Process Synchronization

- Concurrency and Race Conditions
- Mutual exclusion requirements
- Software and hardware solutions
- Semaphores
- Monitors
- Classical IPC problems and solutions
- Deadlock
  - Characterization
  - Detection
  - Recovery
  - Avoidance and Prevention

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffers. We can do so by having an integer count that keeps track of the number of full buffers. Initially, count is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true)
 {
/* produce an item and put in nextProduced*/
    while (count == BUFFER_SIZE)
    ; // do nothing
          buffer [in] = nextProduced;
          in = (in + 1) % BUFFER_SIZE;
          count++;
   }
```

Consumer

```
while (1)
  {
       while (count == 0)
            ; // do nothing
       nextConsumed =  buffer[out];
       out = (out + 1) % BUFFER_SIZE;
       count--;
       /*  consume the item in nextConsumed*/
}
```

Race Condition

- **Race condition**: The situation where several processes access – and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.
- To prevent race conditions, concurrent processes must be **synchronized**.

Race Condition

- count++ could be implemented as

  register1 = count

  register1 = register1 + 1

  count = register1

- count-- could be implemented as

  register2 = count

  register2 = register2 - 1

  count = register2

Race Condition

- If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.
- Interleaving depends upon how the producer and consumer processes are scheduled.

Race Condition

- Consider this execution interleaving with "count = 5" initially:

  S0: producer execute register1 = count {register1 = 5}

  S1: producer execute register1 = register1 + 1 {register1 = 6}

  S2: consumer execute register2 = count {register2 = 5}

  S3: consumer execute register2 = register2 - 1 {register2 = 4}

  S4: producer execute count = register1 {count = 6 }

  S5: consumer execute count = register2 {count = 4}

- The value of **count** may be either 4 or 6, where the correct result should be 5.
- Reverse of s4 & S5 may give counter=6 .

The Critical-Section Problem

- ***n*** processes all competing to use some shared data. **{P_0, P_1, …, P_{n-1} }**
- Each process has a code segment, called ***critical section***, in which the shared data is accessed**.** (changing common variables, write file, update table etc.)
- **Problem** – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section. (mutually exclusive)

Solution to Critical-Section Problem:

must satisfy 3 requirements

1. Mutual Exclusion
2. Progress
3. Bounded Waiting

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections

2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

Solution to Critical-Section Problem:

must satisfy 3 requirements

3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

- Assume that each process executes at a nonzero speed
- No assumption concerning relative speed of the N processes

Initial Attempts to Solve Problem

- Only 2 processes, P_0 and P_1
- General structure of process P_i (other process P_j)

  do {

  entry section

  critical section

  exit section

  reminder section

  } **while (1)**;

- Processes may share some common variables to synchronize their actions.

Algorithm 1

- Shared variables:
  - **int turn**;
    initially **turn = 0**
  - **turn - i** $\Rightarrow$ P_i can enter its critical section
- Process P_i

```
do {
        while (turn != i) ;
                critical section
        turn = j;
                reminder section
} while (1);
```

- Satisfies mutual exclusion, but not progress

Algorithm 2

- Shared variables
  - **boolean flag[2]**;

    initially **flag [0] = flag [1] = false.**

  - **flag [i] = true** $\Rightarrow P_i$ ready to enter its critical section
- Process P_i

```
do {
        flag[i] := true;
        while (flag[j]) ;                    critical section
        flag [i] = false;
            remainder section
} while (1);
```

- Satisfies mutual exclusion, but not progress requirement.

Algorithm 3 (Peterson's Solution)

- Combined shared variables of algorithms 1 and 2.
- Process P_i

```
do {
        flag [i]:= true;
        turn = j;
        while (flag [j] and turn = j) ;
                critical section
        flag [i] = false;
                remainder section
} while (1);
```

- Meets all three requirements; solves the critical-section problem for two processes.

Synchronization Hardware : for Mutual Exclusion

- Many systems provide hardware support for critical section code. This makes programming task easier and improve system efficiently.
- Interrupt Disabling: Uniprocessors could disable interrupts (using timers)
  - Currently running code would execute without execute without preemption .
  - Generally too insufficient on multiprocessor system
  - OS using this not broadly scalable (time consuming)
- The first solution is to disable interrupt while a process is modifying a shared variable.

Repeat

<disable interrupt>

<critical section>

<enable interrupt>

<remainder>

forever

Synchronization Hardware : for Mutual Exclusion

- **Special machine instructions**
- Modern machine provide special atomic (non-interruptable) hardware instructions.
- Either test memory word & set value. (Test and set instructions).
- Or swap contents of memory words(Swap).
- If two atomic instructions are executed simultaneously(each one of different CPU), they will be executed sequentially in some arbitrary order.
- Unfortunatelly for H/W designers, implementation of these atomic instructions in multiprocessor environment is hard.

Mutual Exclusion with Test-and-Set

- critical section using lock.

