# Chapter 7: Deadlocks

Wait for someone who waits for you!

Thanks to the author of the textbook [**SGG**] for providing the base slides. I made several changes/additions.
These slides may incorporate materials kindly provided by Prof. Dakai Zhu.
So I would like to thank him, too.
**Turgay Korkmaz**

# Chapter 7:  Deadlocks

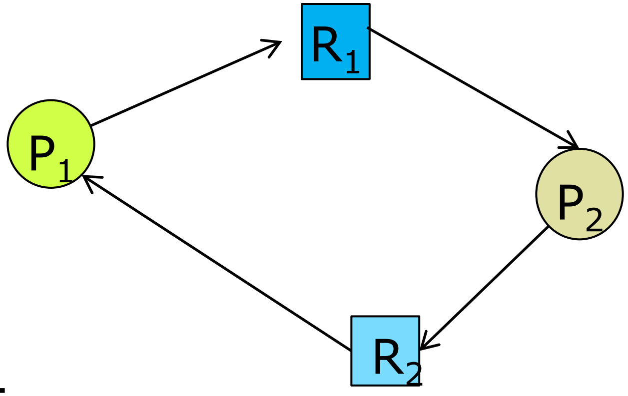
- The Deadlock Problem *
- System Model **
- Deadlock Characterization *****
- Methods for Handling Deadlocks *****
  - Deadlock Prevention ****
  - Deadlock Avoidance ****
  - Deadlock Detection ***
  - Deadlock Recovery **

# Chapter 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

# The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
- Example 1:
  - System has 2 disk drives
  - $P_1$ and $P_2$ each hold one disk drive
  - and each needs another one




- Example 2:
  - Semaphores *A* and *B*, initialized to 1

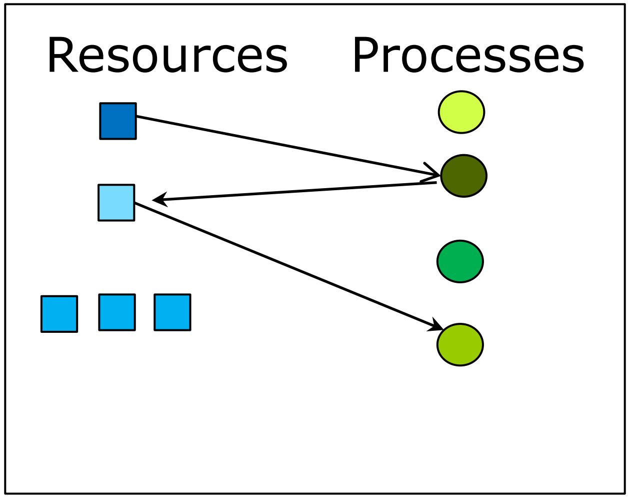
| $P_0$ | $P_1$ |
|---|---|
| wait (A); | wait(B) |
| wait (B); | wait(A) |

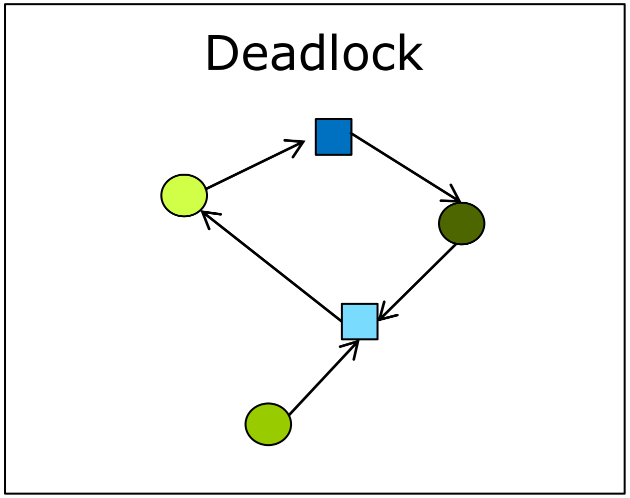
# System Model

- Resource types $R_1, R_2, \ldots, R_m$

  *CPU cycles, memory space, I/O devices*

- Each resource type $R_i$ has $W_i$ instances
- Each process utilizes a resource as follows:
  - **Request**
    - System calls (e.g., open(), allocate())
    - if the requested resource is being used by another process, block/wait until it is released
  - **Use**
  - **Release**
    - System calls (e.g., close(), free())



