# Lecture 5: Synchronization
# Deadlocks,
# Deadlock Risk Management

# Bounded-Buffer Problem using Semaphores

- Three semaphores
  - mutex – for mutually exclusive access to the buffer – initialized to 1
  - used – counting semaphore indicating item count in buffer – initialized to 0
  - free – number of free items – initialized to BUF_SZ

```
void producer() {
   while (1) { /* Generate new item into nextProduced */
               wait(free);
               wait(mutex);
               buffer[in] =  nextProduced; in = (in + 1) % BUF_SZ;
               signal(mutex);
               signal(used);
   }
}
void consumer() {
   while (1) { wait(used);
               wait(mutex);
               nextConsumed = buffer[out];  out = (out + 1) % BUF_SZ;
               signal(mutex);
               signal(free);
               /* Process the item from nextConsumed */
   }
}
```

# Bounded-Buffer Problem using Condition Variables

- One mutex and two condition variables
  - The_mutex – for mutually exclusive access to the buffer
  - cond_empty – condition variable to block thread if empty buffer
  - cond_free – condition variable to block thread if full buffer

```
void producer() {
  while (1) {      /* Vygeneruj položku do proměnné nextProduced */
      pthread_mutex_lock(&the_mutex);       /* protect buffer */
      while (buffer_size >= BUF_SIZE) {       /* buffer full then wait */
          pthread_cond_wait(&cond_full, &the_mutex);
      }
      buffer [in] = nextProduced; in = (in + 1) % BUF_SZ; buffer_size++;
      pthread_cond_signal(&cond_empty);          /* wake up consumer */
      pthread_mutex_unlock(&the_mutex);
} }
void consumer() {
  while (1) {
      pthread_mutex_lock(&the_mutex);       /* protect buffer */
      while (buffer_use == 0) {                  /*  buffer emptythen wait */
          pthread_cond_wait(&cond_empty, &the_mutex);
      }
      nextConsumed =  buffer[out]; out = (out + 1) % BUF_SZ; bufer_size--;
      pthread_cond_signal(&cond_full);   /* wake up consumer */
      pthread_mutex_unlock(&the_mutex);
} }
```

# Readers and Writers with Readers' Priority

## Shared data

- semaphore wrt, readcountmutex;
- int readcount

## Initialization

- wrt = 1; readcountmutex = 1; readcount = 0;

## Implementation

**Writer:**

```
wait(wrt);
....
    writer modifies data
....
signal(wrt);
```

**Reader:**

```
wait(readcountmutex);
readcount++;
if (readcount==1) wait(wrt);
signal(readcountmutex);

    ... read shared data ...

wait(readcountmutex);
readcount--;
if (readcount==0) signal(wrt);
signal(readcountmutex);
```

# Readers and Writers with Writers' Priority

## Shared data

- semaphore wrt, rdr, readcountmutex, writecountmutex;
  int readcount, writecount;

## Initialization

- wrt = 1; rdr = 1; readcountmutex = 1; writecountmutex = 1;
  readcount = 0; writecount = 0;

## Implementation

**Reader:**

```
wait(rdr);
wait(readcountmutex);
readcount++;
if (readcount == 1) wait(wrt);
signal(readcountmutex);
signal(rdr);

    ... read shared data ...

wait(readcountmutex);
readcount--;
if (readcount == 0) signal(wrt);
signal(readcountmutex);
```

**Writer:**

```
wait(writecountmutex);
writecount++;
if (writecount==1) wait(rdr);
signal(writecountmutex);
wait(wrt);

            ... modify shared data ...

signal(wrt);
wait(writecountmutex);
writecount--;
if (writecount==0) release(rdr);
signal(writecountmutex);
```

