Chapter 8: Deadlocks

CS 3423 Operating Systems
Fall 2019
National Tsing Hua University

Overview

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

The Deadlock Problem

- A set of blocked processes each holding some resources and waiting to acquire a resource held by another process in the set
- Ex1: 2 processes and 2 tape drivers
  - Each process holds a tape drive
  - Each process requests another tape drive
- Ex2: 2 processes, and semaphores A & B
  - P1 (hold B, wait A): wait(A), signal(B)
  - P2 (hold A, wait B): wait(B) , signal(A)

Deadlock with Mutex Locks

- Deadlocks can occur via
  - system call, locking, ...
- textbook example
  - process1:
    lock(mutex1)
    lock(mutex2)
    work()
    unlock(mutex2)
    unlock(mutex1)
  - process2:
    lock(mutex2)
    lock(mutex1)
    work()
    unlock(mutex1)
    unlock(mutex2)

Deadlock vs Livelock

- Deadlock:
  - all processes with circular dependency are blocked, cannot make progress
- Livelock:
  - not blocked, but busy with repeatedly failing
  - example: nonblocking version of mutex acquire, maybe bad luck or bad phasing of events
  - example: Ethernet collision
  - solution: randomize backoff

Four Necessary Conditions for Deadlock

1. Mutual exclusion:
   - only one process at a time can use a resource
2. Hold and wait:
   - a process holding at least one resource is waiting to acquire additional resources held by other processes
3. No preemption **of resource**: (!= process/thread preemption!!)
   - a resource can be released only voluntarily by the process holding it, after that process has completed its task
4. Circular wait:
   - there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_i is waiting for a resource that is held by P_{i+1}, for $0 \leq i < n$, *and* P_n is waiting for a resource that is held by P_0.

All four conditions hold simultaneously => deadlock

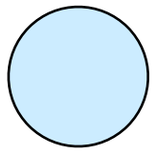
System Model

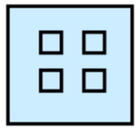
- Resource types R_1, R_2, ... , R_m
  - CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances
  - e.g., a processor with two cores
- Sequence of steps in utilizing a resource:
  1. request
  2. use
  3. release

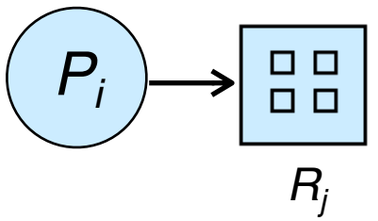
Resource-Allocation Graph $G(V, E)$

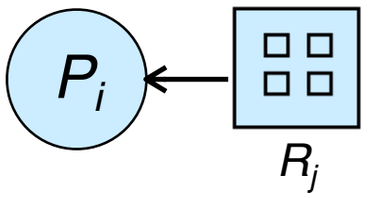
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, (set of processes)
  - $R = \{R_1, R_2, \ldots, R_m\}$, (set of resource types) (each type may have multiple instances)
- request edge
  - directed edge $P_i \rightarrow R_j$
- assignment edge
  - directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