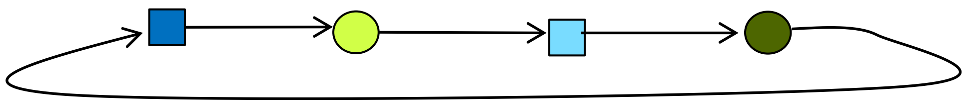



# DEADLOCK CHARACTERIZATION

# Deadlock Characterization

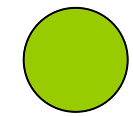
Deadlock can arise if ***four*** conditions hold simultaneously

- **Mutual exclusion:** Only one process can use a resource at a time. Other requesting processes must wait.
- **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 {P0, P1, …, Pn} of waiting processes such that P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, …, Pn–1 is waiting for a resource that is held by Pn, and Pn is waiting for a resource that is held by P0

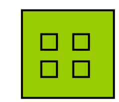


# Resource-Allocation Graph: *G(V, E)*

- 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.



A Process

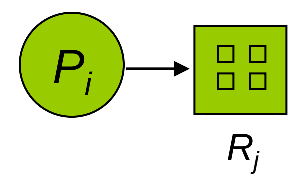


Resource Type with 4 instances

- **Request edge**

  Directed edge $P_i \rightarrow R_j$

  $P_i$ requests instance of $R_j$

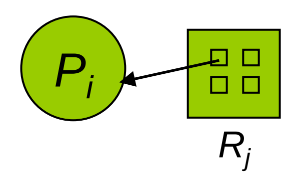



A request edge

- **Assignment edge**

  Directed edge $R_j \rightarrow P_i$

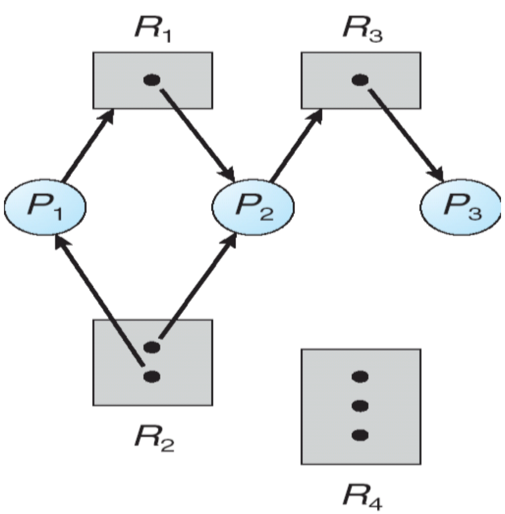
  $P_i$ is holding an instance of $R_j$



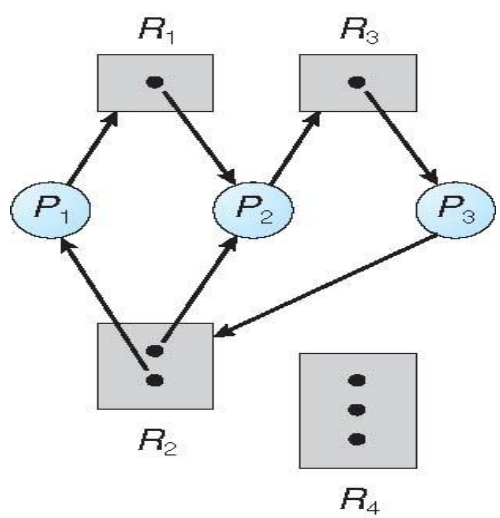

An assignment edge

# Basic Facts

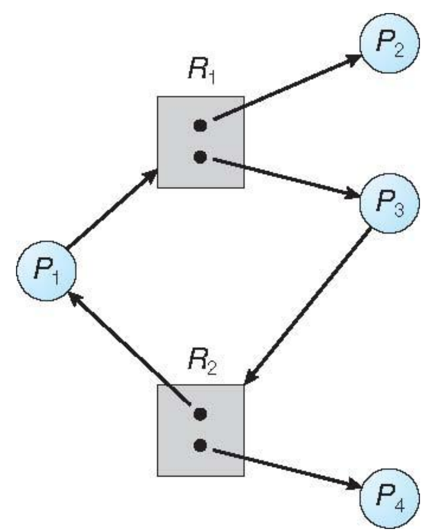
- If graph contains no cycles ⇒ no deadlock
- If graph contains a cycle ⇒ there might be a deadlock
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock




No Deadlock




With A Deadlock




Cycle But No Deadlock

# METHODS FOR HANDLING DEADLOCKS

# Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state:
  - Prevention and
  - Avoidance
- Allow the system to enter a deadlock state but then detect and remove it:
  - Detection and Recovery
  - Ignore the problem and pretend that deadlocks never occur in the system.
    - Used by most operating systems (e.g., UNIX, Java)
    - - performance degradation when there is deadlock
    - - manual intervention is needed (e.g., re-start the system)
    - + easy and cheap

Which method would you select? and why?

# Java Deadlock Example

Thread A

```
class A implements Runnable
{
   private Lock first, second;

   public A(Lock first, Lock second) {
      this.first = first;
      this.second = second;
   }

   public void run() {
      try {
         first.lock();
         // do something
            second.lock();
