Chapter 7: Deadlocks

Deadlocks

- Objectives
  - To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
  - To present a number of different methods for preventing or avoiding deadlocks in a computer system
- Topics
  - The Deadlock Problem
  - System Model
  - Deadlock Characterization
  - Methods for Handling Deadlocks
  - Deadlock Avoidance
  - Deadlock Detection
  - Recovery from Deadlocks

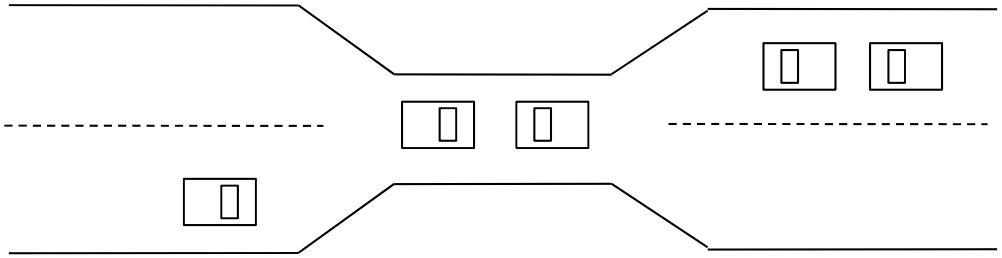
The Deadlock Problem

- Assume a set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
- Example set-up
  - A system operates with 2 disk drives
  - P_1 and P_2 each hold one disk drive and each needs another one
- Example usage
  - Semaphores *A* and *B*, initialized to 1

P_0	P_1
wait (A);	wait(B)
wait (B);	wait(A)

Bridge Crossing Example

- Traffic only possible in one direction at any time



- Each entry section can be viewed as a resource
- If a deadlock occurs, it can only be resolved, if one car backs up
  - Preempting the resource allocation and rollback
- Several cars may have to back up, if a deadlock occurs
- Starvation is possible
- **Note**: Most OSes do not prevent or deal with deadlocks

System Model

- Finite number of 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 (normal operation)
  - Request
  - Use
  - Release
- Request and release are system calls!

Deadlock Characterization (1)

- Deadlocks can arise only, if four conditions hold simultaneously (necessary conditions)

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

Deadlock Characterization (2)

3. No 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 $\{P_0, P_1, \ldots, P_0\}$ 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

Resource Allocation Graph (1)

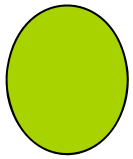
- Deadlocks can be described precisely in terms of a directed graph, the Resource Allocation Graph
  - A set of vertices V and a set of edges E
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$,
    the set consisting of all processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$,
    the set consisting of all resource types in the system

- Request edge – directed edge $P_i \rightarrow R_j$
- Assignment edge – directed edge $R_j \rightarrow P_i$

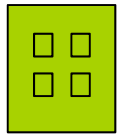
Resource Allocation Graph (2)