- Process 

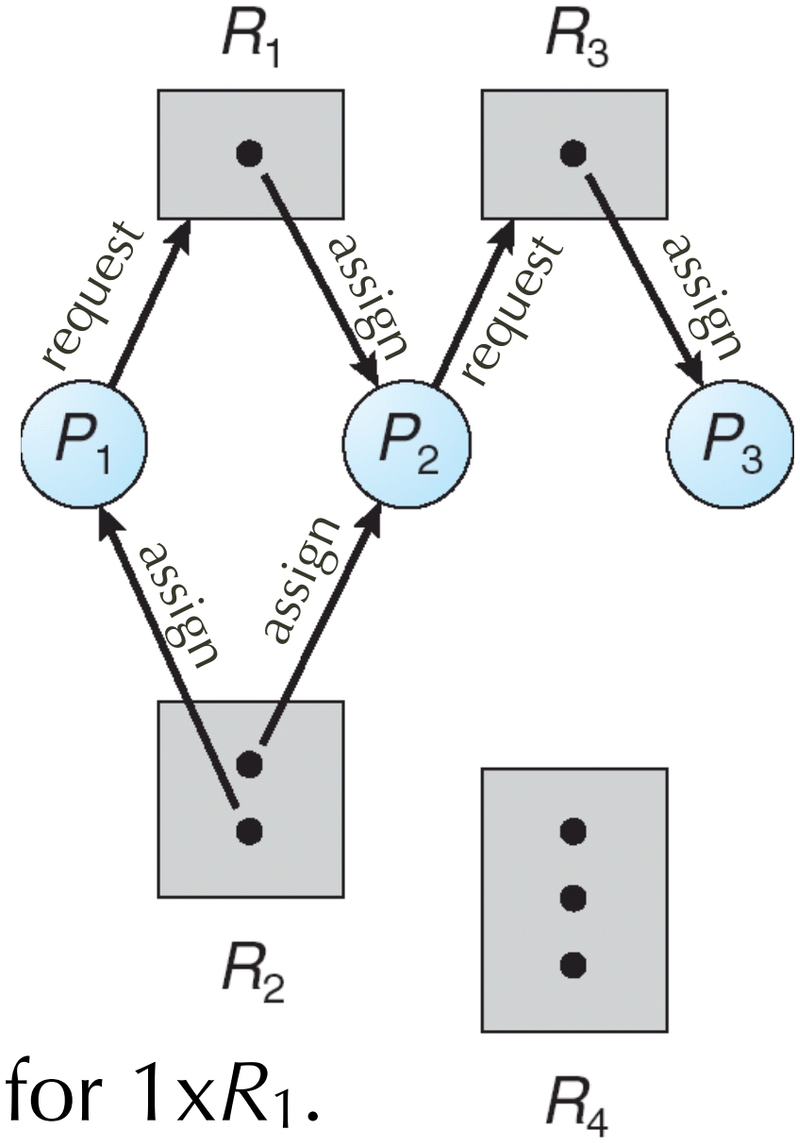
- Resource Type with 4 instances 

- P_i requests instance of R_j 

- P_i is holding an instance of R_j 

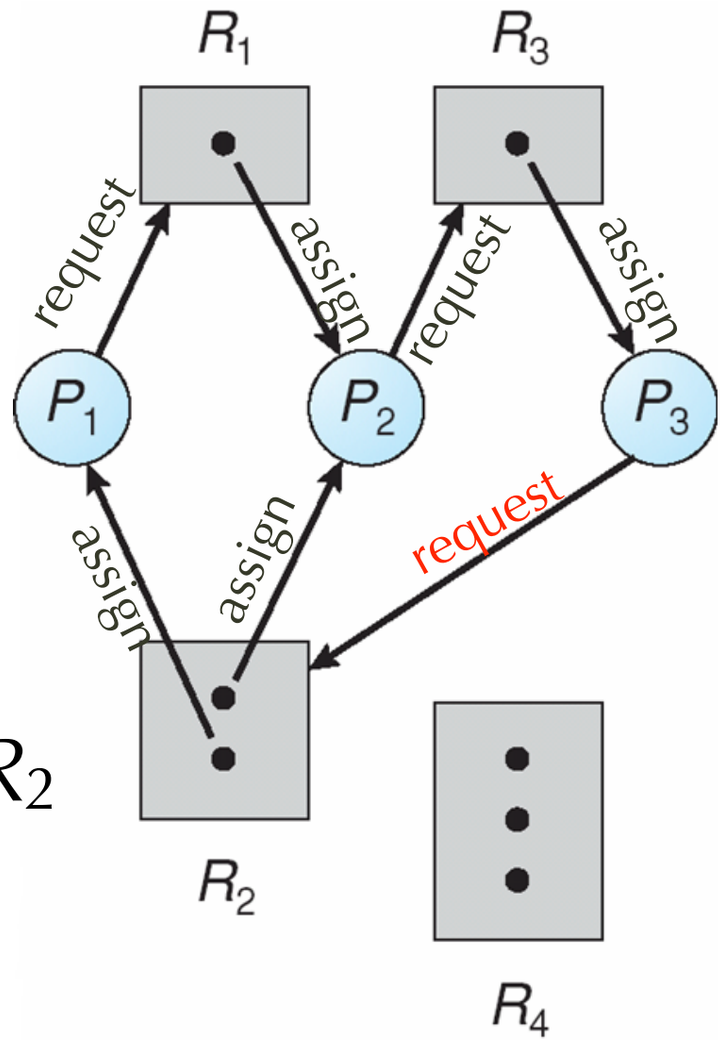
Example of a Resource Allocation Graph

- 3 processes P_1..P_3
- 4 resources R_1..R_4
  - R_1: 1 instance, R_2: 2 instances,
    R_3: 1 instance, R_4: 4 instances
- Request edge
  - P_1 requests R_1, P_2 requests R_3
- assignment edges:
  - P_1 holds 1 instance of R_2,
    P_2 holds 1 instance of R_2
- => P_1 is holding 1xR_2 while waiting for 1xR_1.



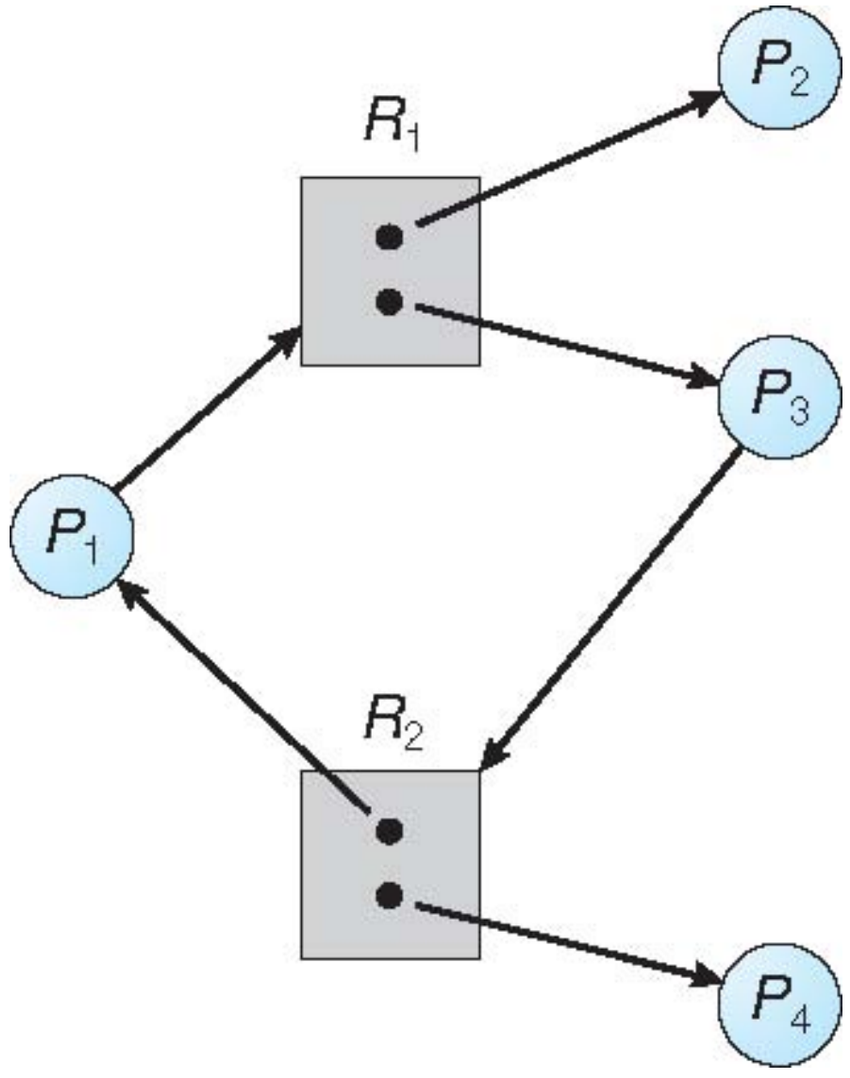
Resource Allocation Graph with a Deadlock