            // do something else
      }
      finally {
         first.unlock();
         second.unlock();
      }
   }
}
```

Thread B

```
class B implements Runnable
{
   private Lock first, second;

   public A(Lock first, Lock second) {
      this.first = first;
      this.second = second;
   }

   public void run() {
      try {
         second.lock();
         // do something
            first.lock();
            // do something else
      }
      finally {
         second.unlock();
         first.lock();
      }
   }
}
```

```
public static void main(String arg[]) {
   Lock lockX = new ReentrantLock();
   Lock lockY = new ReentrantLock();

   Thread threadA = new Thread(new A(lockX,lockY));
   Thread threadB = new Thread(new B(lockX,lockY));

   threadA.start();
   threadB.start();
}
```

When there will a deadlock?

threadA → (second)lockY → threadB → (first)lockX → threadA

But it might happen if threadA gets both first!

# Handling Deadlocks in Java

```
public class ClockApplet extends Applet implements Runnable
{
    private Thread clockThread;
    private boolean ok = false;
    private Object mutex = new Object();

    public void run() {
       while (true) {
          try {
             // sleep for 1 second
             Thread.sleep(1000);

             // repaint the date and time
             repaint();

             // see if we need to suspend ourself
             synchronized (mutex) {
                while (ok == false)
                   mutex.wait();
             }
          }
          catch (InterruptedException e) { }
       }
    }

    public void start() {
       // Figure 7.7
    }

    public void stop() {
       // Figure 7.7
    }

    public void paint(Graphics g) {
      g.drawString(new java.util.Date().toString(),10,30);
    }
}
```

Up to programmer to write a deadlock-free programs!

```
// this method is called when the applet is
// started or we return to the applet
public void start() {
     ok = true;

     if (clockThread == null) {
        clockThread = new Thread(this);
        clockThread.start();
     }
     else {
        synchronized(mutex) {
           mutex.notify();
        }
     }
}

// this method is called when we
// leave the page the applet is on
public void stop() {
     synchronized(mutex) {
        ok = false;
     }
}
```