# Monitors

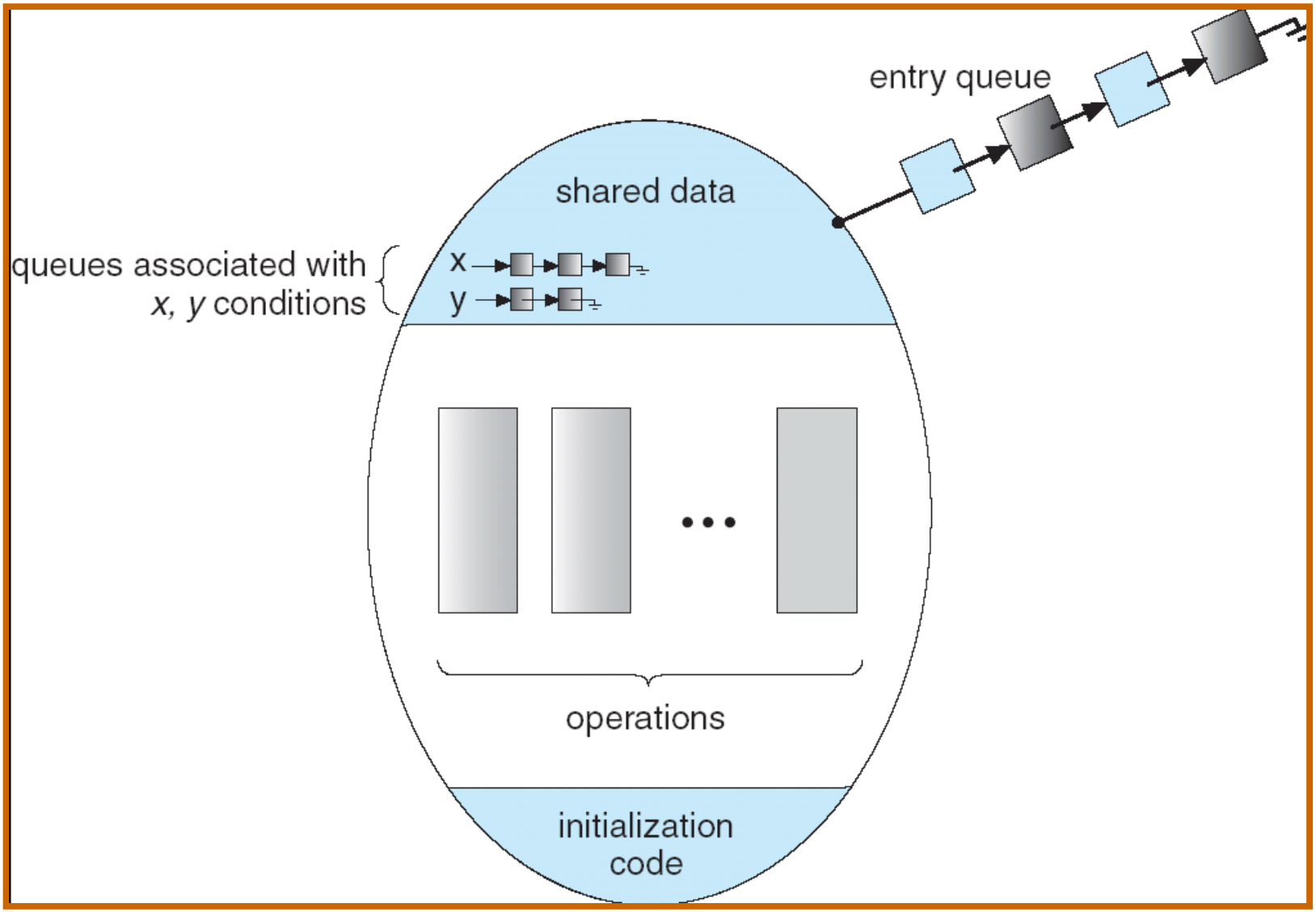
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

```
monitor monitor_name
{
   // shared variable declarations
   condition x, y;  // condition variables declarations
   procedure P1 (…) { …. }
                …
   procedure Pn (…) {……}

   Initialization code ( ….) { … }
                …
   }
}
```

- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended.
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()

# Monitor with Condition Variables



# Semaphores in Java

- Java is using Monitor for synchronization
- User can define counting semaphore as follows:

```
public class CountingSemaphore {
  private int signals = 1;

  public synchronized void wait() throws InterruptedException{
    while(this.signals == 0) wait();
    this.signals--;
  }

  public synchronized void signal() {
    this.signals++;
    this.notify();
  }
}
```

# Spin-lock

- Spin-lock is a general (counting) semaphore using busy waiting instead of blocking
  - Blocking and switching between threads and/or processes may be much more time demanding than the time waste caused by short-time busy waiting
  - One CPU does busy waiting and another CPU executes to clear away the reason for waiting
- Used in multiprocessors to implement short critical sections
  - Typically inside the OS kernel
- Used in many multiprocessor operating systems
  - Windows 2k/XP, Linuxes, ...