- if a graph contains a cycle,
  - a deadlock **<u>may</u>** exist
- In this ex., deadlock exists
  - P_1 waiting for P_2 to release R_1
  - P_2 waiting for P_3 to release R_3
  - P_3 waiting for P_1 or P_2 to release R_2
  - => Cycle



Graph with a Cycle but no Deadlock

- Cycle
  - P_1 waiting for P_2 or P_3 to release R_1
  - P_3 waiting for P_1 or P_4 to release R_2
- No deadlock because
  - P_2 and P_4 wait for no-one => no deadlock between P_1 and P_3.



Basic Facts

- If graph contains no cycles ⇒ no deadlock
- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

Methods for Handling Deadlocks

- Deadlock **prevention**
  - ensures necessary conditions cannot hold
- Deadlock **avoidance**
  - dynamically examines state of resource allocation before assignment
- Deadlock **detection**
  - Allow the system to enter a deadlock state, detect it, and figure out what to do
- Deadlock **recovery**
  - After detection, ways to recover from deadlock state
- Pretend that deadlocks never occur in the system
  - used by most operating systems, including UNIX

Deadlock Prevention

- Limit the ways that requests can be made
  - some similar to race condition but higher level
- Break a necessary condition for deadlock
  - Mutual Exclusion
  - Hold and Wait
  - No preemption
  - Circular Wait

Deadlock Prevention
1. deny Mutual Exclusion

- Allow simultaneous access
  - e.g., sharing read-only files, no need for mutex
  - Use counting semaphores
- However, some resources require mutex!
  - e.g., some printers-- at most one process can print
  - Note: book calls these "non-sharable" but strictly speaking these are "not simultaneously accessible"

Deadlock Prevention
2. eliminate Hold and Wait

- "Atomic" multi-resource allocation.
  Two options:

1. Allocate **<u>all</u>** requested resources **before** process starts
   - Impractical for most applications -- too "static"
2. Process must hold no resource before request
   - if can't get all resources => need to release ALL currently held resources

✳ Disadvantages
   - Low resource utilization;
   - starvation possible

Deadlock Prevention

3. Allow Resource Preemption

- Temporarily take away resource already allocated to a process P_r
  - when P_r wants to requests more resources but must wait
  - => temporarily release **all** resources acquired by P_r, so other processes get a chance to use those resources first
- Somewhat like context switch & scheduling
  - OS needs to track and re-acquire resources that P_r acquired but got taken away
  - Move P_r to ready queue when all its resources can be acquired.

Deadlock Prevention

3. Allow Resource Preemption

- Applicability
  - Resources whose states can be easily saved and restored
  - e.g. CPU registers & memory, database transactions
- Not generally applicable to..
  - mutex locks, semaphores,
  - printers & tape drives

Deadlock Prevention
4. Break Circular-Wait

- Impose a **<u>total ordering</u>** of all resource types
  - Require that each process requests resources in an increasing order of enumeration
  - prevents a circular order of resource acquisition
- Example:
  - F(tape drive) = 1, F(disk drive) = 5, F(printer) = 12
    A process must request tape and disk drive before printer
- proof: counter-example does not exist
  - P_0 (R_0) R_1, P_1 (R_1) R_2, ..., P_N(R_N) -> R_0
- Conflict:
  - $R_0 < R_1 < R_2 < ... < R_N < R_0$ where P_N holds R_N and waits R_0

Deadlock Example

```c
/* 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);
}
```

```c
/* 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);
}
```

Deadlock Example with Lock Ordering

```
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);
}
```

Transactions 1 and 2 execute concurrently.

Transaction 1 transfers $25 from account A to account B, and

Transaction 2 transfers $50 from account B to account A

Deadlock Avoidance

- Simplest and most useful model
  - Needs additional *a priori* knowledge about system
- Requires that each process declare
  - the maximum number of resources of each type that it may need
- Deadlock-avoidance algorithms
  - dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation state is defined by
  - the number of available and allocated resources,
  - and the maximum demands of the processes