Legend

- Process

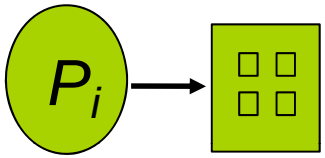


- Resource type with 4 instances

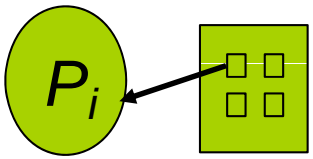


□ or •

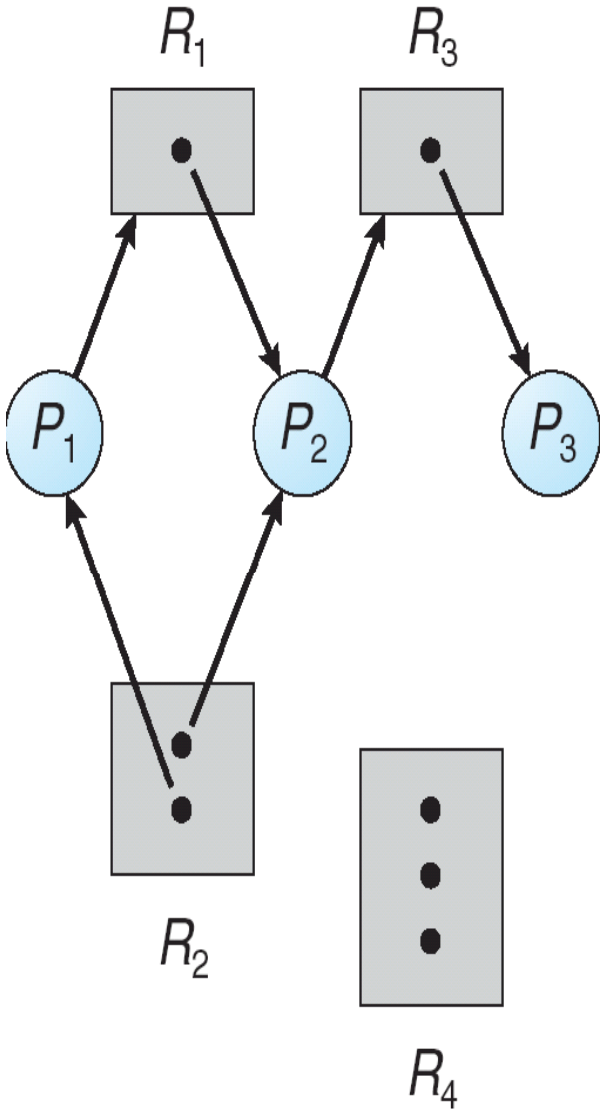
- P_i requests instance of R_j



- P_i is holding an instance of R_j

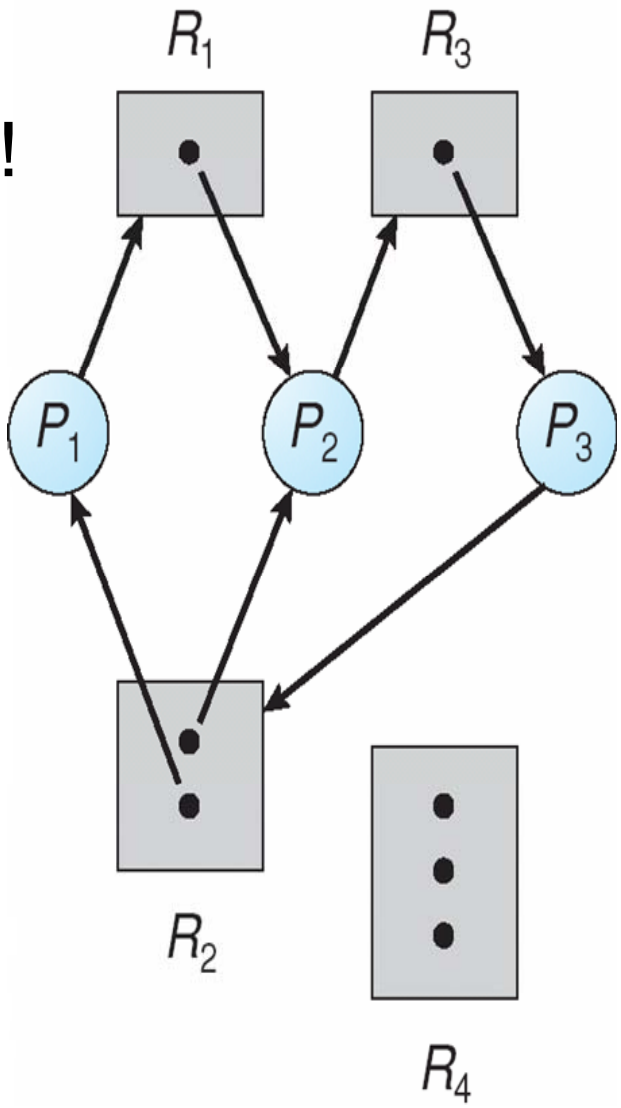


Example: Resource Allocation Graph (1)