Restrain the ways request can be made

# DEADLOCK PREVENTION

# Deadlock Prevention

Make sure at least one of the four conditions cannot hold

- **Mutual Exclusion**
  - Non-sharable resources (e.g., printer) must be allocated exclusively
  - Sharable resources (e.g., read only files) can be shared
- **Hold and Wait**
  - 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
- **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
  - Preempt resources (e.g., cpu) and put process into waiting queue; preempted 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 (e.g., F(tape)=1, F(disk)=5, F(print)=12)

- low device utilization, reduce system throughput, starvation possible

Single instance of a resource type

Use a resource-allocation graph

Multiple instances of a resource type

Use the banker's algorithm

# DEADLOCK AVOIDANCE

# Deadlock Avoidance

- Require additional *a priori* information about how resources are to be requested.
- Suppose the system knows that
  - P will request first the printer then the tape while
  - Q will request first the tape then the printer
- Now the system can avoid deadlock by not allowing P or Q while the other made some allocation!
- Solutions/Algorithms differ in how much additional information to require!

# Deadlock Avoidance cont'd

- Simplest and most useful model requires each process to declare the *maximum number* of resources of each type that it may need.
- When a process requests an available resource, the deadlock-avoidance algorithm dynamically examines the resource-allocation state to **ensure that there can never be a circular-wait condition**
- In other words, the system must decide if immediate allocation leaves the system in a safe state.

# Safe State

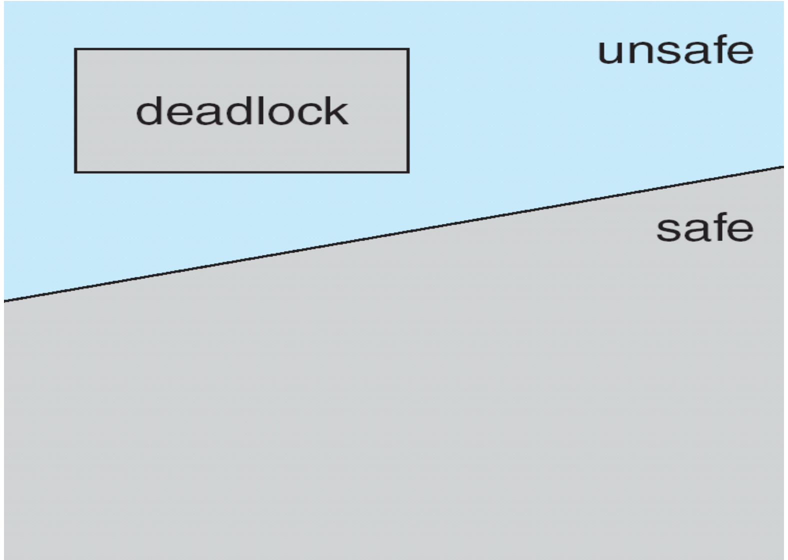
- The system can allocate resources to each process in some order and still avoid deadlock
- System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_{i-1}, \boldsymbol{P_i}, P_{i+1}, \ldots, P_n>$ such that
  - the resources for $P_i$ can be satisfied by currently available resources and resources held by all the $P_j$, with $j < i$
- That is:

*Otherwise, we have* ***unsafe state****.*

  - 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

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



# Example

Max number of tapes is 12.

| | Max need | Currently holding |
|---|---|---|
| P0 | 10 | 5 |
| P1 | 4 | 2 |
| P2 | 9 | 2 |

- Does <P1, P0, P2> satisfy safety condition?
- Suppose P2 gets one more tape,

| | | |
|---|---|---|
| P0 | 10 | 5 |
| P1 | 4 | 2 |
| P2 | 9 | 3 |

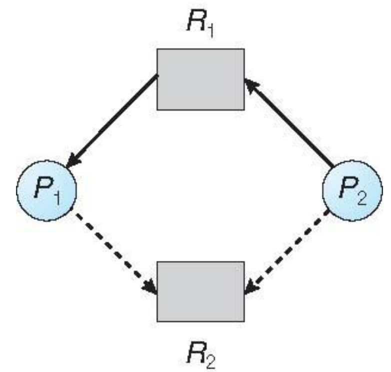
- Are we still in safe state? Why or why not?
- Don't allow P2 get the 3rd tape to avoid deadlock