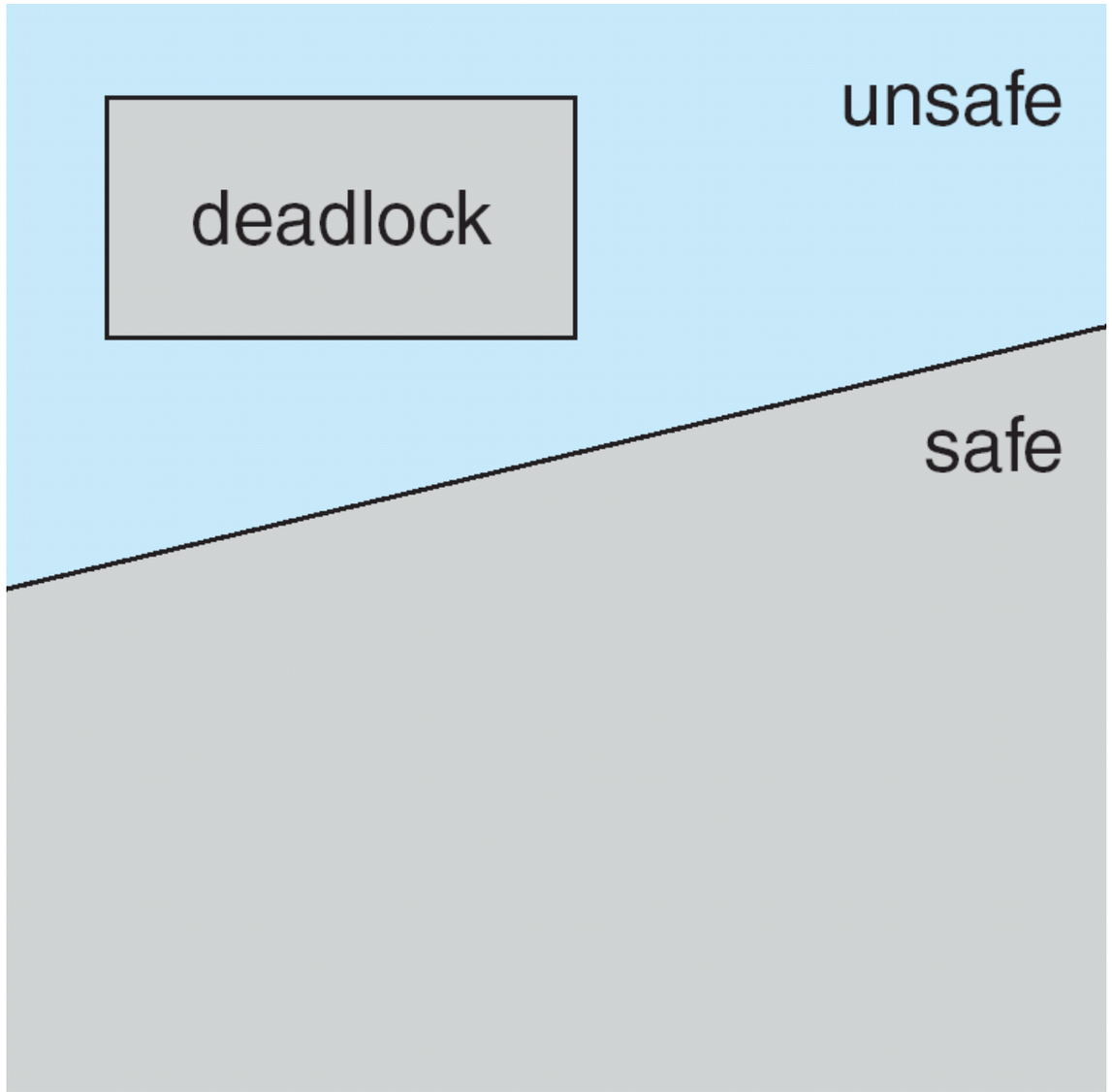
Safe State

- System is in safe state if
  - there exists sequence <P_1, P_2, …, P_n> of processes in systems
  - P_i requests resources = curr. available + those held by P_j, $j < i$
- OS must decide if allocation leaves system in *safe state*
  - If P_i resource needs are not immediately available, => P_i waits until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources

Basic Facts

- If a system is in safe state
  - ⇒ no deadlocks
- If a system is in unsafe state
  - ⇒ possibility of deadlock
- Avoidance
  - ⇒ ensure that a system will never enter an unsafe state.

Safe, Unsafe, Deadlock State



Deadlock-Avoidance Algorithms

- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the banker's algorithm

- Both require a priori knowledge about resource needs (for "claims")

Resource-Allocation Graph Scheme

- Claim edge $P_i \rightarrow R_j$ (dashed line)
  - process P_j <u>may request</u> resource R_j
  - Resources must be claimed *a priori* in the system
- Request edge $P_i \rightarrow R_j$ (solid line)
  - Converted from a Claim edge when a process requests a resource
  - process P_j <u>waits for</u> resource R_j
- Assignment edge $R_j \rightarrow P_i$ (solid line)
  - Converted from a request edge when the resource is allocated to the process -- by flipping the direction of arrow!
  - Resource R_j is assigned to process P_j
- Assignment edge reconverts to a *claim edge* (flip direction back)

