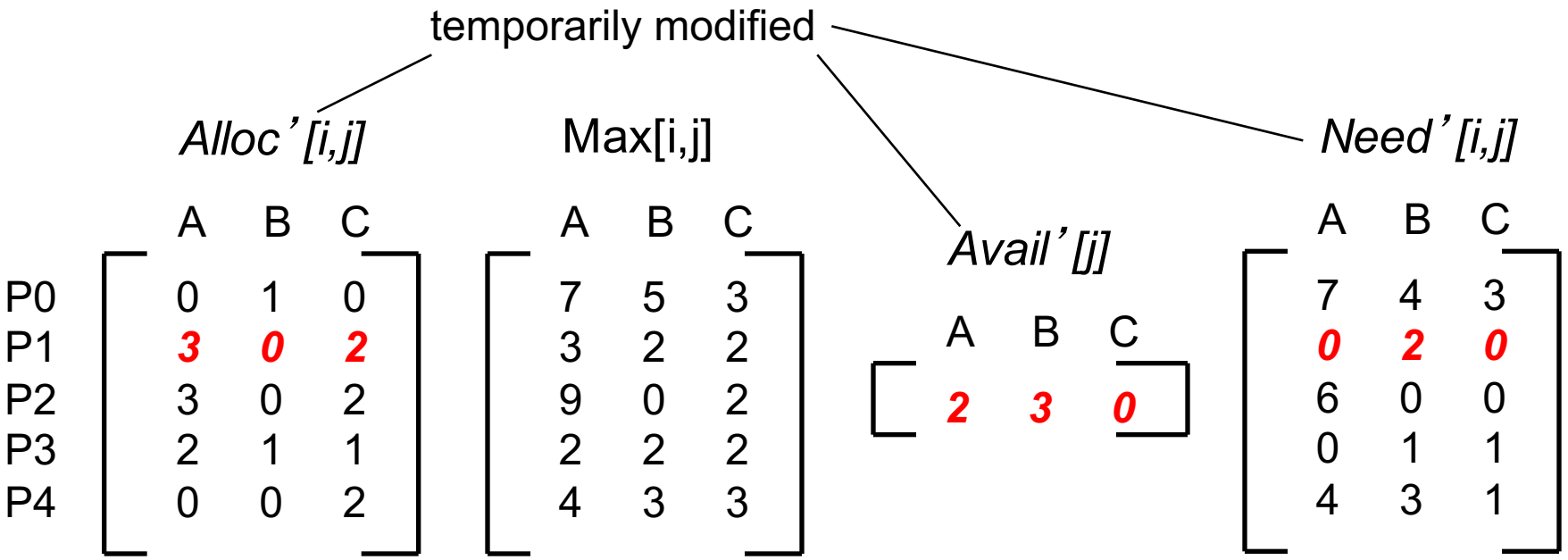
where Need[i,j] is computed given Alloc[i,j] and Max[i,j]

Deadlock Avoidance

- Suppose we have a new request from process P1, $Request_1$ = <1,0,2>.
  - Can the system satisfy this request without becoming unsafe? Execute the Resource-Request Algorithm.

1. $Request_1 \leq Need_1$ = <1,2,2> Good!
2. $Request_1 \leq Available$ = <3,3,2> Good!
3. Temporarily recompute Avail, Need, and Alloc
   - see next slide

Deadlock Avoidance



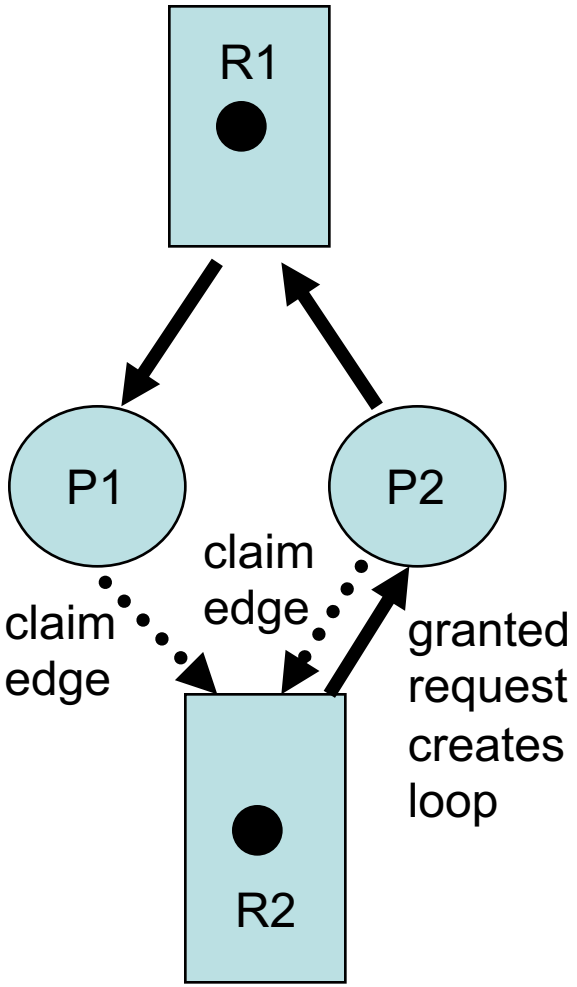
- Execute Banker's Algorithm on this revised state. We find that the sequence <P1,P3,P4,P0,P2> is safe.
- Thus we can grant the Request$_1$ immediately, and permanently update the matrices and vectors.