```
do
{
    acquire lock;
        critical section
    release lock;
        remainder section
}while(TRUE);
```

```
boolean TestAndSet (boolean *lock)
{
    boolean rv = *lock
    *lock = TRUE;
     return rv;
}
```

Mutual Exclusion with Test-and-Set

- Shared data:
  boolean lock = false;

- Process P_i

```
do {
        while (TestAndSet(lock)) ;
            // critical section
        lock = false;
// remainder section
      }
```

Process can enter critical section only when **lock = FALSE**, if it is Test & Set lock TRUE to acquire the lock

Synchronization Hardware

- Atomically swap two variables.

```
void Swap(boolean &a, boolean &b)
    {
        boolean temp = a;
        a = b;
        b = temp;
    }
```

The instruction exchanges contents of a register with that of a memory location

Synchronization Hardware : for Mutual Exclusion

- Shared data (initialized to **false**):

  boolean lock;

- Each process has boolean variable key; initialized to true.

- **Process P_i**

```
do {
        key = true;
        while (key = = true)
   Swap(lock,key);
            critical section
    Swap(lock,key);
            remainder section
    }
```

"process can enter critical section only if lock==false & key == true."

"lock==true and key ==false means lock is acquired."

Mutual Exclusion Machine Instructions

- **Advantages**
  - Applicable to any number of processes on either a single processor or multiple processors sharing main memory
  - It is simple and therefore easy to verify
  - It can be used to support multiple critical sections

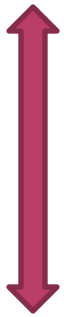
Mutual Exclusion Machine Instructions

- **Disadvantages**
  - Busy-waiting consumes processor time
  - Starvation is possible when a process leaves a critical section and more than one process is waiting.
  - Deadlock
    - If a low priority process has the critical region and a higher priority process needs, the higher priority process will obtain the processor to wait for the critical region

Semaphores

- Special variable called a semaphore is used for signaling
- A process is suspended until, it has received a specific signal.
- Semaphore is a variable that has an integer value
  - May be initialized to a nonnegative number
  - Wait operation decrements the semaphore value
  - Signal operation increments semaphore value
- Two standard operations modify S: wait() and signal()
- can only be accessed via two indivisible (atomic) operations

```
wait (S)
{
    while S≤ 0
        do no-op;
    S--;
}
```



```
signal (S):
{
    S++;
}
```

Semaphore : Critical Section of *n* Processes

- Shared data:

  semaphore mutex; //initially *mutex* = 1

- Process *Pi:*

