

CS370 Operating Systems

Colorado State University

Yashwant K Malaiya

Fall 2025 L13

Deadlocks



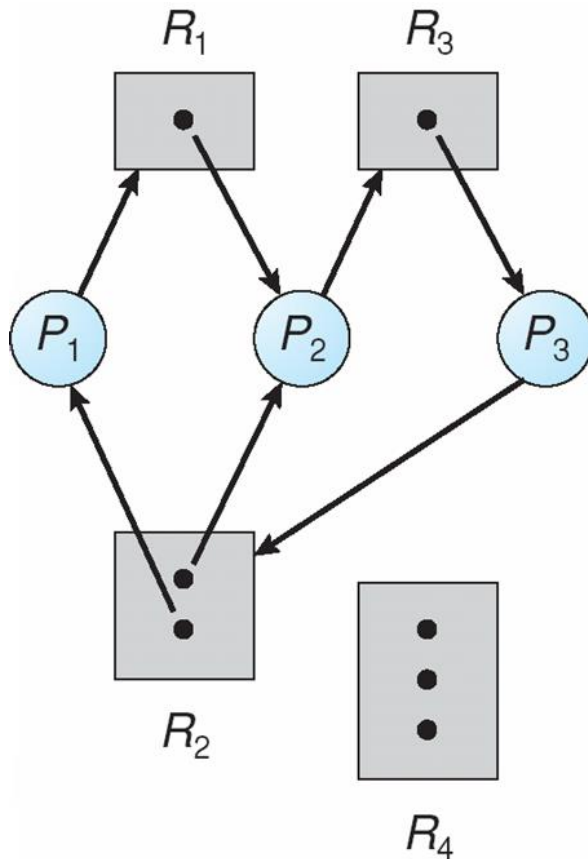
Slides based on

- Text by Silberschatz, Galvin, Gagne
- Various sources

Chapter 8: Deadlocks

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

Resource Allocation Graph With A Deadlock



Does a deadlock exist?

At this point, two minimal cycles exist in the system:

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

Processes P_1 , P_2 , and P_3 are deadlocked.

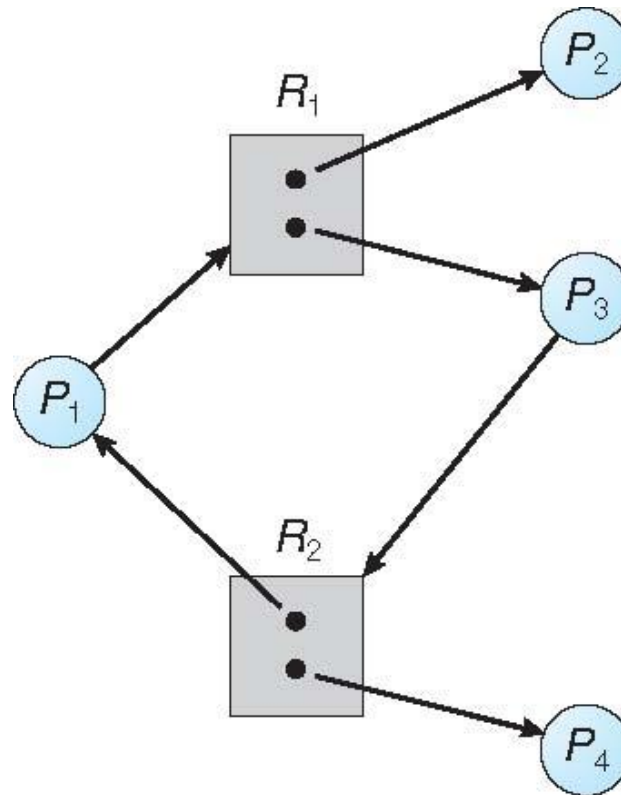
Graph With A Cycle But No Deadlock

Is there a deadlock?