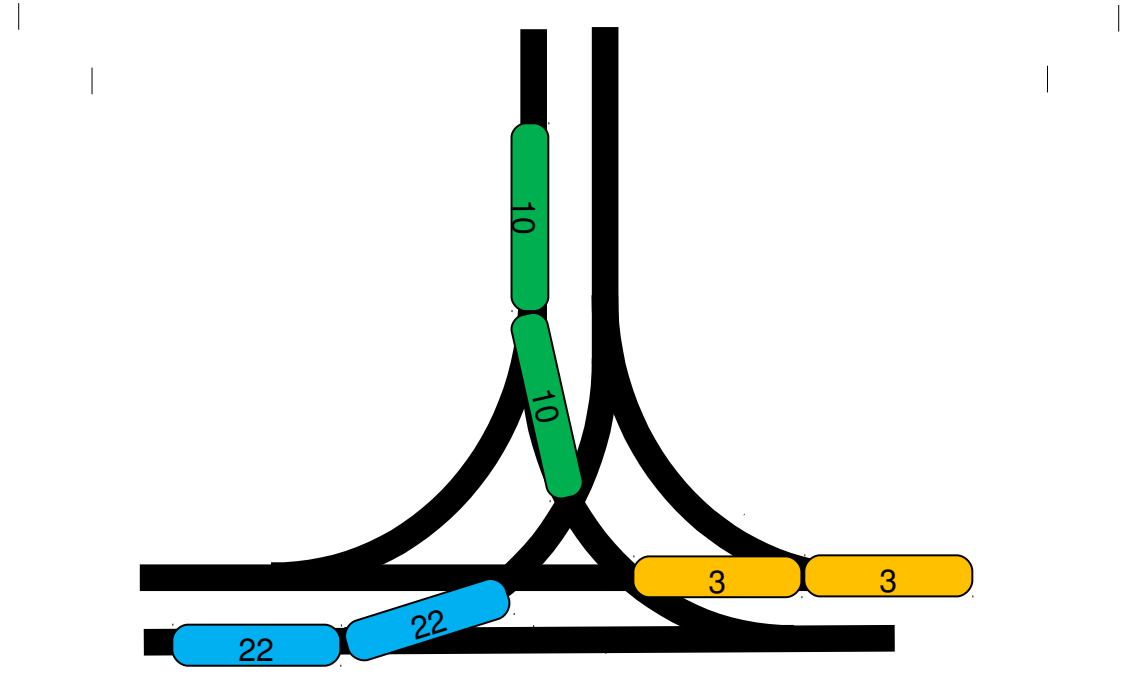
# Synchronization Examples

- Windows XP Synchronization
  - Uses interrupt masks to protect access to global resources on uniprocessor systems
  - Uses spinlocks on multiprocessor systems
  - Also provides dispatcher objects which may act as either mutexes and semaphores
  - Dispatcher objects may also provide events
    - An event acts much like a condition variable
- Linux Synchronization
  - Disables interrupts to implement short critical sections
  - Provides semaphores and spin locks
- Pthreads Synchronization
  - Pthreads API is OS-independent and the detailed implementation depends on the particular OS
  - By POSIX, it provides
    - mutex locks
    - condition variables (monitors)
    - read-write locks (for long critical sections)
    - spin locks

# Deadlock

# Deadlock in real-life

Charles square – no tram can continue, one need to move back
Resources are tracks – for crossing you need to allocate other tracks for your motion




- "It takes money to make money"
- You can't get a job without experience; you can't get experience without a job

# Dining Philosophers Problem

## Shared data

- semaphore chopStick[ ] = new Semaphore[5];

## Initialization

- for(i=0; i<5; i++) chopStick[i] = 1;

## Implementation of philosopher *i*:

```
do  {
      chopStick[i].wait;
      chopStick[(i+1) % 5].wait;
            eating();              //  Now eating
      chopStick[i].signal;
      chopStick[(i+1) % 5].signal;
            thinking();            //  Now thinking
 } while (TRUE) ;
```

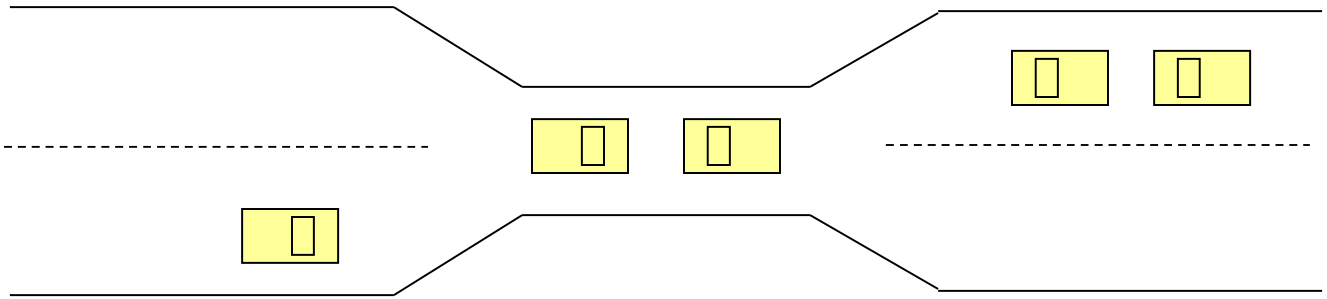
- This solution contains NO deadlock prevention
  - A rigorous avoidance of deadlock for this task is very complicated

# 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
  - System has 2 files.
  - $P_1$ and $P_2$ each holds one file for writing and needs another one.
- Example
  - Semaphores *A* and *B*, initialized to 1 (mutexes)

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

# Bridge Crossing Example



- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.