Resource-Allocation Graph

- for single-instance resources
- Claim edge

  ⇢ P_i may request for R_j

  convertible to request edge

  as the process runs

  → Request edge

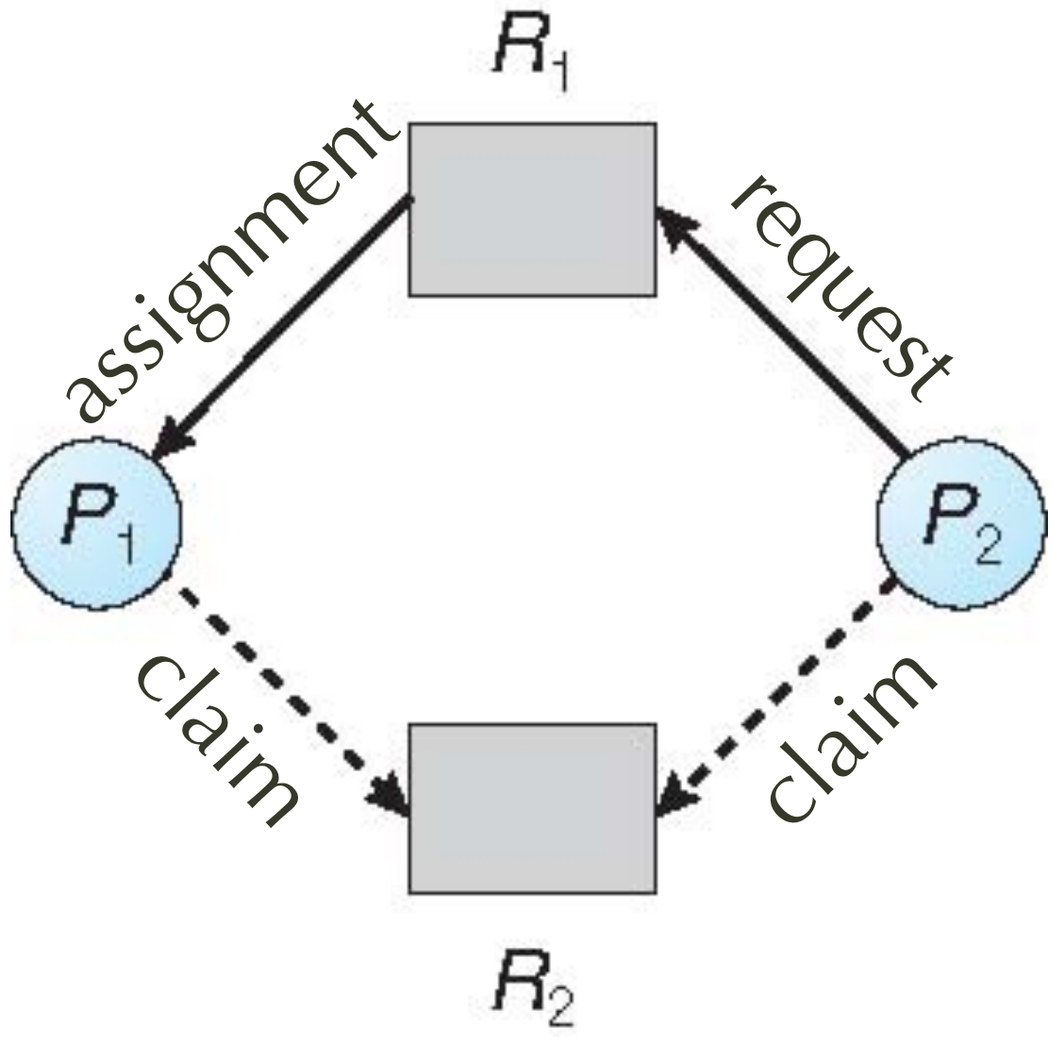
  P_i waits for R_j

  ← Assignment edge

  R_j assigned to P_i

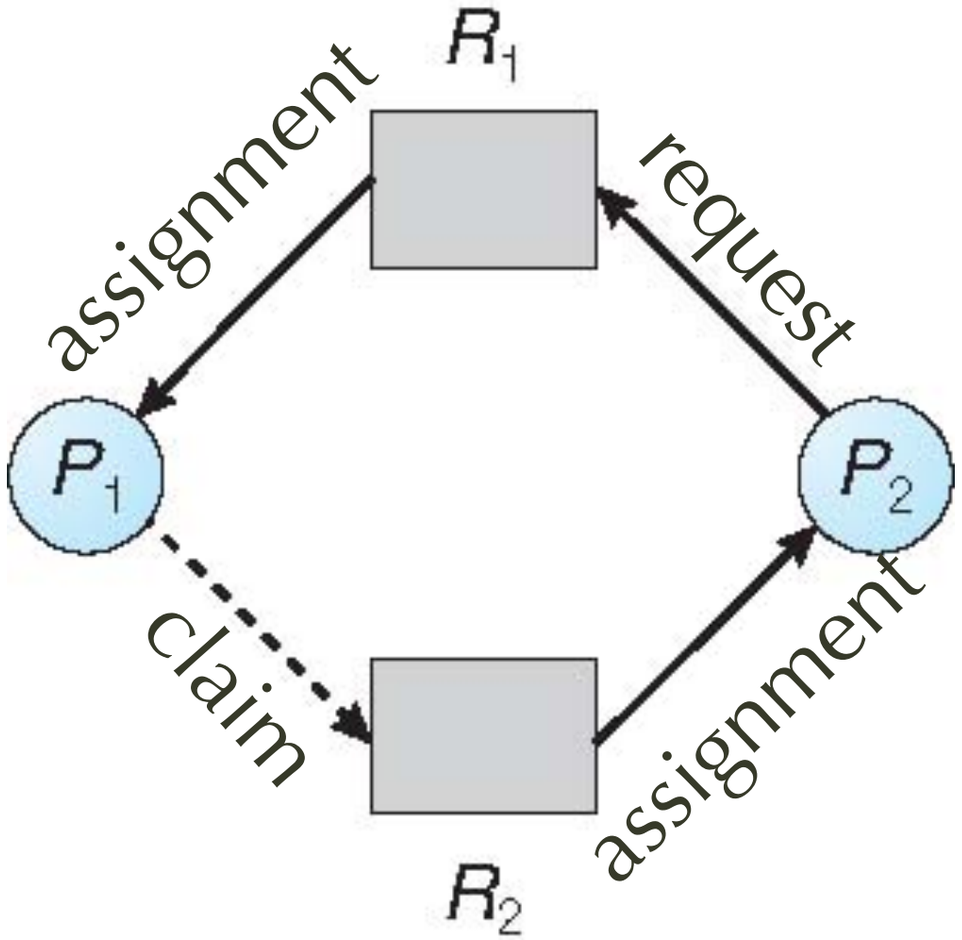
  convertible to claim edge

  as resource is released by process



Unsafe State In Resource-Allocation Graph

- Resources must be claimed *a priori*
- Request granted only if no cycle created
- Check safety by cycle detection, $O(n^2)$
- Example: R_2 can't be allocated to P_2



Resource-Allocation Graph Algorithm

- For process P_i to request a resource R_j
- The request can be granted only if
  - converting request edge to assignment edge does not result in a cycle in the RAG

Multi-instance resources: Safe State with Safe Sequence

- 12 tape drives
- assume at time t_0

	Max need	Current holding
P_0	10	5
P_1	4	2
P_2	9	2

- safe sequence: $<P_1, P_0, P_2>$
  - P_1 needs 2 more, 3 available

	Max	hold	avail
P_0	10	5	3 - 2 = 1
P_1	4	2 + 2	
P_2	9	2	

  - P_1 finishes, free up all disks

	Max	hold	avail
P_0	10	5	1+4 = 5
P_1	4	0	
P_2	9	2	

Safe State with Safe Sequence (cont'd)

- 12 tape drives, safe sequence: <P_1, P_0, P_2>
  - P_0 needs 5 more, 5 available

	Max	hold	avail
P_0	10	5 + 5	5 - 5 = 0
P_1	4	0	
P_2	9	2	

  - P_0 finishes, free up all disks

	Max	hold	avail
P_0	10	10-10	0+10 = 10
P_1	4	0	
P_2	9	2	

  - P_2 needs (9-2=)7 more, 10 avail.