P_4 will release its instance of resource type R_2 . That resource can then be allocated to P_3 , breaking the cycle. Thus, there is no deadlock.

If a resource-allocation graph does not have a cycle, then the system is **not** in a deadlocked state.

If there is a cycle, then the system may or may not be in a deadlocked state.



Basic Facts

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

Methods for Handling Deadlocks

Approach	Resource allocation policy	Scheme	Notes
Prevention	Conservative, undercommits resources	Requesting all resources at once	Good for processes with a single burst of activity
		Preemption	Good when preemption cost is small
		Resource ordering	Compile time enforcement possible
Avoidance	midway	Find at least one safe path (dynamic)	Future max requirement must be known
Detection	Liberal	Invoked periodically	Preemption may be needed

Deadlock Prevention

For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can **prevent** the occurrence of a deadlock.

- **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
- **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 $\{P_0, P_1, \dots, P_n\}$ of waiting processes that are circularly waiting.



Deadlock Prevention: Limit Mutual Exclusion

For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can **prevent** the occurrence of a deadlock.

Restrain the ways request can be made:

- **Limit Mutual Exclusion** –
 - not required for sharable resources (e.g., read-only files)
 - (Mutual Exclusion must hold for non-sharable resources)



Deadlock Prevention: Limit Hold and Wait

- **Limit Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
 1. Require process to request and be allocated all its resources before it begins execution
 2. Allow a process to request resources when it is holding none.

Ex: Copy data from DVD, sort file, and print

 - First request DVD and disk file
 - Then request file and printer,
 - then start
- Disadvantage: starvation possible

Deadlock Prevention: Limit No Preemption

- **Limit 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

Deadlock Prevention: Limit Circular Wait

- **Limit Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration
- Assign each resource a unique number
 - Disk drive: 1
 - Printer: 2 ...
 - Request resources in increasing order
 - *Example soon*

Dining philosophers problem: Necessary conditions **for** deadlock

Relax conditions to
avoid deadlock

- Mutual exclusion
 - 2 philosophers *cannot share* the same chopstick
- Hold-and-wait
 - A philosopher *picks up one* chopstick at a time
 - Will not let go of the first while it *waits for the second* one
- No preemption
 - A philosopher *does not snatch chopsticks* held by some other philosopher
- Circular wait
 - Could happen if each philosopher *picks chopstick with the same hand* first

Deadlock Example: numbering

```
/* 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);
}
```

Assume that thread one is the first to acquire the locks and does so in the order (1) first mutex, (2) second mutex.

Solution: Lock-order verifier

“**Witness**” records the relationship that **first mutex must be acquired before second mutex**. If thread two later acquires the locks out of order, witness generates a warning message on the system console.

Allows deadlock. Redesign to avoid.

