Session Plan

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

System Model

- System consists of resources
- Resource types $R_1, R_2, . . ., R_m$

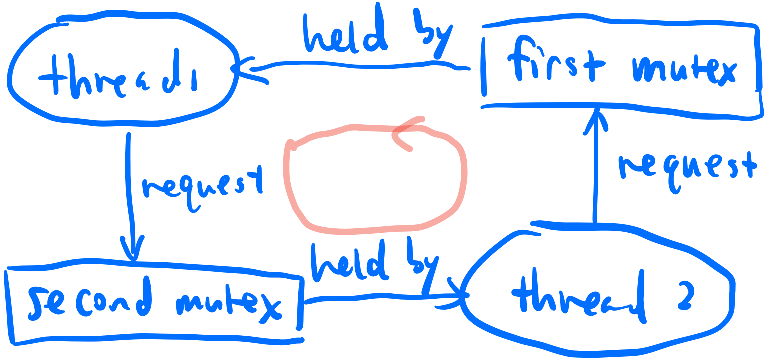
    CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
    - **request**
    - **use**
    - **release**

Deadlock in Multithreaded Applications

```
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

pthread_mutex_init(&first_mutex,NULL);
pthread_mutex_init(&second_mutex,NULL);
```

circular waiting




```
/* 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);
}
```

held by thread 1
?

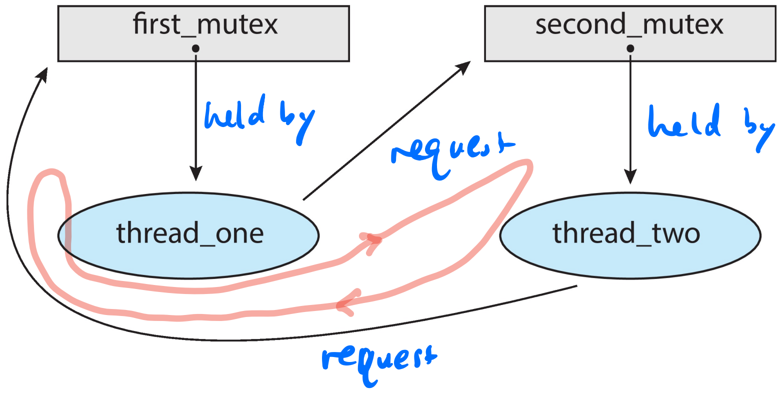
```
/* thread_two runs in this function */
void *do_work_two(void *param)
{
    pthread_mutex_lock(&second_mutex);
    pthread_mutex_lock(&first_mutex);
    /**
     * Do some work
     */
    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);

    pthread_exit(0);
}
```

held by thread 2
?

Deadlock in Multithreaded Applications

- Deadlock is possible if thread 1 acquires **`first_mutex`** and thread 2 acquires **`second_mutex`**. Thread 1 then waits for **`second_mutex`** and thread 2 waits for **`first_mutex`**.
- Can be illustrated with a **resource allocation graph**:



sharable read-only files

Deadlock Characterization

non-sharable writing in RW
chopsticks

- Deadlock can arise if four conditions hold simultaneously.
  - Mutual exclusion: only one process at a time can use a resource
  - Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes e.g., philosophers' chopsticks
  - No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task
  - Circular wait: there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

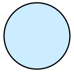
hold and wait

Resource-Allocation Graph

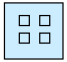
- A set of vertices V and a set of edges E.
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- request edge – directed edge $P_i \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (cont.)