# Resource-Allocation Graph Scheme

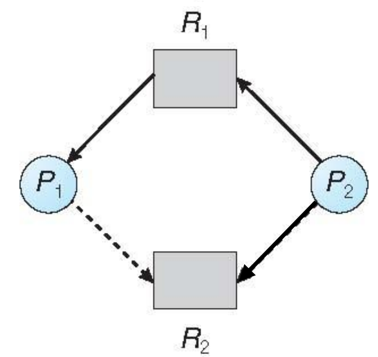
## Single instance of a resource type

- Claim resources *a priori* in the system
- Add **Claim edge** $P_i \rightarrow R_j$
  - Process $P_i$ **may** request resource $R_j$ (represented by a dashed line)

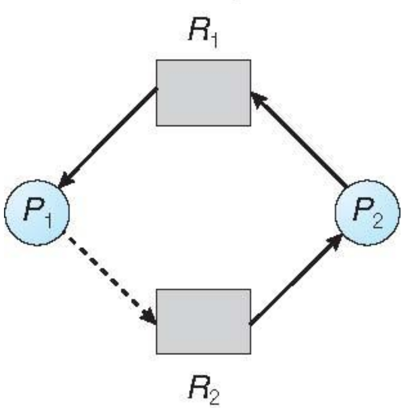
Claim → Request → Assignment edges




- Suppose that process $P_2$ requests the resource $R_2$, can we grant this request?
  - *If we grant it, there will be cycle (unsafe state)*

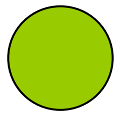


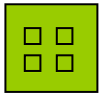

- The request can be granted only if converting the request edge to an assignment edge **does not result in the formation of a cycle**
  - Cycle-detection algorithm (e.g., DFS)
    - Adjacency list O(N + E), or adjacency matrix $O(N^2)$.



# Recall: Resource-Allocation Graph: *G(V, 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.

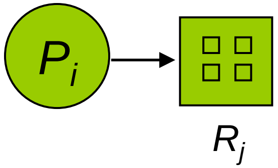

A Process


Resource Type with 4 instances

- **Request edge**

  Directed edge $P_i \rightarrow R_j$

  $P_i$ requests instance of $R_j$

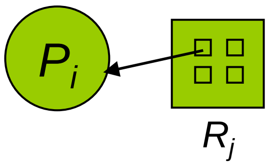


A request edge

- **Assignment edge**

  Directed edge $R_j \rightarrow P_i$

  $P_i$ is holding an instance of $R_j$



An assignment edge

# Banker's Algorithm
## Multiple instances of a resource type

- Each new process must a priori declare maximum number of resources it may use
- The system determines if the system will be safe if all requested resources are allocated
  - If yes, then allocate resources
  - Otherwise, the new process must wait until more resources are released by others
- When a process gets all its resources, it must return them in a finite amount of time

# Data Structures for the Banker's Algorithm

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

- ***Available[j] = k***, *j=0,1, 2, ... m-1*
  - There are $k$ instances of resource type $R_j$ available

*i=0,1,2,..., n-1*
*j=0,1, 2, ... m-1*

- ***Max[i,j] = k***,
  - $P_i$ may request at most $k$ instances of resource type $R_j$
- ***Allocation[i,j] = k***,
  - $P_i$ is currently allocated $k$ instances of $R_j$
- ***Need[i,j] = k***,
  - $P_i$ may need $k$ more instances of $R_j$ to complete its task