Deadlock may happen *even* 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);
}
```

Lock ordering:
First *from* lock, then *to* lock

Ex: 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 is possible, even with lock ordering.

Deadlock Avoidance

Manage resource allocation to ensure the system never enters an unsafe state.

Deadlock Avoidance

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

Avoidance: amount and type of information needed

- **Resource allocation state**
 - Number of available and allocated resources
 - Maximum demands of processes
- *Dynamically* examine resource allocation state
 - Ensure circular-wait cannot exist
- Simplest model:
 - Declare maximum number of resources for each type
 - Use information to avoid deadlock

Safe Sequence

System must decide if immediate allocation leaves the system in a safe state

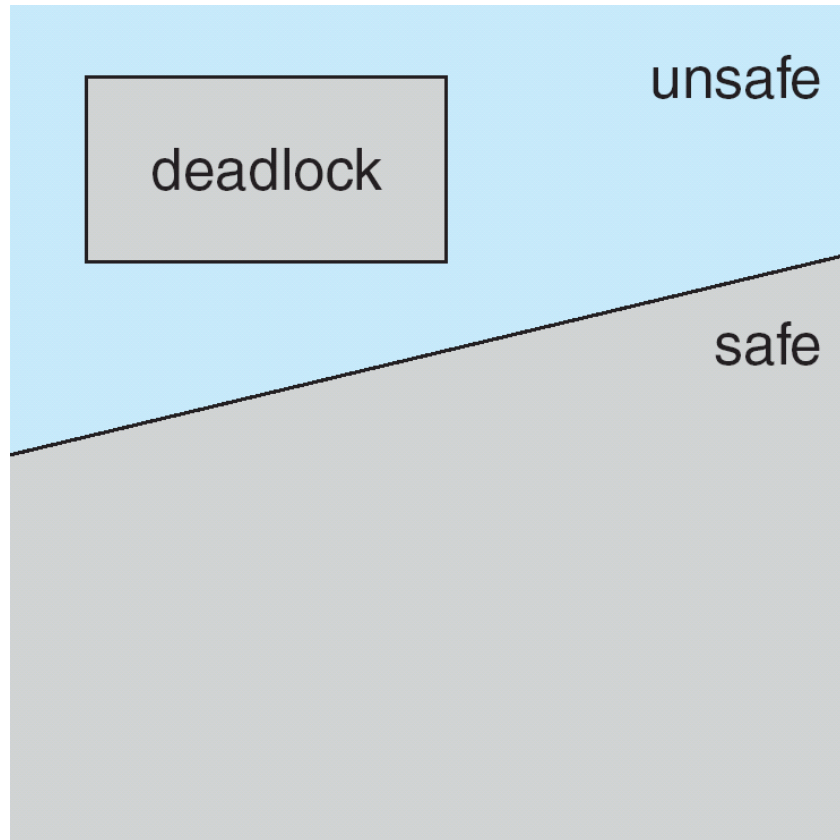
System is in **safe state** if there exists a **sequence** $\langle P_1, P_2, \dots, P_n \rangle$ of ALL the processes such that

- 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 $j < i$
 - That is
 - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished and released resources
 - When P_i terminates, P_{i+1} can obtain its needed resources, and so on
- If no such sequence exists: system state is **unsafe**

Deadlock avoidance: Safe states

- If the system can:
 - Allocate resources to each process in some order
 - Up to the maximum for the process
 - Still avoid deadlock
 - Then it is in a **safe state**
- A system is safe ONLY IF there is a safe sequence
- A safe state is not a deadlocked state
 - Deadlocked state is an unsafe state
 - Not all unsafe states are deadlock

Safe, Unsafe, Deadlock State



Examples of safe and unsafe states in next 3 slides

Example A: Assume 12 Units in the system

	Max need	Current holding
av		3
P0	10	5
P1	4	2
P2	9	2

At time T0 (shown):

9 units allocated

3 (12-9) units available


*A unit could be a drive,
a block of memory etc.*

- Is the system at time **T0** in a safe state?
 - Try sequence $\langle P1, P0, P2 \rangle$
 - P1 can be given 2 units
 - When P1 releases its resources; there are now 5 available units
 - P0 uses 5 and subsequently releases them (10 available now)
 - P2 can then proceed.
- Thus $\langle P1, P0, P2 \rangle$ is a safe sequence, and at T0 system was in a safe state

More detailed look 

Example A: Assume 12 Units in the system (timing)