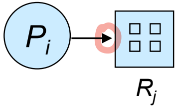
- Process



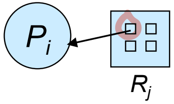
- Resource Type with 4 instances



- P_i requests instance of R_j



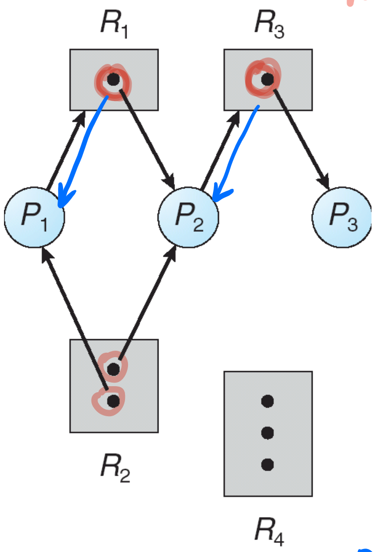
- P_i is holding an instance of R_j



Example of a Resource Allocation Graph

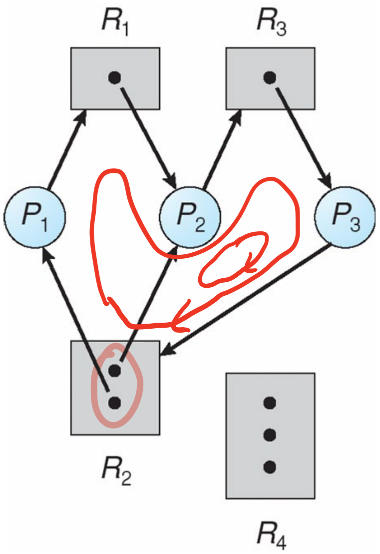
- One instance of R_1
- Two instances of R_2
- One instance of R_3
- Three instance of R_4
- P_1 holds one instance of R_2 and is waiting for an instance of R_1
- P_2 holds one instance of R_1, one instance of R_2, and is waiting for an instance of R_3
- P_3 is holds one instance of R_3

Not deadlocked

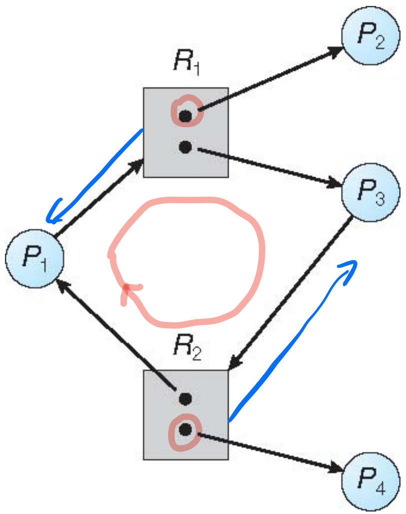


$\langle P_3, P_2, P_1 \rangle$

Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



< P2/P4, P1/P3 >

Basic Facts

no circular waiting

- 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

back edges ⇒ cycle

$O(V+E)$ time complexity of DFS

visit a node $O(1) * V$

Σ incident edges of each node $= O(E)$

sparse $\rightarrow O(n)$ dense $\rightarrow O(n^2)$

Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

Deadlock Prevention

- Restrain the ways request can be made
  - **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
  - **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources

    all at once

    all-or-nothing
    - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
    - Low resource utilization; starvation possible

Deadlock Prevention (cont.)

- **No Preemption** –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

App monitor Printer

$1 \rightarrow 2 \rightarrow 3$

Circular Wait

- Invalidating the circular wait condition is most common.
- Simply assign each resource (i.e. mutex locks) a unique number.
- Resources must be acquired in order.
- If:
  `first_mutex = 1`
  `second_mutex = 5`

  code for **`thread_two`** could not be written as follows:

```
/* 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);
}
```

Deadlock Avoidance

scheduling

- Requires that the system has some additional *a priori* information available
  - Simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need
  - The deadlock-avoidance algorithm 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

$\overbrace{P_1, P_2, \ldots, P_{i-1}}^{\text{completed}}, P_i$ being executed

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in safe state if there exists a sequence <P_1, P_2, …, P_n> of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$

  not held by anyone

  predecessors of P_i
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait 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, and so on