  *Need[i,j] = Max[i,j] – Allocation[i,j]*

# Safety Algorithm

## Find out if the system is safe or not

1. Initialize *Work* and *Finish*

   *Work is like tmp Available*

   *Work[j] = Available[j]* for *j=0,1, 2, … m-1*

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

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

   *Finish[i] == false*

   *Need[i,j]≤ Work[j]* for *j=0,1, 2, … m-1*

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

   Complexity
   $O(m \times n^2)$

3. *Work[j] = Work[j] + Allocation[i,j]* for *j=0,1, 2, … m-1*
   *Finish[i] = true*
   go to step 2

4. If *Finish[i] == true* for all *i*, then the system is in a safe state else not

# Resource-Request Algorithm for Process $P_i$

***Determine if the request can be safely granted***

***Request[i,j] = k,*** $P_i$ 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:

   $Available = Available - Request_i;$

   $Allocation_i = Allocation_i + Request_i;$

   $Need_i = Need_i - Request_i;$

   Call the safety algorithm

   - *If safe* $\Rightarrow$ *the resources are allocated to Pi*
   - *If unsafe* $\Rightarrow$ $P_i$ *must wait, and restore the old resource-allocation*

# Example of Banker's Algorithm

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

| | Allocation | Max | Finish | Work= | Available | Max-Allocation Need |
|---|---|---|---|---|---|---|
| | A B C | A B C | | | A B C | A B C |
| $P_0$ | 0 1 0 | 7 5 3 | f t | 10 5 7 | 3 3 2 | 7 4 3 |
| $P_1$ | 2 0 0 | 3 2 2 | f t | 5 3 2 | | 1 2 2 |
| $P_2$ | 3 0 2 | 9 0 2 | f t | 10 4 7 | | 6 0 0 |
| $P_3$ | 2 1 1 | 2 2 2 | f t | 7 4 3 | | 0 1 1 |
| $P_4$ | 0 0 2 | 4 3 3 | f t | 7 4 5 | | 4 3 1 |

- Is the system in a safe state?
- Yes, < $P_1$, $P_3$, $P_4$, $P_2$, $P_0$> satisfies safety criteria

# Example (Cont.): $P_1$ Request (1,0,2)

- Check Request ≤ *Need*$_i$, i.e., (1,0,2) ≤ (1,2,2) ⇒ true
- Check *Request ≤ Available*, i.e. (1,0,2) ≤ (3,3,2) ⇒ true
- Pretend to allocate and update the 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 | |

- Execute safety algorithm
- < $P_1$, $P_3$, $P_4$, $P_0$, $P_2$> satisfies safety requirement

# Example (Cont.): Exercise

- Can request for (3,3,0) by $P_4$ be granted?
  - No because resources are not available
- Can request for (0,2,0) by $P_0$ be granted?
  - No. Actually, resources are available but they cannot be granted since the resulting state would be unsafe!