Is the state at T0 safe? Detailed look for instants T0, T1, T2, etc..



	Max need	Current holding	+2 allo to P1	P1 releases all
		T0	T1	T2	T3	T4	T5
av		3	1	5	0	10	3
P0	10	5	5	5	10 done	0	0
P1	4	2	4 done	0	0	0	0
P2	9	2	2	2	2	2	9 done

Thus the state at T0 is safe.

Example B: 12 Units initially available in the system

	Max need	T0	T1 safe?
Av		3	2
P0	10	5	5
P1	4	2	2
P2	9	2	3 Is that OK?

Before T1:

3 units available

At T1:

2 units available

- At time **T1**, P2 is allocated 1 more units. Is that a good decision?
 - Now only P1 can proceed (already has 2, and given be given 2 more)
 - When P1 releases its resources; there are 4 units
 - P0 needs 5 more, P2 needs 6 more. Deadlock.
 - **Mistake** in granting P2 the additional unit.
- The state at **T1** is not a safe state. Wasn't a good decision.

Avoidance Algorithms

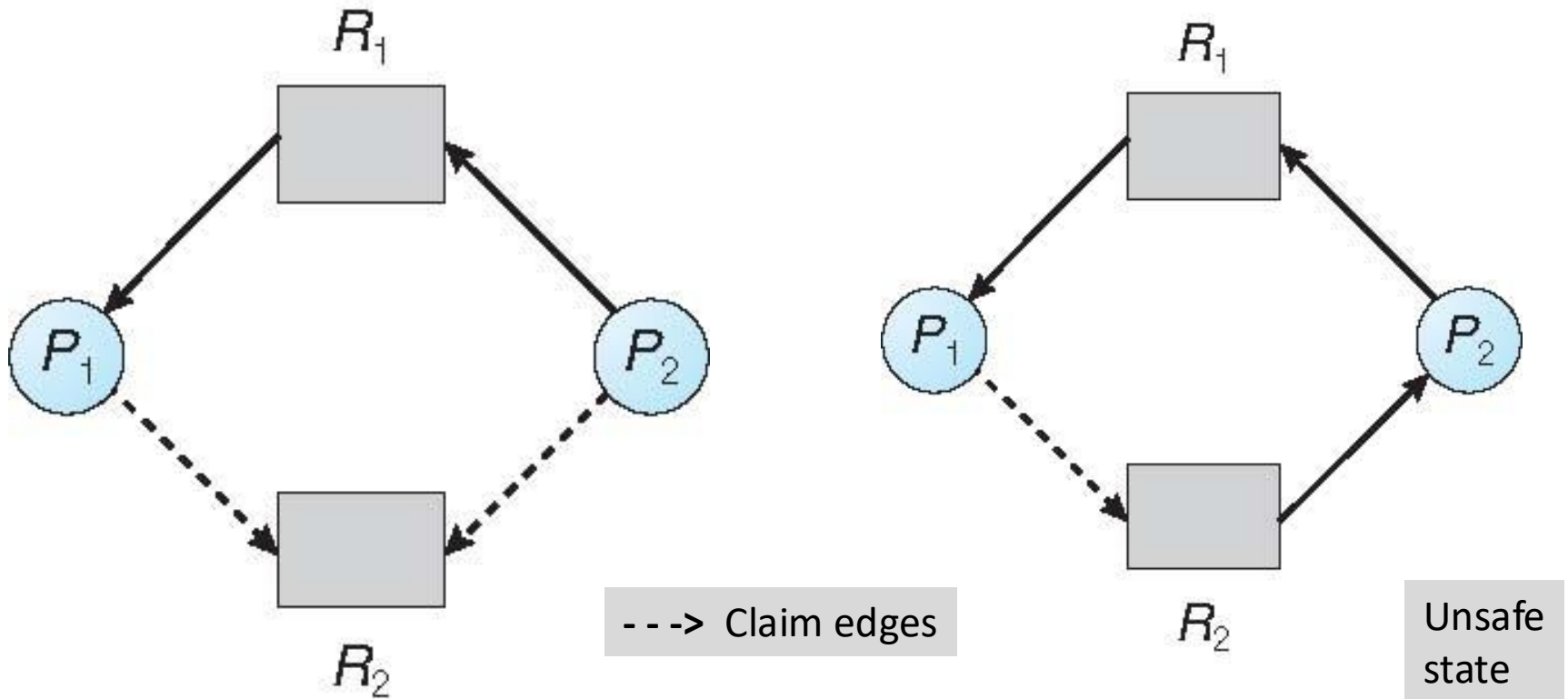
- Dynamic
- Single instance of a resource type
 - Use a resource-allocation graph scheme
- Multiple instances of a resource type
 - Use the banker's algorithm (Dijkstra)



Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_i **may** request resource R_j ; represented by a **dashed** line. This is new.
- 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
- Requirement: Resources must be claimed *a priori* in the system

Resource-Allocation Graph



Suppose P_2 requests R_2 . Can R_2 be allocated to P_2 ?

Although R_2 is currently free, we cannot allocate it to P_2 , since this action will create a cycle getting system in an unsafe state. If P_1 requests R_2 , and P_2 requests R_1 , then a deadlock will occur.

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: examining a request

- Multiple instances of resources.
- Each process must a priori claim maximum use
- When a process requests a resource,
 - it may have to wait until the resource becomes available ([resource request algorithm](#))
 - Request should not be granted if the resulting system state is unsafe ([safety algorithm](#))
- When a process gets all its resources it must return them in a finite amount of time
- Modeled after a banker in a small-town making loans.

Data Structures for the Banker's Algorithm

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

Processes vs resources:

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

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

Safety Algorithm: Is this a safe state?

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