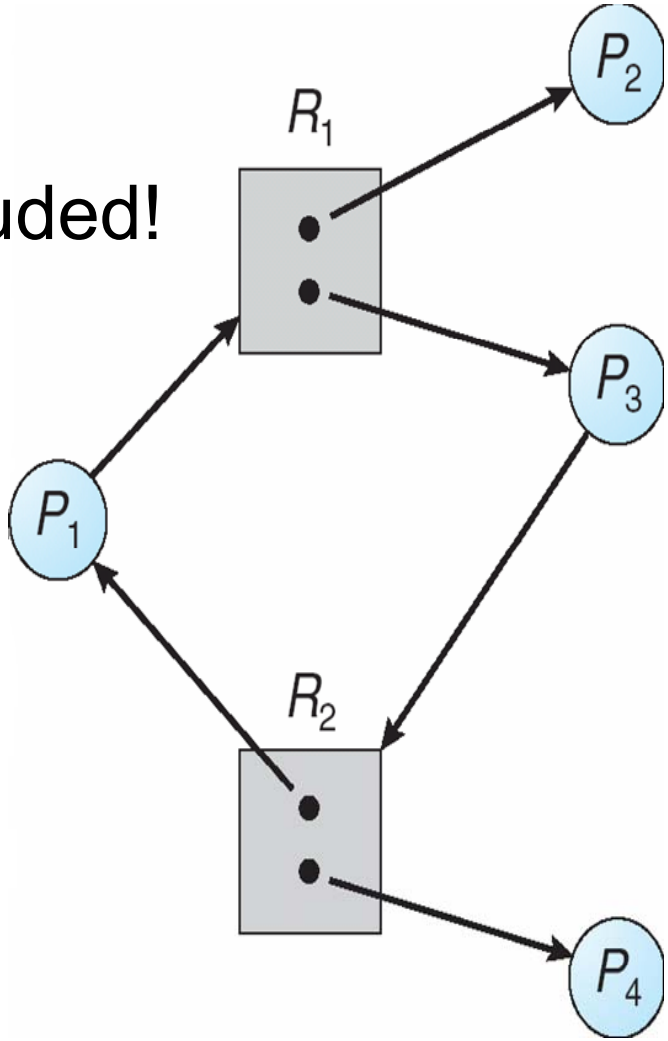
Example: Resource Allocation Graph (2)

Deadlock included!



Example: Resource Allocation Graph (3)

No Deadlock,
but a cycle included!



Basic Observation and Facts

- If a graph contains no cycles ⇒ no deadlock
- If a graph contains a cycle ⇒
  - If only one instance per resource type available
    ⇒ deadlock
  - If several instances per resource type available
    ⇒ only the possibility of deadlocks exists

Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state
- 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 Avoidance (1)

- Restrain the ways resource requests can be made

- Mutual Exclusion
  - Not required for sharable resources; must hold for non-sharable resources
- Hold and Wait
  - Must guarantee that whenever a process requests a resource, it does not hold any other (unrelated) resources
  - Requires process to request and see all its resources allocated before it begins with the execution, or allow the process to request resources only, when the process has none
  - Low resource utilization; starvation possible

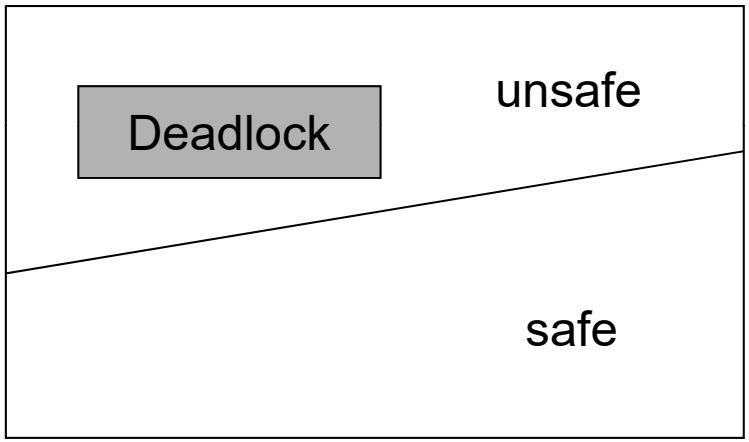
Deadlock Avoidance (2)

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

Safe State (1)

- Additional information on the “how” of resource requests, such as orders of resource needs or numbers of resources demanded, to be exploited

- When a process requests an available resource, the system must decide, if an immediate allocation leaves the system in a safe state!