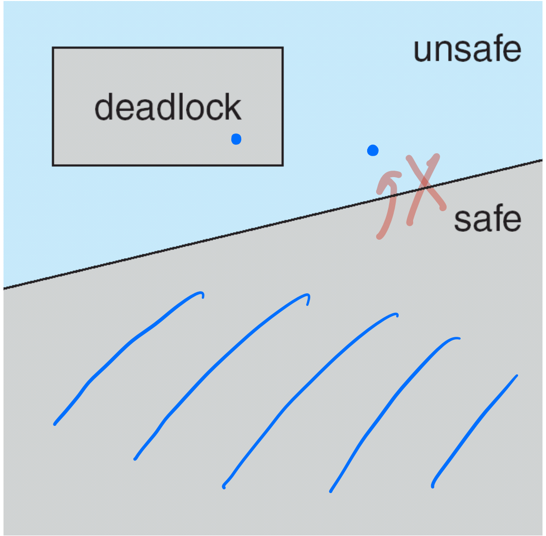
Basic Facts

we can complete all processes

- 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.

↳ never enter a deadlocked state

Safe, Unsafe, Deadlock State



Avoidance Algorithms

- Single instance of a resource type
  - Use a resource-allocation graph

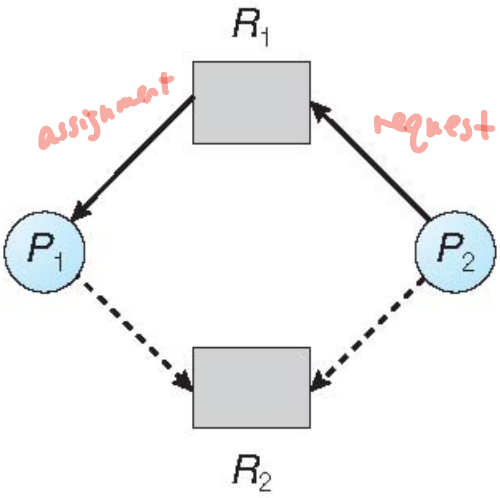
- Multiple instances of a resource type
  - Use the Banker's algorithm

Resource-Allocation Graph Scheme

request edges
assignment edges

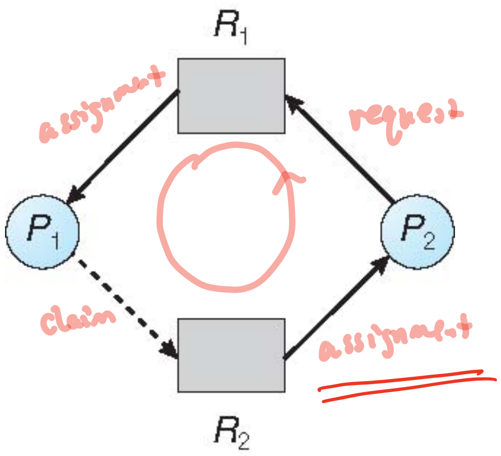
- Claim edge $P_i \rightarrow R_j$ indicated that process P_i may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system

Resource-Allocation Graph



t=0 safe <P1, P2>

Unsafe State in Resource-Allocation Graph



unsafe

t= 1 P2 requests R2
t= 2 R2 is assigned to P2 unsafe

t= 3 P1 requests R2 => deadlocked

Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

Banker's Algorithm

Dijkstra 1964

- Multiple instances
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

Data Structures for the Banker's Algorithm

unallocated

Let n = number of processes, and m = number of resources types.

- **Available**: Vector of length m. If available $[j] = k$, there are k instances of resource type R_j available

quota

- **Max**: $n \times m$ matrix. If $Max\,[i,j] = k$, then process P_i may request at most k instances of resource type R_j

currently allocated

- **Allocation**: $n \times m$ matrix. If Allocation$[i,j] = k$ then P_i is currently allocated k instances of R_j

future

- **Need**: $n \times m$ matrix. If $Need[i,j] = k$, then P_i may need k more instances of R_j to complete its task

$$Need\,[i,j] = Max[i,j] - Allocation\,[i,j]$$

① Allocation + Available = Total # of resources

② Need + Allocation = Max

Safety Algorithm

whether the current system is in a safe state or not

temporary

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

   Work = Available

   Finish **[*i*] = *false* for *i* = 0, 1, …, *n*- 1**

2. Find an ***i*** such that both:

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

   (b) $\boldsymbol{Need_i \leq Work}$

   If no such ***i*** exists, go to step 4

Simulation

3. $\boldsymbol{Work = Work + Allocation_i}$ pretend P_i is completed

   Finish*[*i*] = *true

   go to step 2

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

whether to approve the request or not

Resource-Request Algorithm for Process P_i

Request**$_i$* = request vector for process ***P**$_i$*. If ***Request**$_i$* **[*j*] = *k then process ***P**$_i$* wants ***k*** instances of resource type ***R**$_j$*