Work = Initially Available resources

Finish [i] = initially false for $i = 0, 1, \dots, n-1$ (processes done)

2. Find a process i such that both:

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

(b) **Need** _{i} ≤ **Work**

If no such i exists, go to step 4

3. **Work** = **Work** + **Allocation** _{i}

Finish [i] = true

go to step 2

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

n = number of processes,
 m = number of resources types
Need _{i} : additional res needed
Work: res currently free
Finish _{i} : processes finished
Allocation _{i} : allocated to i

Resource-Request Algorithm for Process P_i

Notation: $Request_i$ = request vector for process P_i .

If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j

Algorithm: *Should the allocation request be granted?*

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. **Is allocation safe?:** **Pretend** to allocate requested resources to P_i by modifying the state as follows:
 - $Available = Available - Request_i$
 - $Allocation_i = Allocation_i + Request_i$
 - $Need_i = Need_i - Request_i$
 - If safe \Rightarrow the resources are allocated to P_i
 - If unsafe $\Rightarrow P_i$ must wait, and the old resource-allocation state is preserved.

Use safety algorithm here

Example 1A: Banker's Algorithm

- 5 processes P_0 through P_4 ;
- 3 resource types: A (10 instances), B (5 instances), and C (7 instances)
- Is it a safe state?

The Need matrix is redundant

Process	Max			Allocation			Need		
type	A	B	C	A	B	C	A	B	C
Currently available				3	3	2			
P0	7	5	3	0	1	0	7	4	3
P1	3	2	2	2	0	0	1	2	2
P2	9	0	2	3	0	2	6	0	0
P3	2	2	2	2	1	1	0	1	1
P4	4	3	3	0	0	2	4	3	1

Example 1A: Banker's Algorithm

- Is it a safe state?
- Yes, since the sequence $\langle P1, P3, P4, P2, P0 \rangle$ satisfies safety criteria

How did we get to this state?

Process	Max			Allocation			Need		
type	A	B	C	A	B	C	A	B	C
available				3	3	2			
P0	7	5	3	0	1	0	7	4	3
P1	3	2	2	2	0	0	1	2	2
P2	9	0	2	3	0	2	6	0	0
P3	2	2	2	2	1	1	0	1	1
P4	4	3	3	0	0	2	4	3	1

"Work"

Why did we choose P1?

P1 run to completion. Available becomes $[3\ 3\ 2] + [2\ 0\ 0] = [5\ 3\ 2]$

P3 run to completion. Available becomes $[5\ 3\ 2] + [2\ 1\ 1] = [7\ 4\ 3]$

P4 run to completion. Available becomes $[7\ 4\ 3] + [0\ 0\ 2] = [7\ 4\ 5]$

P2 run to completion. Available becomes $[7\ 4\ 5] + [3\ 0\ 2] = [10\ 4\ 7]$

P0 run to completion. Available becomes $[10\ 4\ 7] + [0\ 1\ 0] = [10\ 5\ 7]$

Hence state above is safe.