# 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**
  - **use**
  - **release**

# Deadlock Characterization

Deadlock **can** occur if all **four conditions hold simultaneously**.

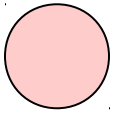
- **Mutual exclusion:** only one process at a time can use a resource.
- **Hold and wait:** a process holding at least one resource is waiting to acquire additional resources held by other processes.
- **No preemption:** a resource can be released only voluntarily by the process holding it, after that process has completed its task.
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_n, P_0\}$ 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 $P_0$.

**NECESSARY** condition! (not sufficient)

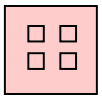
Coffman's conditions [E. G. Coffman, 1971]

# Resource-Allocation Graph

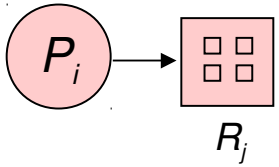
- A set of vertices $V$ and a set of edges $E$
- V is partitioned into two types (bipartite graph):
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- Request edge –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



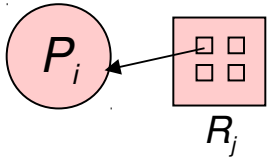
- Resource Type with 4 instances



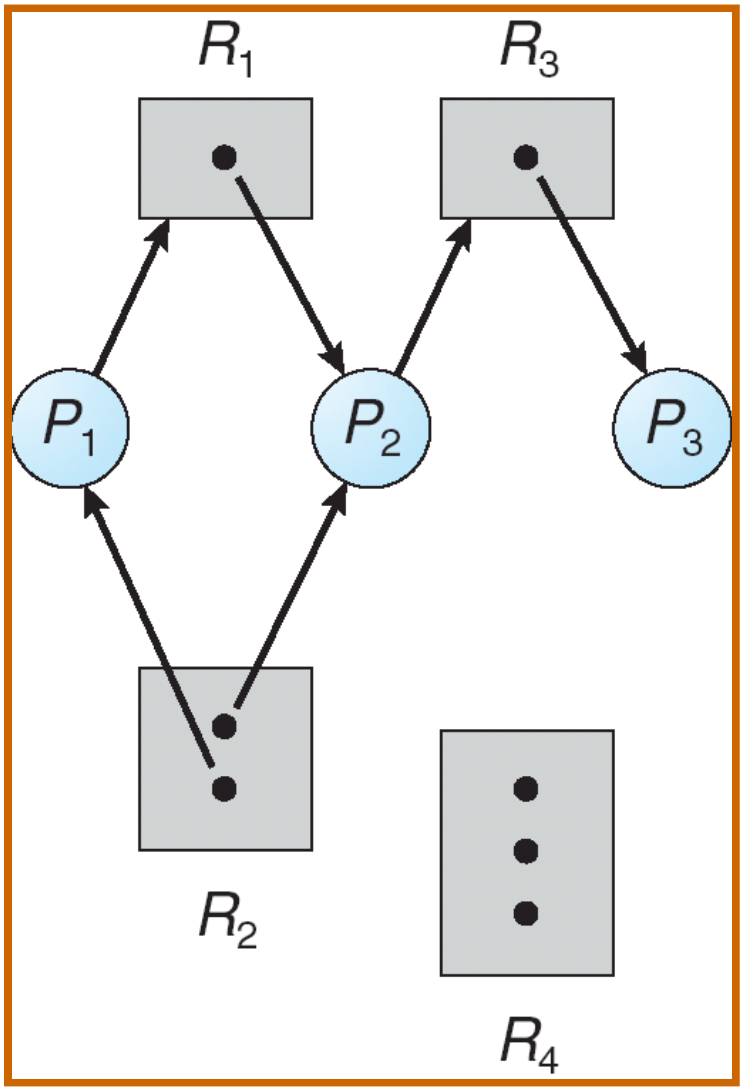
- $P_i$ requests an instance of $R_j$



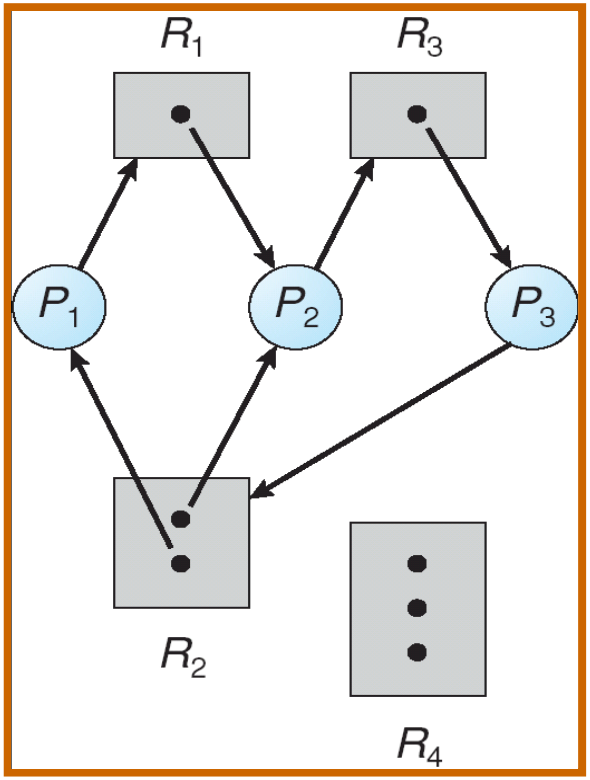

- $P_i$ is holding an instance of $R_j$



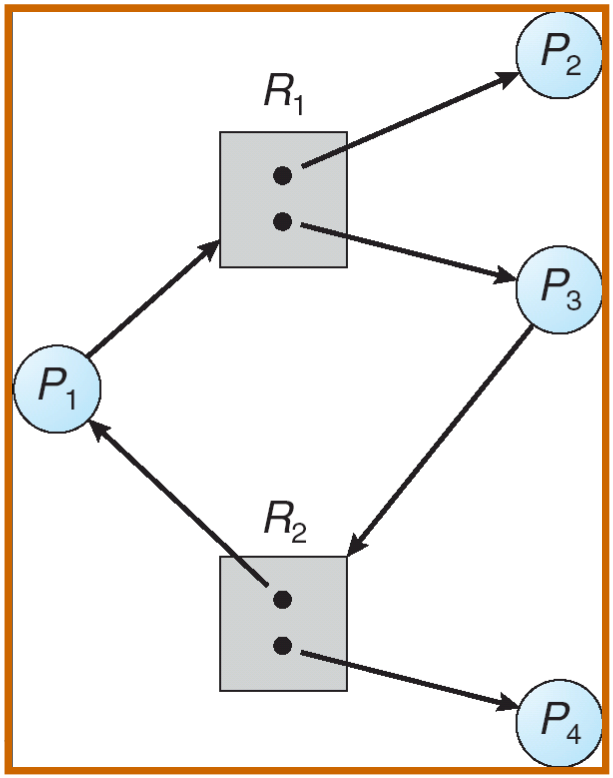
# Example of a Resource Allocation Graph



# Resource Allocation Graph With A Cycle




Deadlock

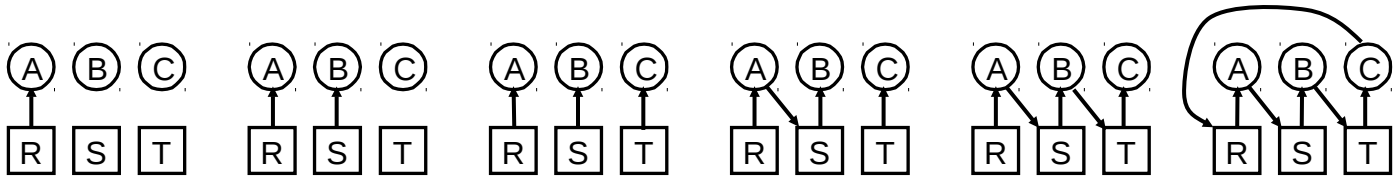



No Deadlock

- Conclusions:
  - 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.

# Can Scheduling Avoid Deadlocks?

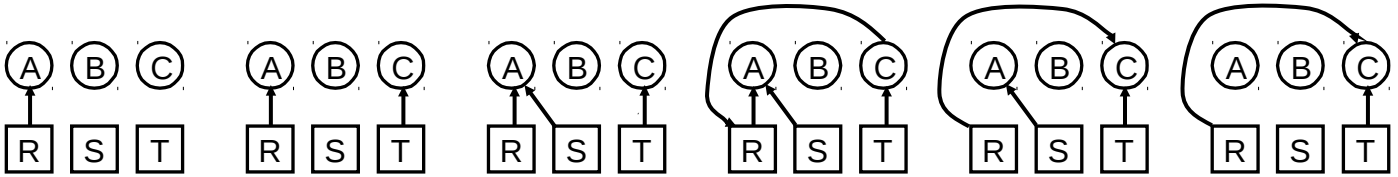
- Consider an example:
  - Processes **A**, **B**, **C** compete for 3 single-instance resources R, S, T




1. *A* requests *R*
2. *C* requests *T*
3. *A* requests *S*
4. *C* requests *R*
5. *A* releases *R*
6. *A* releases *S*

*no deadlock results*

No more problems with B




- Can a careful scheduling avoid deadlocks?
  - What are the conditions?
  - What algorithm to use?