Allow system to enter deadlock state

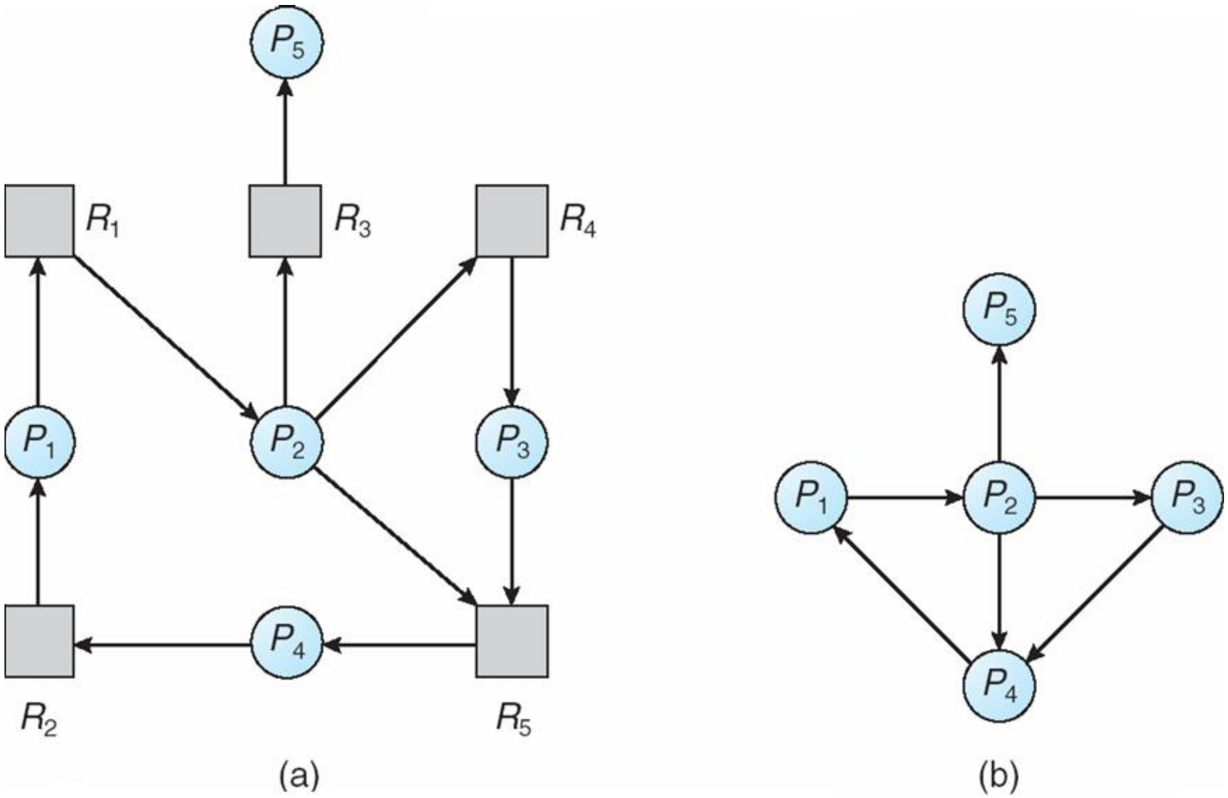
- Detection algorithm (same as safety algorithm)
- Recovery scheme

# DEADLOCK DETECTION

# 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. $R_j$.

# Detection Algorithm

**(recall safety algorithm)**

1. Initialize *Work* and *Finish*:

   *Work[j] = Available[j]* for *j=0,1, 2, … m-1*

   **if** $\boldsymbol{Allocation_i \neq 0}$**,** for *i=0,1, 2, … n-1*

   **then** *Finish[i] = false*

   **else** ***Finish[i] = true***

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

   *Finish[i] == false*

   $\boldsymbol{Request_i} \leq Work$

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

3. $Work = Work + Allocation_i$

   *Finish*[*i*] = *true*

   go to step 2

4. If *Finish[i]* == true for all *i*, then no deadlock; **otherwise, the system is in deadlock state.**

**Algorithm requires an order of O(*m* x *n*$^2$) operations**

If *Finish*[*i*]==*false*, then $P_i$ is deadlocked

# Example of Detection Algorithm

- Five processes $P_0$ through $P_4$;
- Three resource types ***A*** (7 instances), ***B*** (2), ***C*** (6)
- 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 | |

- <$P_0$, $P_2$, $P_3$, $P_1$, $P_4$> will result in *Finish[i]* = true for all *i*, so no deadlock

# Example (Cont.)

- $P_2$ requests an additional instance of type $C$

| | 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 $P_0$, 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 frequently
  - Performance overhead
- If invoked at rather wide intervals,
  - 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

Resource Preemption

# 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 the 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

# End of Chapter 7

None of the basic approaches alone is enough, but they can be combined for different types of resources!