Ex 1B: Assume now P_1 Requests (1,0,2)

- Check that $Request_i \leq Need_i$ **and** $Request_i \leq Available$. $(1,0,2) \leq (3,3,2) \rightarrow \text{true}$.
- Check for safety after pretend allocation. P_1 allocation would be $(2\ 0\ 0) + (1\ 0\ 2) = 3\ 0\ 2$

Process	Max			Pretend Allocation			Need		
type	A	B	C	A	B	C	A	B	C
Available				2	3	0			
P0	7	5	3	0	1	0	7	4	3
P1	3	2	2	3	0	2	0	2	0
P2	9	0	2	3	0	2	6	0	0
P3	2	2	2	2	1	1	0	1	1
P4	4	3	3	0	0	2	4	3	1

Sequence $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ satisfies safety requirement.

Hence state above is safe, thus the allocation would be safe.

Ex 1C,1D: Additional Requests ..

- Given State is (same as previous slide)

Process	Max			Allocation			Need		
type	A	B	C	A	B	C	A	B	C
available				2	3	0			
P0	7	5	3	0	1	0	7	4	3
P1	3	2	2	3	0	2	0	2	0
P2	9	0	2	3	0	2	6	0	0
P3	2	2	2	2	1	1	0	1	1
P4	4	3	3	0	0	2	4	3	1

P4 request for (3,3,0): cannot be granted - resources are not available.

P0 request for (0,2,0): cannot be granted since the resulting state is unsafe.

Check yourself.

Bankers Algorithm: Practical Issues

- Processes may not know in advance about their maximum resource needs
- Number of processes is not fixed
 - Varies dynamically
- Resources thought to be available can disappear
- Few systems use this algorithm

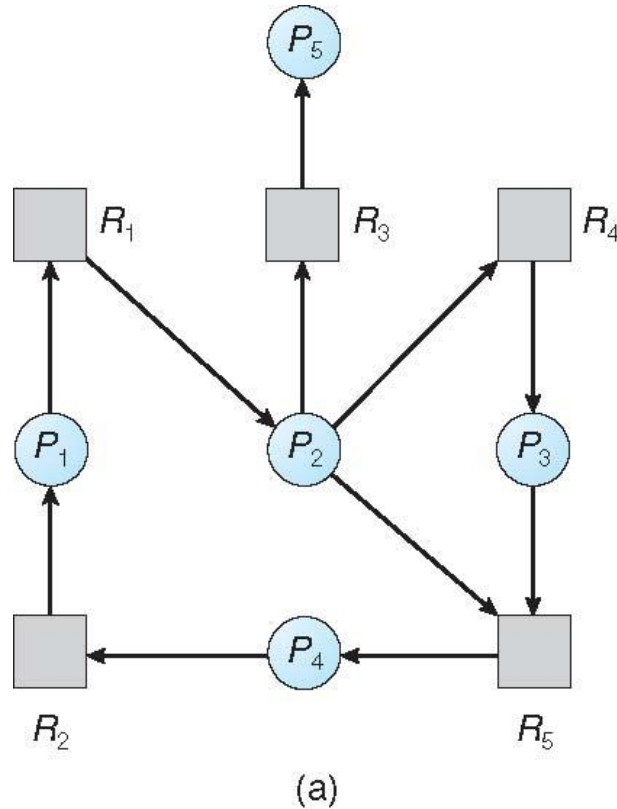
Deadlock Detection

- Allow system to enter deadlock state. If that happens, detect the deadlock and do something about it.
- Detection algorithm
 - Single instance of each resource:
 - wait-for graph
 - Multiple instances:
 - detection algorithm (based on Banker's algorithm)
- Recovery scheme

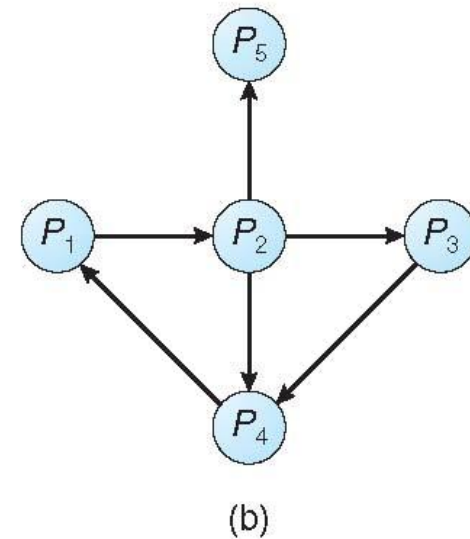
Single Instance of Each Resource Type

- Maintain **wait-for** graph (based on resource allocation graph)
 - Nodes are processes
 - $P_i \rightarrow P_j$ if P_i is waiting for P_j
 - *Deadlock if cycles*
- 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

Has cycles. Deadlock.

Several Instances of a Resource Type

Banker's algorithm: Can requests by all process be satisfied?

- **Available:** A vector of length m indicates the number of available (currently free) resources of each type
- **Allocation:** An $n \times m$ matrix defines the number of resources of each type currently allocated to each process
- **Request:** An $n \times m$ 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** = initially available