Safe State (2)

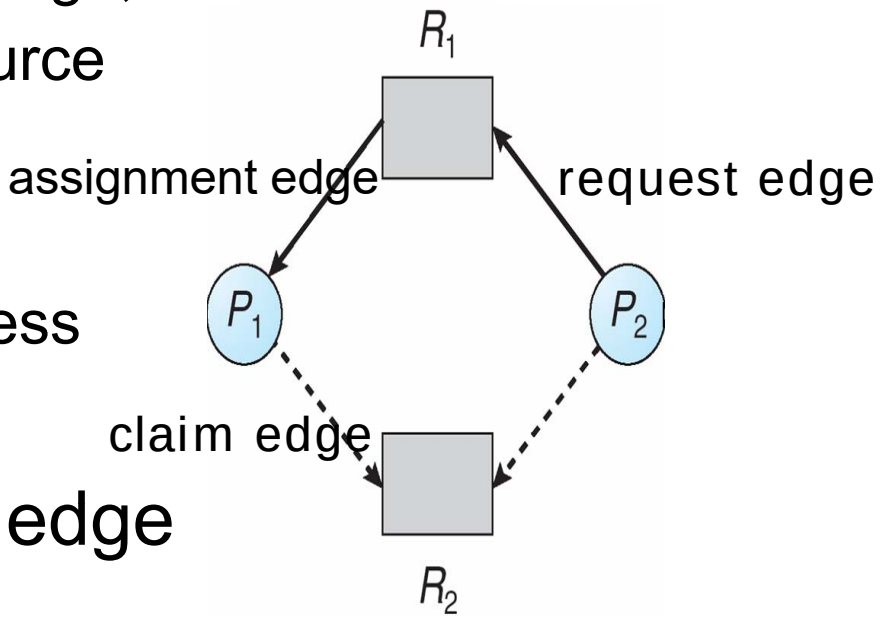
- A system is in a safe state, if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL processes in the system such that for each P_i, resources that P_i can still request can be satisfied by currently available resources plus those resources held by all P_j, with $j < i$
  - Ordering of processes
- That is:
  - If P_i's resource needs are not immediately available, P_i can wait until all $P_{j<i}$ have finished
  - When P_j is finished, P_i can obtain resources needed, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its resources needed …

Avoidance Algorithms

- Single instance of a resource type
  - Use a Resource Allocation Graph and cycle detection
- Multiple instances of a resource type
  - Use Banker's Algorithm

Resource Allocation Graph Scheme

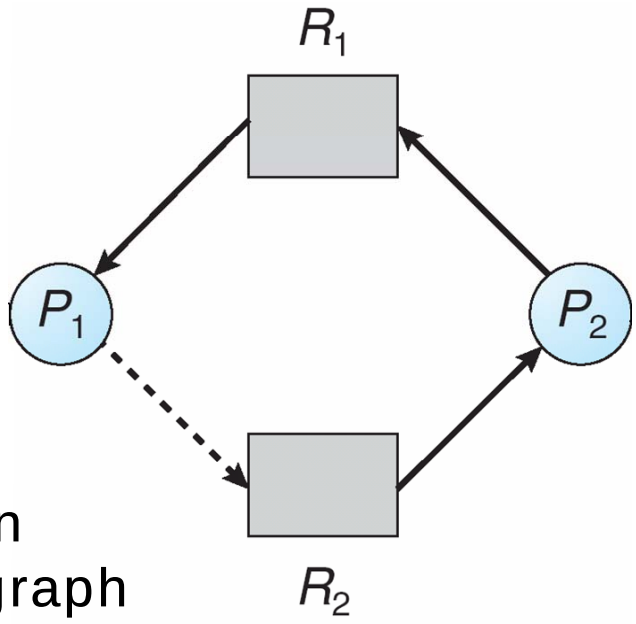
- Claim edge P_i –> R_j indicated that process P_i may request resource R_j; represented by a dashed line
  - Claim edge converts to request edge, when a process requests a resource
  - Request edge converted to an assignment edge, when the resource is allocated to the process



- When a resource is released by a process, the assignment edge converts back to a claim edge
- Resources must be claimed *a priori* in the system

Resource Allocation Graph Algorithm

- Suppose that process P_2 requests a resource R_2
- 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
  - Cycle detection algorithm on resource graph after potential assignment of resource to requesting process



Unsafe state, cycle in resource allocation graph

Banker's Algorithm

- Multiple instances of resources
- Each process must claim maximum use a priori
- 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 Structure for Banker's Algorithm (1)

- Let n = number of processes and
  m = number of resources types

- Available
  - Vector of length m
  - *Available[j]* = *k*, there are *k* instances of resource type R_j available
- Max
  - *n* x *m* matrix
  - *Max[i,j]* = *k*, process P_i may request at most *k* instances of resource type R_j

Data Structure for Banker's Algorithm (2)

- Allocation
  - n x m matrix.
  - *Allocation*[*i,j*] = k, process P_i is currently allocated k instances of R_j
- Need
  - n x m matrix
  - *Need*[*i,j*] = k, process P_i may need k more instances of R_j to complete its task
    - N*eed*[*i,j*] = *Max*[*i,j*] – *Allocation*[*i,j*]

Safety Determination Algorithm (1)

How to find out that a system is in a safe state?