# Approaches to Handling Deadlocks

- **Ostrich approach**: Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.
- **Deadlock Prevention**: Take such precautions that deadlock state is unlikely.
- **Deadlock Avoidance**: Ensure that the system will *never* enter a deadlock state.
- **Detect & Recover:** Allow the system to enter a deadlock state and then recover.

# Deadlock Prevention

Try to break at least one of the Coffman conditions

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources; must hold for nonsharable resources.
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources.
  - 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.
  - 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.

# Deadlock Avoidance

Requires that the system has some additional *a priori* information available.

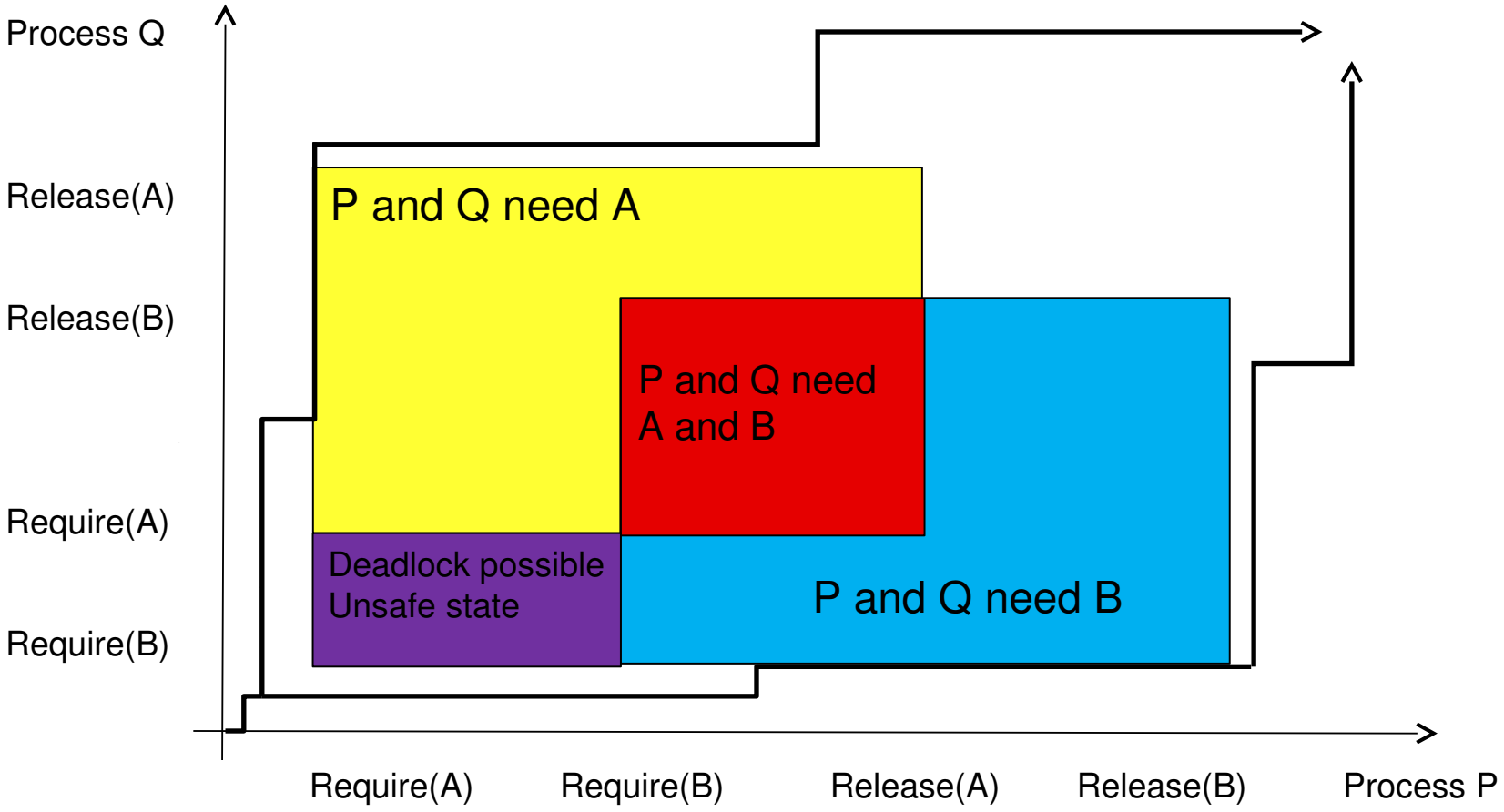
- Simplest and most useful model requires that each process declares the *maximum number* of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.