	Max	hold	avail
P_0	10	0	10-7 = 3
P_1	4	0	
P_2	9	2+7	

Unsafe State without Safe Sequence

- 12 tape drives
- assume at time t_1
- If P_2 is allocated 1 more drive

	Max	Curr. holding	Available
P_0	10	5	2 instead of 3
P_1	4	2	
P_2	9	3 instead of 2	

- then no safe sequence exists because remaining 1 drive cannot satisfy any process (all need more than 1 drive)

  => System enters unsafe state

	Max	hold	avail
P_0	10	5	2-1 =1
P_1	4	2	
P_2	9	3+1=4	

Banker's Algorithm

- For multiple instances of each resource type
  - Each process must *a priori* claim maximum use
- Banker's algorithm
  - use a general "safety algorithm" to predetermine if any safe sequence exists after allocation
  - allocate only if a safety sequence exists
- Safety Algorithm
  - find a process that can be satisfied by freeing resources
  - free that process's resources, repeat until all process are satisfied

Data Structures for the Banker's Algorithm

- n = # processes, m = # resources
  - **int** Available[m]; // [j] =#instances of resource type R_j that exist
  - **int** Work[m]; // [j] = #instances of resource type R_j not yet allocated
- **int** Max[n][m]; // P_i may request up to Max[i,j] instances of resource type R_j
- **int** Allocation[n][m]; // P_i is currently allocated Allocation[i,j] instances of R_j
- **int** Need[n][m]; // P_i may need up to Need[i,j] more instances of R_j to complete its task
- Need[i,j] = Max[i,j] – Allocation[i,j]

Safety Algorithm

1. **int** *Work*[*m*] = *Available*[:];
   bool *Finish*[*n*] = [false]*n;*

2. Find an i such that both:

   (a) *Finish*[i] == false

   (b) $Need_i$[:] ≤ *Work*[:] /* means resources needed by P_i can all be satisfied, including trivially - i.e., zero need */

   If no such i exists, go to step 4

3. /* step 2 just found process *Pi* that is safe, so */
   Finish[i] = true /* run *Pi* and get its allocation back! */
   Work[:] = *Work*[:] + $Allocation_i$[:] /* give rsrc back to pool */
   go to step 2

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

Example of Safety test in Banker's Algorithm

- 5 processes $P_0..P_4$
- 3 resource types:
  - A (10 instances),
  - B (5 instances), and
  - C (7 instances)
- Snapshot at time T_0

	Max			Alloc			Need = Max - Alloc			Available		
P \ R	A	B	C	A	B	C	A	B	C	A	B	C
P_0	7	5	3	0	1	0	7	4	3			
P_1	3	2	2	2	0	0	1	2	2			
P_2	9	0	2	3	0	2	6	0	0	10	5	7
P_3	2	2	2	2	1	1	0	1	1	-) **7**	**2**	**5**
P_4	4	3	3	0	0	2	4	3	1	= 3	3	2
Total				**7**	**2**	**5**						

Steps of Safety Algorithm (1/8)

- Step 1: Initialize
  - $m = 3$, $n = 5$
  - Work[:m] = Available[:m] = [3, 3, 2]
  - Finish[:n] = [False, False, False, False, False]
- Step 2: for $i = 0$
  - Need[0] = [7, 4, 3]
  - Finish[0] is False, and Need[0] ≤ Work is False
    => P_0 must wait

[7, 4, 3] not ≤ [3, 3, 2]

Steps of Safety Algorithm (2/8)

- Step 2: for $i = 1$
  - Need[1] = [1, 2, 2]
  - Finish[1] is False, Need[1] $\leq$ Work => True

[1, 2, 2] ≤ [3, 3, 2]

  - => P_1 is added to safe sequence.
- Step 3

[3, 3, 2] [2, 0, 0]

  - Work = Work + Allocation[1] => [5, 3, 2]
  - Finish = [False, True, False, False, False]

Steps of Safety Algorithm (3/8)

- Step 2: for $i = 2$
  - Need[2] = [6, 0, 0]
    [6, 0, 0] not ≤ [5, 3, 2]
  - Finish[2] is false, Need[2] ≤ Work => False
  - So P_2 must wait.

Steps of Safety Algorithm (4/8)

- Step 2: for $i = 3$
  - Need[3] = [0, 1, 1]
  - Finish[3] = False and Need[3] ≤ Work => True
    => P_3 is added to sequence.

[0, 1, 1] ≤ [5, 3, 2]

- Step 3:
  - Work = Work + Allocation[3] => [7, 4, 3]
  - Finish = [False, True, False, True, False]

[5, 3, 2] [2, 1, 1]

Steps of Safety Algorithm (5/8)

- Step 2: for $i = 4$
  - Need[4] = [4, 3, 1]
  - Finish[4] = False and Need[4] ≤ Work => True
    => P_4 is added to sequence.

[4, 3, 1] ≤ [7, 4, 3]

- Step 3:
  - Work = Work + Allocation[4] => [7, 4, 5]
  - Finish = [False, True, False, True, True]

[7, 4, 3] [0, 0, 2]

Steps of Safety Algorithm (6/8)

- Step 2: for $i = 0$
  - Need[0] = [7, 4, 3]
  - $[7, 4, 3] \leq [7, 4, 5]$
  - Finish[0] = False and Need[0] ≤ Work => True
    => P_0 is added to sequence.
- Step 3:
  - $[7, 4, 5]$ $[0, 1, 0]$
  - Work = Work + Allocation[0] => [7, 5, 5]
  - Finish = [True, True, False, True, True]

Steps of Safety Algorithm (7/8)

- Step 2: for $i = 1$
  - Finish[1] = True
    => P_1 already in sequence, so skip.