Let *Work* and *Finish* be temporary vectors of length *m* and *n*, respectively

1. Initialize

   Work = *Available*

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

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

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

   (b) $Need_i \leq Work_i$

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

Safety Determination Algorithm (2)

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,
   else unsafe

- Requires O(*m* x n^2) operations
  - Tries to find possible sequence of processes finishing and releasing their resources

Resource Request Algorithm for Process P_i

How to determine if a request can be safely granted?

- *Request* = request vector for process P_i
  - $Request_i[j] = k$, process P_i wants k instances of resource 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 yet available
3. Pretend to allocate requested resources to P_i by modifying the state as follows:
   - $Available = Available - Request_i$;
   - $Allocation_i = Allocation_i + Request_i$;
   - $Need_i = Need_i - Request_i$;
   - *If safe ⇒ Resources are allocated and P_i can execute*
   - *If unsafe ⇒ P_i must wait, old resource allocation state restored*

Example of Banker's Algorithm (1)

- 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	

then ….

Example of Banker's Algorithm (2)

- 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 $< P_1, P_3, P_4, P_2, P_0>$ satisfies the safety criteria

Example: P_1 Requests (1,0,2)

- Now check that the new Request ≤ Available
  that is, (1,0,2) ≤ (3,3,2) ["old" available value] ⇒ true
- Check new possible state

	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 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence
  < P_1, P_3, P_4, P_0, P_2> satisfies safety requirement
  - Can next request for (3,3,0) by P_4 be granted? No, lack of R.
  - Can next request for (0,2,0) by P_0 be granted? No, unsafe!