# Safe State

- 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 safe sequence of all processes.
- **Sequence** of processes $<P_1, P_2, \ldots, P_n>$ is **safe** if for each $P_i$, the resources that $P_i$ can still request can be satisfied by currently available resources + resources held by all the $P_k$, with $k<i$.
  - If $P_i$ resource needs are not immediately available, then $P_i$ can wait until all $P_k$ have finished.
  - When $P_k$ 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.

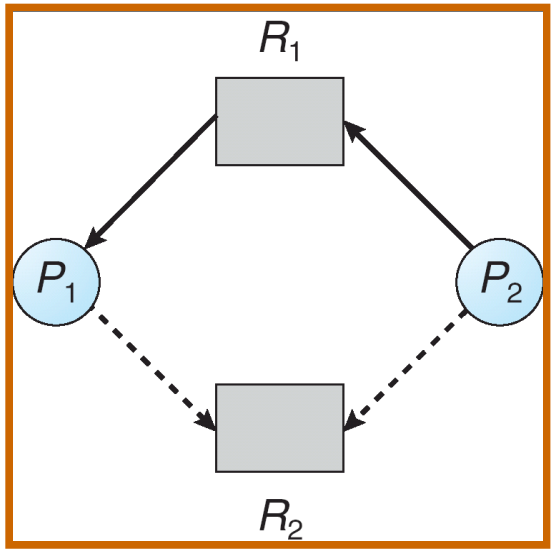
# Basic Facts – System states

- 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.

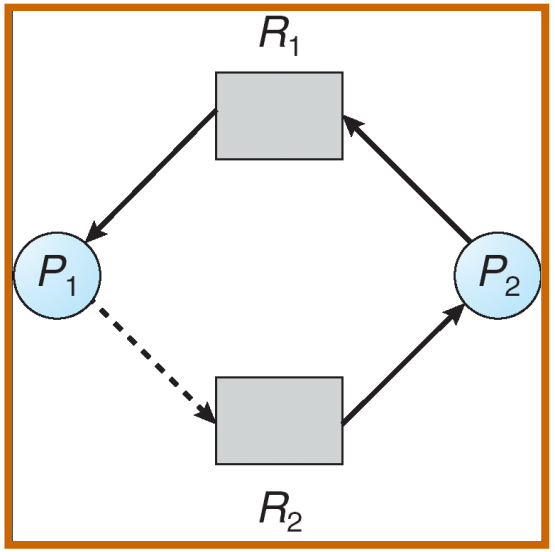


# Resource-Allocation Graph Algorithm

- *Claim edge* $P_i \rightarrow R_j$ indicates that process $P_j$ may request resource $R_j$
  - represented by a dashed line.
- Claim edge changes to a request edge when the process actually requests the resource.
- When a resource is released by a process, assignment edge reconverts to a claim edge.
- Resources must be claimed in the system *a priori*.




Resource-Allocation Graph For Deadlock Avoidance




Unsafe State In Resource-Allocation Graph

# Banker's Algorithm

- Banker's behavior (example of one resource type with many instances):
  - Clients are asking for loans up-to an agreed limit
  - The banker knows that not all clients need their limit simultaneously
  - All clients must achieve their limits at some point of time but not necessarily simultaneously
  - After fulfilling their needs, the clients will pay-back their loans
  - Example:
    - The banker knows that all 4 clients need 22 units together, but he has only total 10 units

| Client | Used | Max. |
|---|---|---|
| Adam | 0 | 6 |
| Eve | 0 | 5 |
| Joe | 0 | 4 |
| Mary | 0 | 7 |

| Client | Used | Max. |
|---|---|---|
| Adam | 1 | 6 |
| Eve | 1 | 5 |
| Joe | 2 | 4 |
| Mary | 4 | 7 |

| Client |
|---|
| Adam |
| Eve |
| Joe |
| Mary |

# Banker's Algorithm (cont.)

- Always keep so many resources that satisfy the needs of at least one client
- 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

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.
- *Max:* $n$ x $m$ matrix. If *Max* [$i,j$] = $k$, then process $P_i$ may request at most $k$ instances of resource type $R_j$.
- *Allocation:* $n$ x $m$ matrix. If Allocation[$i,j$] = $k$ then $P_i$ is currently allocated $k$ instances of $R_j$.
- *Need:* $n$ x $m$ 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

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

   *Work* = *Available*

   *Finish* [*i*] = *false* for *i* = 1,2, …, *n.*

2. Find and *i* such that both:

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

   (b) $Need_i \leq Work$

   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 the system is in a safe state.