1. If ***Request**$_i$* $\leq$ ***Need**$_i$* go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If ***Request**$_i$* $\leq$ ***Available***, go to step 3. Otherwise ***P**$_i$* must wait, since resources are not available
3. Pretend to allocate requested resources to ***P**$_i$* by modifying the state as follows:

Simulation

***Available = Available – Request**$_i$;*

***Allocation**$_i$* **=** ***Allocation**$_i$* **+** ***Request**$_i$;*

***Need**$_i$* **=** ***Need**$_i$* **–** ***Request**$_i$;*

- If safe $\Rightarrow$ the resources are allocated to ***P**$_i$*
- If unsafe $\Rightarrow$ ***P**$_i$* must wait, and the old resource-allocation state is restored

if this request has been approved
$\Rightarrow$ enter a new state s'
run the safety algorithm on s'

$\Rightarrow$ we should not approve $request_i$

Example of Banker's Algorithm

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

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Example (cont.)

- The content of the matrix ***Need*** is defined to be ***Max – Allocation***

	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

- The system is in a safe state since the sequence < P_1, P_3, P_4, P_0, P_2> satisfies safety criteria

Safety algorithm

$m = 3 \qquad n = 5$

$$WORK = AVAILABLE = \begin{array}{ccc} A & B & C \\ 3 & 3 & 2 \end{array}$$

$$FINISH = \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \\ f & f & f & f & f \\ t & t & t & t & t \end{array}$$

For $i = 0$

$$Need_0 = \begin{array}{ccc} A & B & C \\ 7 & 4 & 3 \\ \vee & \vee & \vee \end{array}$$

$WORK = 3\ 3\ 2$

So P_0 must wait

For $i = 1$

$$Need_1 = \begin{array}{ccc} A & B & C \\ 1 & 2 & 2 \\ \wedge & \wedge & \| \end{array}$$

$\wedge$

$WORK = 3\ 3\ 2$

So p_1 can be kept in safe sequence

$\langle P_1 \rangle$

$WORK = WORK + Alloc_1$

$332 \qquad 200$

$= 532$

For $i = 2$

$$Need_2 = \begin{array}{ccc} A & B & C \\ 6 & 0 & 0 \\ \vee & \wedge & \wedge \end{array}$$

$\vee$

$WORK = 5\ 3\ 2$

So P_2 must wait

For $i = 3$

$$Need_3 = \begin{array}{ccc} A & B & C \\ 0 & 1 & 1 \\ \wedge & \wedge & \wedge \end{array}$$

$\wedge$

$WORK = 5\ 3\ 2$

So P_3 can be kept in safe sequence

$\langle P_1, P_3 \rangle$

$WORK = WORK + Alloc\ 3$

$= 532 + 211$

$= 743$

For $i = 4$

$$Need_4 = \begin{array}{ccc} A & B & C \\ 4 & 3 & 1 \\ \wedge & \wedge & \wedge \end{array}$$

$\wedge$

$WORK = 743$

So P_4 can be kept in safe sequence

$\langle P_1, P_3, P_4 \rangle$

$WORK = WORK + Alloc\ 4$

$= 743 + 002$

$= 745$

For $i = 0$

$$Need_0 = \begin{array}{ccc} A & B & C \\ 7 & 4 & 3 \end{array}$$

$\wedge$

$WORK = 745$

P_0 can be kept in safe seq

$\langle P_1, P_3, P_4, P_0 \rangle$

WORK = WORK + Alloc 0

= 745 + 010

= 755

For i = 2

$\text{Need}_2 = \begin{array}{ccc} A & B & C \\ 6 & 0 & 0 \end{array}$

$\wedge$

WORK = 755

P_2 can be kept in safe seq

$\langle P_1, P_3, P_4, P_0, P_2 \rangle$

a safe sequence

∴ The system is in a safe state

Example: P_1 Request (1,0,2)