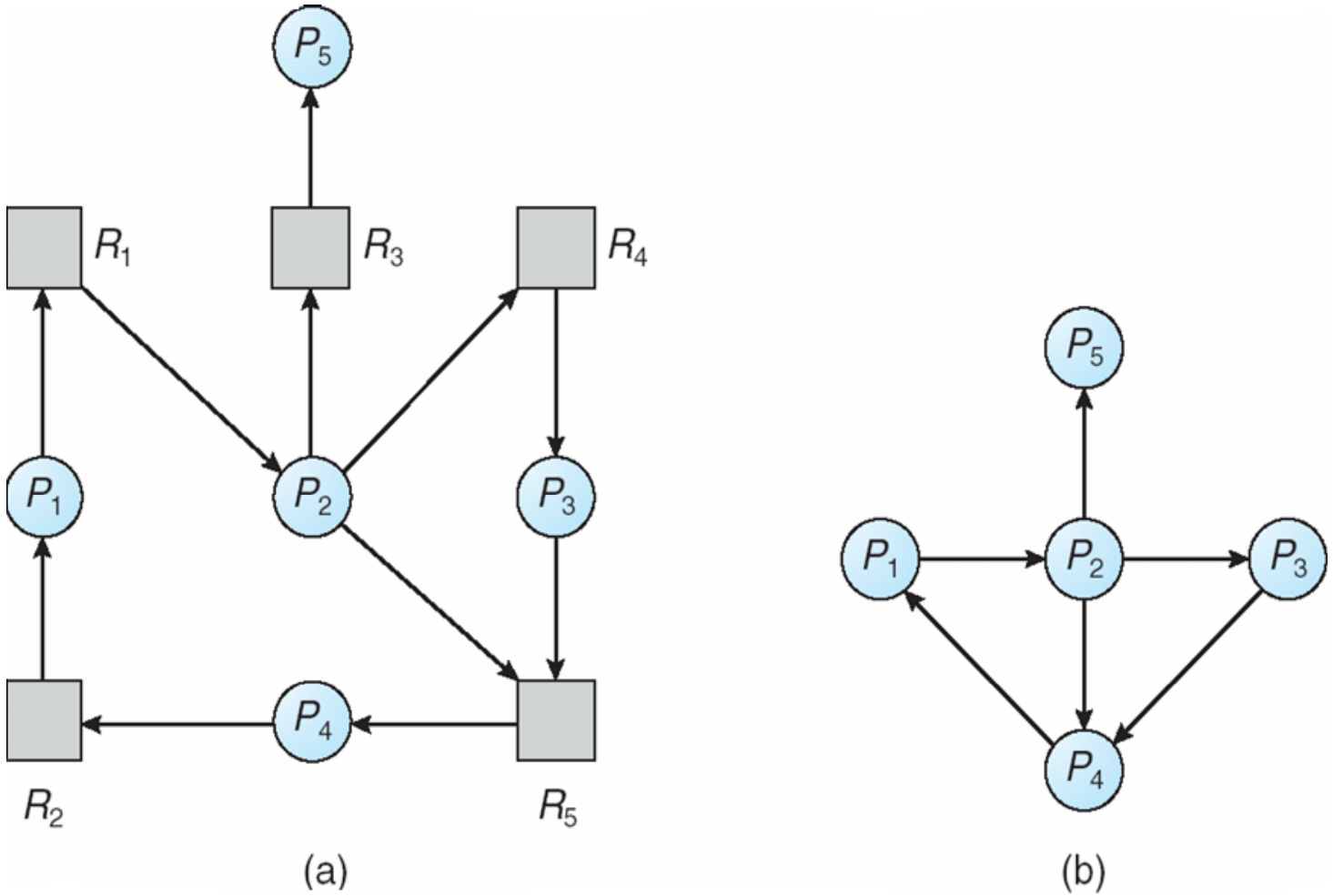
Deadlock Detection

- If the system does NOT employ deadlock prevention or avoidance algorithms:
  - Allow the system to enter deadlock state
  - Apply detection algorithms
  - Apply pre-determined recovery scheme

Single Instance of Each Resource Type

- Maintain *wait-for* graph (variant of resource allocation graph), which removes all resource nodes and collapses appropriate edges
  - 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 $O(n^2)$ operations, where n is the number of vertices in the graph

Resource Allocation Graph/Wait-for Graph



Resource-Allocation Graph Corresponding wait-for graph

Several Instances of a Resource Type

- *wait-for* graphs do NOT work for multiple instances
- 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,* process P_i is requesting *k* more instances of resource type R_j.

Detection Algorithm (1)

How to detect if a request can end up in a deadlocked state?

1. Let *Work* and *Finish* be vectors of length *m* and *n*, respectively Initialize:
   (a) *Work* = *Available*
   (b) For *i* = 1,2, …, *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

Detection Algorithm (2)

3. *Work* = *Work* + *Allocation*$_i$
   Finish[*i*] = *true*
   go to step 2

4. If *Finish*[*i*] == false, for some *i*, 1 δ *i* δ *n*,
   the system is in deadlock state.
   Moreover, if *Finish*[*i*] == *false*, P_i is deadlocked

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

Example of Detection Algorithm (1)

- 5 processes P_0 through P_4
  - 2 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 of Detection Algorithm (2)

- P_2 requests an additional instance of type C

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

- State of the 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 the algorithm would not be able to tell which of the many deadlocked processes “caused” the deadlock!

Recovery: Process Termination

- Abort all deadlocked processes
  - Partial computation of many processes may be lost
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should the abort start?
  - Priority of the process
  - How long process has computed and how much longer is has to completion?
  - Resources the process has used
  - Resources process needs to complete
  - How many processes will be needed to terminate?
  - Is process interactive or batch?

Recovery: Resource Preemption

- Take away resources from other processes until the deadlock is resolved
- Issues to be addressed
  - Selecting a victim and minimize cost
    - *E.g.*, number of held resources, running time, estimated time to completion
  - Rollback: Return to some safe state, restart process for that state
    - Preempted process must be restartable from earlier checkpoint
    - Total restart, if no prior “safe state” identifiable
  - Starvation
    - Same process may always be picked as victim
    - *E.g.*, include number of rollback in cost factor

Stay safe and healthy!

And remember, next Sunday, March 29, 2020
at 2.00am you need to shift to 3.00am (CEST).

You may do so already the night before or
set your alarm on Sunday an hour earlier ☺