```
do {
    wait(mutex);
        critical section
    signal(mutex);
        remainder section
} while (1);
```

Semaphore : as a General Synchronization Tool

- Execute B in P_j only after A executed in P_i
- Use semaphore *flag* initialized to 0, shared by both P_i and P_i
- Code:

P_i P_j

□ □

A *wait*(*flag*)

signal(*flag*) B

Semaphore Implementation

- Disadvantage of other mutual exclusion solutions is busy waiting.
- One process is executing in critical section, others are waiting in loop, such semaphores are called as spinlock.
- To overcome busy waiting: wait( ) and signal( ) are modified.

1. When process executes & finds semaphore value is not +ve , it must wait. Instead of waiting process can block itself. block( ) operation places process from busy to waiting state(blocked).
2. Then control is transferred to CPU scheduler, which selects another process to execute.
3. Process blocked should be restarted when some other process executes a signal( ) operation. The process is restarted by wakeup( ) operation.

Semaphore Implementation

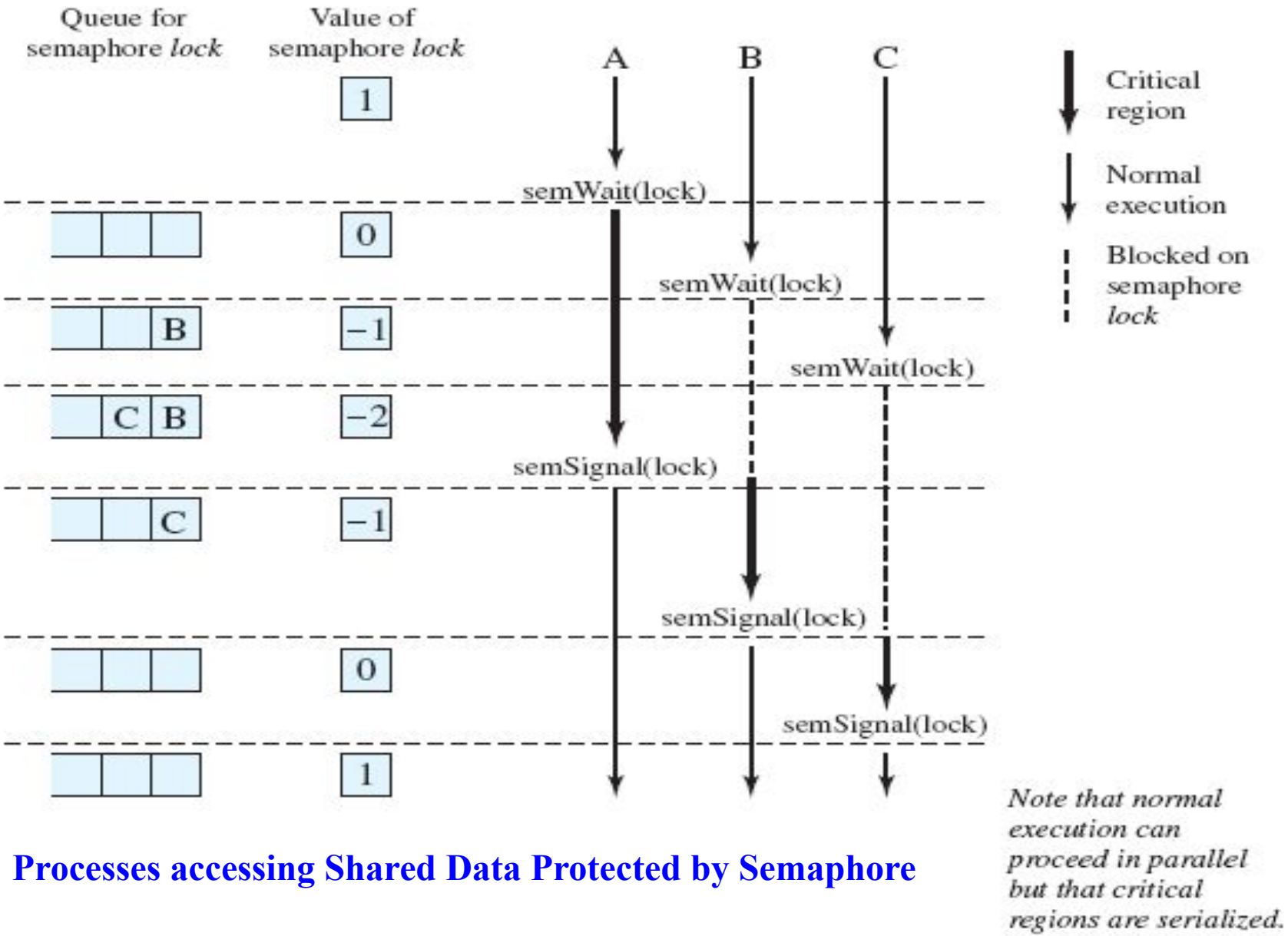
- Each semaphore is associated with waiting queue.
- No two processes can execute wait( ) and signal on the same semaphore at the same time.

- **To avoid spinlock, define semaphore as a record.**

```
typedef struct
{
   int value;
   struct process *L;
}semaphore;
```

If semaphore value is –ve, its magnitude=no of processes in waiting queue.

Semaphore Implementation



Processes accessing Shared Data Protected by Semaphore

Semaphore Implementation

- Semaphore operations now defined as:

```
void wait(S)
    {
        S.value--;
          if (S.value < 0)
            {
               add this process to  S.L;
        block;
            }
     }
```