A B C $request_1 = (1,0,2)$

1. $request_1 < Need_1$

   $102 < 122$

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true)

2. $request_1 < Available$

   $102 < 332$

S' ⇒

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

3. pretend $request_1$ is approved

   $Need_1 = Need_1 - request_1$

   $= 122 - 102 = 020$

   $Alloc_1 = Alloc_1 + request_1$

   $= 200 + 102 = 302$

   $Available = Available - request_1$

   $= 332 - 102$

   $= 230$

- Executing safety algorithm shows that sequence < P_1, P_3, P_4, P_0, P_2> satisfies safety requirement
- Can request for (3,3,0) by $\boldsymbol{P_4}$ be granted?
- Can request for (0,2,0) by $\boldsymbol{P_0}$ be granted?

run the safety algo on S'

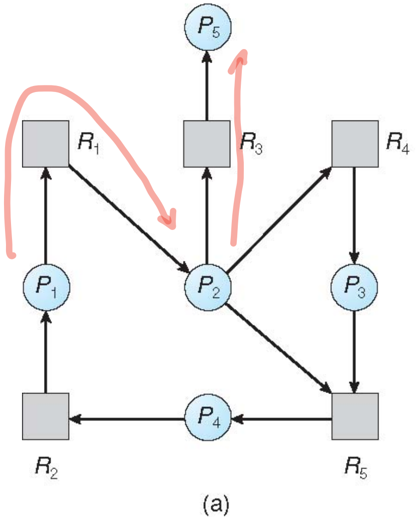
Deadlock Detection

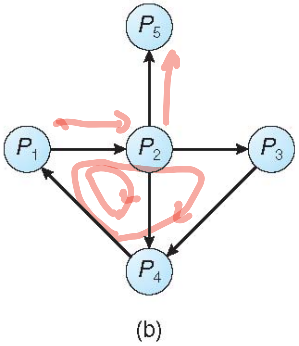
- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

Single Instance of Each Resource Type

- Maintain wait-for graph
  - Nodes 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
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph





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.

Need

Safety algorithm is the detection version

Detection Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length ***m*** and ***n***, respectively
   Initialize:
   (a) ***Work = Available***
   (b) For ***i* = 1,2, …, *n***, if $\boldsymbol{Allocation_i \neq 0}$, then
   Finish*[i] = *false; otherwise, ***Finish*[i] = *true***

2. Find an index ***i*** such that both:
   (a) ***Finish*[*i*] == *false***
   (b) $\boldsymbol{Request_i \leq Work}$

   If no such ***i*** exists, go to step 4

Detection Algorithm (cont.)

3. $\textbf{\textit{Work}} = \textbf{\textit{Work}} + \textbf{\textit{Allocation}}_i$
   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 P_i is deadlocked

Algorithm requires an order of $O(m \times n^2)$ 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***

Example (Cont.)

- $\boldsymbol{P_2}$ requests an additional instance of type $\boldsymbol{C}$

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

- State of system?
  - Can reclaim resources held by process $\boldsymbol{P_0}$, but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes $\boldsymbol{P_1}$, $\boldsymbol{P_2}$, $\boldsymbol{P_3}$, and $\boldsymbol{P_4}$

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 and 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?
    1. Priority of the process
    2. How long process has computed, and how much longer to completion
    3. Resources the process has used
    4. Resources process needs to complete
    5. How many processes will need to be terminated
    6. 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

Exit Slips

- Take 1-2 minutes to reflect on this lecture
- On a sheet of paper write:
  - One thing you learned in this lecture
  - One thing you didn’t understand