Steps of Safety Algorithm (8/8)

- Step 2: for $i = 2$
  - Need[2] = [6, 0, 0]
  - Finish[2] = False and Need[2] ≤ Work => True
    => P_2 is added to sequence.

[6, 0, 0] ≤ [7, 5, 5]

- Step 3:
  - Work = Work + Allocation[2] => [10, 5, 7]
  - Finish = [True, True, True, True, True]
- Finished! sequence = [1, 3, 4, 0, 2]

[7, 5, 5] [3, 0, 2]

Resource-Request Algorithm for Process P_i

- $Request_i$ = request vector for process P_i.
- $Request_i[j] = k$ means process P_i wants k instances of resource R_j

- Algorithm Outcome of Resource-Request by *Pi*
  - safe ⇒ the resources are allocated to P_i
  - unsafe ⇒ P_i must wait, and the old resource-allocation state is restored

Resource-Request Algorithm for Process P_i

1. If $Request_i[:] > Need_i[:]$

   then raise error condition: process exceeded its max. claim

2. If $Request_i > Available$,
   then P_i must wait, since resources are not available

3. Pretend to allocate requested resources to P_i by modifying the state as follows:

   $Available[:] = Available[:] - Request_i[:];$

   $Allocation_i[:] = Allocation_i[:] + Request_i[:];$

   $Need_i[:] = Need_i[:] - Request_i[:];$

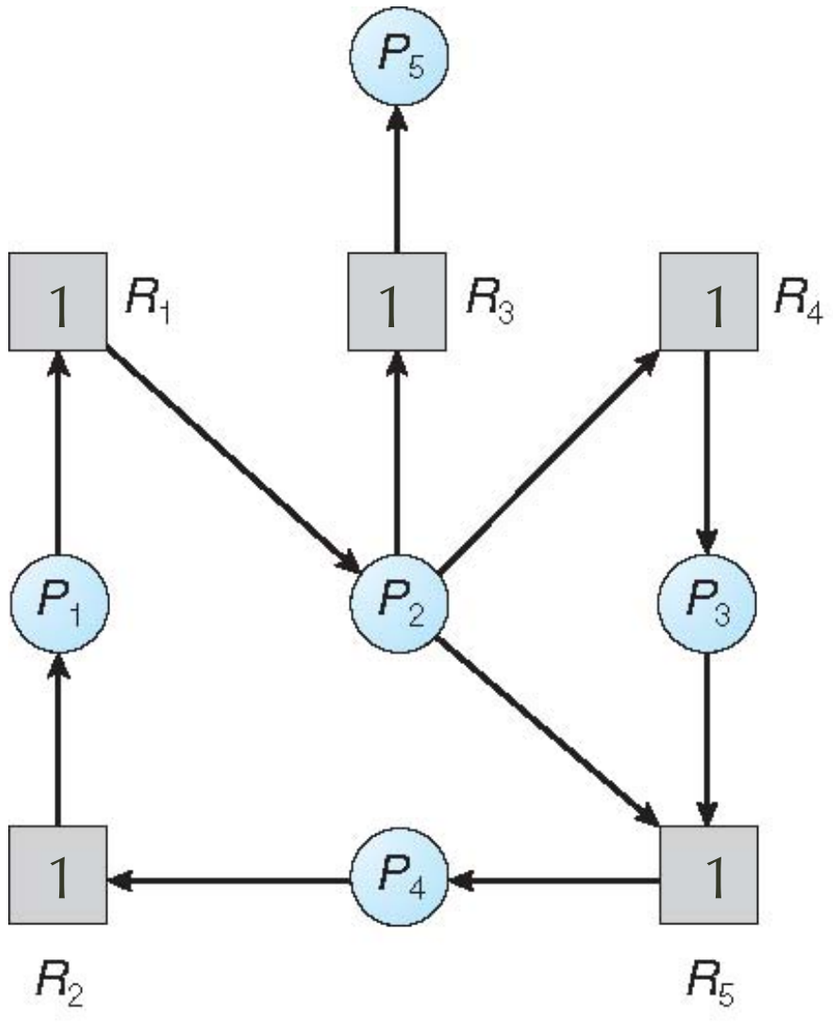
Deadlock Detection and Recovery

Single Instance of Each Resource Type

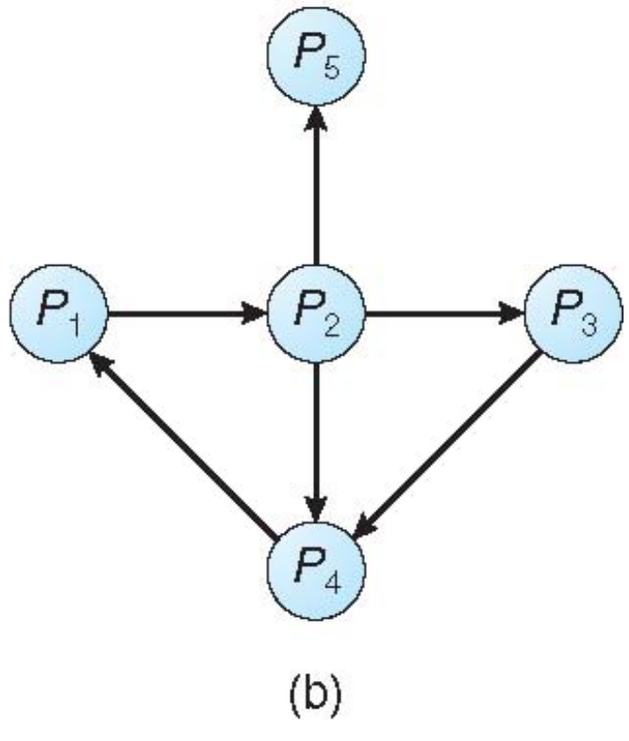
- Maintain **wait-for** graph
  - Vertices are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an algorithm that searches for a cycle in the graph.
  - If there is a cycle, there exists a deadlock
- Cycle detection in a graph $O(\boldsymbol{n^2})$ operations, where ***n*** is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph

Wait-for Graph is applicable only to single instance per resource type, but not multi-instance!!