 - (b) For $i = 1, 2, \dots, n$, if **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) **Request_i** \leq **Work**If no such i exists, go to step 4
3. **Work = Work + Allocation_i**
Finish[i] = true
go to step 2 (find next process)
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

n = number of processes,
 m = number of resources types
Work: res currently free
Finish_i: processes finished
Allocation_i: allocated to i

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)
- Sequence $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in ***Finish[i] = true*** for all i. **No deadlock**

Process	Allocation			Request		
type	A	B	C	A	B	C
available	0	0	0			
P0	0	1	0	0	0	0
P1	2	0	0	2	0	2
P2	3	0	3	0	0	0
P3	2	1	1	1	0	0
P4	0	0	2	0	0	2

After	work		
ini	0	0	0
P0	0	1	0
P2	3	1	3
P3	5	2	4
P1	7	2	4
P4	7	2	6

sequence

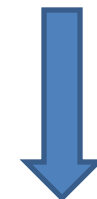
Example of Detection Algorithm (cont)

- P_2 requests an additional instance of type C

Process	Allocation			Request		
type	A	B	C	A	B	C
available	0	0	0			
P0	0	1	0	0	0	0
P1	2	0	0	2	0	2
P2	3	0	3	0	0	1
P3	2	1	1	1	0	0
P4	0	0	2	0	0	2

Sequence

After	work		
ini	0	0	0
P0	0	1	0
P2	-	-	-



- State of system?
 - Can reclaim resources held by process P_0 , but insufficient resources to fulfill other processes' requests
 - Deadlock exists, consisting of processes 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 and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

Recovery from Deadlock: Process Termination

Choices

- 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 rollbacks in cost factor

Deadlock recovery through rollbacks

- **Checkpoint** process periodically
 - Contains memory image and resource state
- Deadlock detection tells us *which* resources are needed
- Process owning a needed resource
 - **Rolled back** to before it acquired needed resource
 - Work done since rolled back checkpoint discarded
 - **Assign** resource to deadlocked process

Livelocks

In a livelock two processes need each other's resource

- Both run and make no progress, but neither process blocks
- Use CPU quantum over and over without making progress

Ex: If fork fails because process table is full

- Wait for some time and try again
- But there could be a collection of processes each trying to do the same thing
- Avoided by ensuring that only one process (chosen randomly or by priority) takes action

Two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass. But they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time.