Deadlock Avoidance

- Given this new state, suppose we have a 2nd request:
  - from P4, such that $Request_4$ = <3,3,0>. Show that this cannot be granted because
    - $Request_4 \not\leq Available$ = <2,3,0>.
    - That is, there are not enough available resources.
  - from P0, such that $Request_0$ = <0,2,0>. Show that this cannot be granted because no safe sequence can be found.

Deadlock Avoidance

- Using a Resource Allocation Graph for deadlock avoidance
  - each process identifies possible future claims, drawn as claim edges (dotted lines)
  - If converting a claim edge to a request edge creates a loop, then don't grant request
    - limited applicability - only valid when there is 1 instance of each resource
  - example: Granting P2's request for R2 creates a loop, so don't grant it



Deadlock *Detection*

- Banker's Algorithm is used for *avoidance*
  - Knowing the maximum needs, the allocated resources and available resources, determine if the one *proposed* request will tip the system into an unsafe state – if so, then employ *avoidance before the fact*
- In deadlock *detection*, don't assume maximum needs are known in advance
  - Take the system as is: we know the allocated resources, available resources, and requests *that have been made* – using this information, determine if a deadlock exists => *detection after the fact*
  - Detection solution is similar to Banker's algorithm: find a sequence of releases by processes such that all requested needs can be met

Deadlock Detection

- **Deadlock Detection Algorithm**:

  Assume we have a matrix **Alloc[I,j]** of m resources allocated to n processes. Assume also we have a matrix **Request[i,j]** of m resources already requested by the m processes. (no more Max or Need)

1. Let Work and Finish be vectors length m and n respectively. Initialize Work = Available. For i=0,…n-1, if $Alloc_i$==0 then Finish[i]=true, else Finish[i]=false
2. Find a process i such that both
   - Finish[i]==false, and
   - $Request_i \leq Work$

   If no such i exists, go to step 4.
3. Work = Work + $Alloc_i$

   Finish[i] = true

   Go to step 2.
4. If Finish[i]==false for some i, then the system is in a deadlocked state. Moreover, if Finish[i]==false, then process P_i is deadlocked

No hold and wait so no deadlock

Intuition: if all prior processes release all their resources, is there enough to meet the requested needs of remaining processes?

Deadlock Detection

- Example:
  - 3 resources (A,B,C) with total instances equal to (7,2,6)
  - 5 processes
  - At time t0, the allocated resources Alloc[i,j], requested resources Request[i,j], and Available resources Avail[j], are:

Alloc[i,j]

	A	B	C
P0	0	1	0
P1	2	0	0
P2	3	0	3
P3	2	1	1
P4	0	0	2

Request[i,j]

	A	B	C
P0	0	0	0
P1	2	0	2
P2	0	0	0
P3	1	0	0
P4	0	0	2

Avail[j]

A	B	C
0	0	0

Is this system deadlocked? No, the detection algorithm finds a sequence <P0,P2,P3,P1,P4> that releases all resources such that Finish[i]==true for all i.

Deadlock Detection

- Example (cont)
  - From previous example, suppose now P2 makes one additional request for an instance of type C. The Request matrix would change the line for P2 to $Request_2$ = <0 0 1>.
    - The detection algorithm finds that process P0 can release its resources, but the number of available resources is not enough to prevent deadlock for processes P1, P2, P3, and P4.

Deadlock Detection

- When/how often should the detection algorithm run?
  - Depends on how often deadlock is likely to occur
  - Depends on how quickly deadlock grows after it occurs, i.e. how many processes get pulled into deadlock and on what time scale
  - Could check at each resource request – this is costly
  - Could check periodically – but what is a good time interval?
  - Could check if CPU utilization suddenly drops – this might be an indication that there's deadlock, and processes are no longer executing, but what's a good threshold?
  - Could check if resource utilization exceeds some threshold, but what's a good threshold?

Deadlock Recovery

- After OS has detected which processes are deadlocked, then OS can:
  - Terminate all processes - draconian
  - Terminate one process at a time until the deadlock cycle is eliminated
    - Check if there is still deadlock after each process is terminated. If not, then stop.
  - Preempt some processes – temporarily take away a resource from current owner and give it to another process but don't terminate process
    - e.g. give access to a laser printer – this is risky if you're in middle or printing a documents
  - Rollback some processes to a checkpoint – assuming that processes have saved their state at some checkpoint

Supplementary Slides

Deadlock Avoidance

- Questions about Banker's Algorithm (cont):
  - If a safe sequence is not found, could we reorder our search to find another safe sequence? do we have to search exhaustively all possible sequences and/or backtrack our search?
    - Answer: no amount of reordering will find a safe sequence if a safe sequence couldn't be found the first time, so we don't have to search exhaustively or backtrack. Reasoning:
    - This is because each step of the algorithm only releases more resources. At each step, B.A. expands the number of processes whose needs can be met by what has been cumulatively released by the previous processes.
    - example: suppose there are 3 processes. We run B.A. and it selects P0 first whose max needs can be met (acquires and releases its max resources, increasing the number of available resources). Now suppose neither P1 nor P2 can satisfy their max needs. So we have not found a safe sequence yet. If we run the banker's algorithm again, and try to select P1 or P2 instead of P0, we see that neither P1 nor P2 could possibly have their needs met since there are even fewer resources than before (P0 has not released its currently held resources). Thus if B.A. can't find a safe sequence the first time, then no amount of reordering will find another safe sequence.