Resource-Allocation Graph



Corresponding wait-for graph

Several Instances of a Resource Type

- Available:
  - A vector of length *m* indicates the number of available resources of each type
- Allocation:
  - An *n* x *m* matrix defines the number of resources of each type currently allocated to each process
- Request:
  - An *n* x *m* matrix indicates the current request of each process.
  - If Request[i][j] = *k*, then process P_i is requesting *k* more instances of resource type R_j.

Detection Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length ***m*** and ***n***, respectively.
   Initialize:

   (a) ***Work*****[:] = *****Available*****[:]**

   (b) for $i = 1,2, \ldots, n$,
   if $Allocation_i \neq 0$:
   $Finish[i]$ = False;
   else $Finish[i]$ = True

2. Find an index i such that both:

   (a) $Finish[i]$ == False

   (b) $Request_i \leq Work$

   If no such i exists, go to step 4

Detection Algorithm (Cont.)

3. ***Work*** **=** ***Work*** **+** $\boldsymbol{Allocation_i}$
   Finish$[i]$ **=** True
   go to step 2

4. If ***Finish****[i]* **==** False, for some i, $1 \leq i \leq$ ***n***,
   then the system is in deadlock state.
   Moreover, if ***Finish***$[i]$ **==** False, then $\boldsymbol{P}_i$ is
   deadlocked

Algorithm requires an order of O(m x n^2) operations to detect whether the system is in deadlocked state

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), B (2 instances), and C (6 instances)
- Snapshot at time T_0:

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

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in ***Finish***[i] == True for all i

Steps of Deadlock Detection

- Step 1: Initialize
  - $m = 3$, $n = 5$
  - Work[:m] = Available[:m] = [0, 0, 0]
  - Finish[:n] = [False, False, False, False, False]
- Step 2: for i = 0

  $[0, 0, 0] \leq [0, 0, 0]$
  - Finish[0] is False, and Request[0] ≤ Work is True
    => P_0 is added to sequence.
- Step 3:
  - Work += Allocation[0] => [0, 1, 0]
  - Finish = [True, False, False, False, False]

Steps of Deadlock Detection

- Step 2: for $i = 1$

  [2, 0, 2] not ≤ [0, 1, 0]

  - Finish[1] is False, but Request[1] ≤ Work is False
    => P_1 must wait

- Step 2: for $i = 2$

  [0, 0, 0] ≤ [0, 1, 0]

  - Finish[2] is False, and Request[2] ≤ Work is True
    => P_2 is added to sequence

- Step 3:

  [3, 0, 3]

  - Work += Allocation[2] => [3, 1, 3]
  - Finish = [True, False, True, False, False]

Steps of Deadlock Detection

- Step 2: for $i = 3$

  $[1, 0, 0] \leq [3, 1, 3]$

  - Finish[3] is False, and Request[3] ≤ Work is True
    => P_3 is added to sequence

- Step 3:

  $[2, 1, 1]$

  - Work += Allocation[3] => [5, 2, 4]
  - Finish = [True, False, True, True, False]

Steps of Deadlock Detection

- Step 2: for i = 4

  [0, 0, 2] ≤ [5, 2, 4]

  - Finish[4] is False, and Request[4] ≤ Work is True
    => P_4 is added to sequence

- Step 3:

  [0, 0, 2]

  - Work += Allocation[4] => [5, 2, 6]
  - Finish = [True, False, True, True, True]

Steps of Deadlock Detection

- Step 2: for $i = 0$
  - Finish[0] is True
    => *continue*
- Step 2: for $i = 1$

$$[2, 0, 2] \leq [5, 2, 6]$$

  - Finish[1] is False and Request[1] ≤ Work is True
- Step 3:

$$[2, 0, 0]$$

  - Work += Allocation[4] => [7, 2, 6]
  - Finish = [True, True, True, True, True]

Another Example of Detection Algorithm

- Suppose Request[2] is [0, 0, 1] instead.

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

- Step 2: for $i = 0$
  - Finish[0] is False, and Request[0] ≤ Work is True
    => P_0 is added to sequence.
- Step 3:
  - Work += Allocation[0] => [0, 1, 0]

Steps of Deadlock Detection

- Step 2: for $i = 1$

  [2, 0, 2] not ≤ [0, 1, 0]

  - Finish[1] is False, but Request[1] ≤ Work is False
    => P_1 must wait

- Step 2: for $i = 2$

  [0, 0, 1] not ≤ [0, 1, 0]

  - Finish[2] is False, and Request[2] ≤ Work is False
    => P_2 must wait

- Step 2: for i = 3

  [1, 0, 0] not ≤ [0, 1, 0]

  - Finish[3] is False, and Request[3] ≤ Work is False
    => P_3 must wait

Steps of Deadlock Detection

- Step 2: for $i = 4$

  [0, 0, 2] not ≤ [0, 1, 0]

  - Finish[4] is False, and Request[4] ≤ Work is False
    => P_4 must wait

- you can continue looping, but it won't make any difference. Therefore deadlock

  - Finish = [True, False, False, False, False]
    => could free up resources held by P_0, but would not be enough to satisfy the other processes.
    => Deadlock is between P_1, P_2, P_3, and P_4.

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily,
  - there may be many cycles in the resource graph
  - so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  - Priority of the process
  - How long process has computed, and how much longer to completion
  - Resources the process has used
  - Resources process needs to complete
  - How many processes will need to be terminated
  - Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- Selecting a victim –
  - minimize cost
- Rollback –
  - return to some safe state, restart process for that state
- Starvation –
  - same process may always be picked as victim, include number of rollback in cost factor