# Resource-Request Algorithm for Process $P_i$

$Request_i$ = request vector for process $P_i$.
$Request_i[\ j\ ] == k$ means that 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. Test 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 ⇒ the resources are allocated to Pi.*
- *If unsafe ⇒ Pi must wait, and the old resource-allocation state is restored*

# Example of Banker's Algorithm

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

| | Allocation | Max | Need | Total |
|---|---|---|---|---|
| | A B C | A B C | A B C | A B C |
| $P_0$ | 0 1 0 | 7 5 3 | 7 4 3 | 10 5 7 |
| $P_1$ | 2 0 0 | 3 2 2 | 1 2 2 | *Allocated* |
| $P_2$ | 3 0 2 | 9 0 2 | 6 0 0 | 7 2 5 |
| $P_3$ | 2 1 1 | 2 2 2 | 0 1 1 | *Available* |
| $P_4$ | 0 0 2 | 4 3 3 | 4 3 1 | 3 3 2 |

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

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

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

| | *Allocation* | *Max* | *Need* | *Available* |
|---|---|---|---|---|
| | *A B C* | *A B C* | *A B C* | *A B C* |
| $P_0$ | 0 1 0 | 7 5 3 | 7 4 3 | **2 3 0** |
| $P_1$ | 3 0 2 | 3 2 2 | **0 2 0** | |
| $P_2$ | 3 0 2 | 9 0 2 | 6 0 0 | |
| $P_3$ | 2 1 1 | 2 2 2 | 0 1 1 | |
| $P_4$ | 0 0 2 | 4 3 3 | 4 3 1 | |

- 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 P4 be granted?
- Can request for (0,2,0) by P0 be granted?

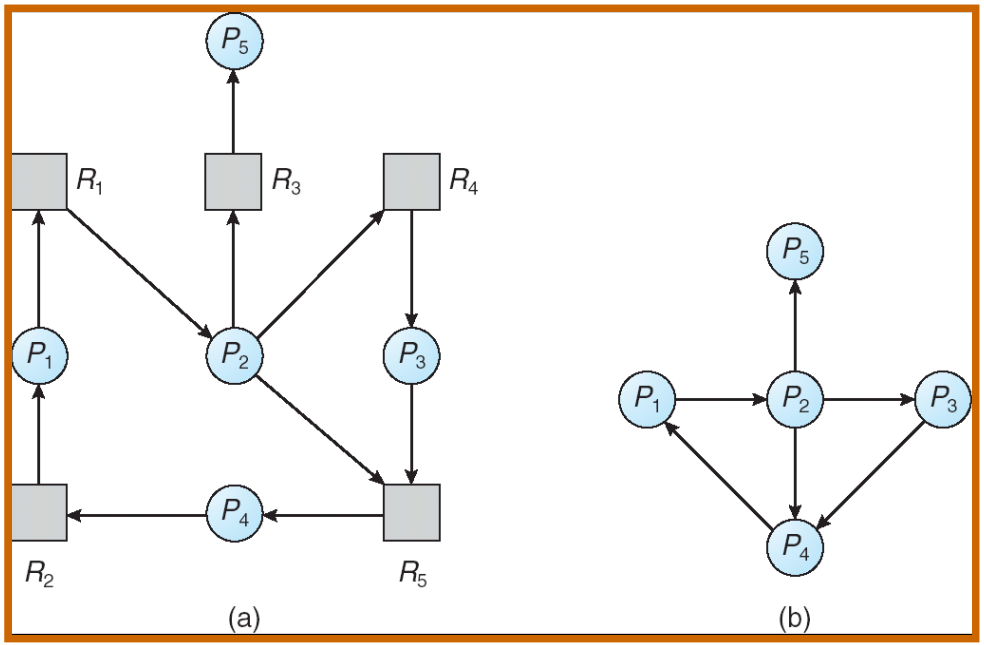
# Deadlock Detection

- 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.
- 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




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 $Allocation_i \neq 0$, 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.
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

**The 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* | *Total* |
|---|---|---|---|
| | *A B C* | *A B C* | *A B C* |
| $P_0$ | 0 1 0 | 0 0 0 | 7 2 6 |
| $P_1$ | 2 0 0 | 2 0 2 | *Allocated* |
| $P_2$ | 3 0 3 | 0 0 0 | 7 2 6 |
| $P_3$ | 2 1 1 | 1 0 0 | *Available* |
| $P_4$ | 0 0 2 | 0 0 2 | **0 0 0** |

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

# Example (Cont.)

- $P_2$ requests an additional instance of type *C*. The *Request* matrix changes

| | *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 system?
  - System would now get deadlocked
  - 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

- Abort all deadlocked processes
  - Very expensive
- 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.

End of Lecture 6

# Questions?