```
void signal(S)
    {
        S.value++;
        if (S.value <= 0)
          {
      remove a process P from S.L;
      wakeup(P);
          }
      }
```

Semaphore as a General Synchronization Tool

- Execute B in P_j only after A executed in P_i
- Use semaphore *flag* initialized to 0
- Code:

P_i	P_j
A	*wait*(*flag*)
signal(*flag*)	B

Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.
- Let S and Q be two semaphores initialized to 1

  P_0 P_1

  $wait(S);$ $wait(Q);$

  $wait(Q);$ $wait(S);$

  □ □

  $signal(S);$ $signal(Q);$

  $signal(Q)$ $signal(S);$

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

Two Types of Semaphores

- ***Counting* semaphore** – integer value can range over an unrestricted domain.
- ***Binary* semaphore** – integer value can range only between 0 and 1; can be simpler to implement.
- Can implement a counting semaphore *S* as a binary semaphore.

Binary Semaphores

```
Srtuct  binary_semaphore
{
   enum{zero, one} value;
   struct process *L;
};

void wait (binary_semaphore s)
{
  if (s.value = = 1)
       s.value = 0;
  else
        {
            plcae this process to S.L.
             block( );
         }
}
```

```
void signal(binary_semaphore s)
{
   if (s.queue is empty ( ) )
        s.value = 1;
    else
    {
       remove a process fro s.queue;
       wakeup(p);
     }
}
```

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

- Shared data

  semaphore full, empty, mutex;

  Initially:

  full = 0, empty = n, mutex = 1

- Assume buffer consists of *n* buffers, each can hold one item.
- Semaphore mutex provides Mutual Exclusion for access to the buffer pool.
- Semaphore full counts number of full buffers.
- Semaphore empty counts number of empty buffers.

Bounded-Buffer Problem

Producer Process

```
do {
        …
        produce an item in nextp
        …
        wait(empty);

        wait(mutex);

        …
        add nextp to buffer
        …
        signal(mutex);

        signal(full);

    } while (1);
```

Consumer Process

```
do {
        wait(full)

        wait(mutex);

        …
        remove an item from buffer to nextc
        …
        signal(mutex);

        signal(empty);

        …
        consume the item in nextc
        …
    } while (1);
```

Readers-Writers Problem

- Two types of Processes:
- **Writers:** modify a shared object.
- **Readers:** they just read. They do not modify shared object.

- Any number of readers may simultaneously read the file
- Only one writer at a time may write to the file
- If a writer is writing to the file, no reader may read it or no writer may write it.
- Reader-writer problem has several variations, involving priorities:

1. Reader have priority: No reader will be waiting if no writer writing.(writer may suffer starvation.)
2. Writer have priority: if writer is writing no process can read. (reader's starvation.)

Readers-Writers Problem

- Shared data

  semaphore mutex, wrt;

  Initially

  mutex = 1, wrt = 1, readcount = 0

How many processes currently reading object.

Readers-Writers Problem

- **wrt :**

1. Mutual exclusion for writers.
2. The semaphore wrt is common to both reader and writer processes.
3. Used by first & last reader that enter or exits critical section.
4. It is not used by readers who enter or exit while other readers are in their critical section.

- If writer is in critical section and n readers are waiting then one reader is queued on wrt and n-1 readers queued on mutex.
- When writer executes signal(wrt). We can resume execution of waiting reader or writer but decision is made by scheduler.
- mutex semaphore is used to ensure mutual exclusion when variable readcount is updated.

Readers-Writers Problem

mutex = 1

wrt = 1

readcount = 0

- **The structure of a writer process**

```
do {
        wait (wrt) ;
            //    writing is performed
        signal (wrt) ;
} while (TRUE);
```

Allow only one writer at a time to write

Readers-Writers Problem

- The structure of a reader process

```
do {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)
                wait (wrt) ;
        signal (mutex) ;

                // reading

        wait (mutex) ;
        readcount  - - ;
        if (readcount  == 0)
signal (wrt) ;
        signal (mutex) ;

} while (TRUE);
```

Proceed only if no writer is writing; disallow writers once we proceed

Signal a writer only when there are no more active readers

mutex = 1

wrt = 1

readcount = 0

The Readers/Writers Problem

- Multiple readers (customers) want to read the database — access schedules, flight availability, etc.
- Multiple writers (the airline) want to write the database — update the schedules
- It is acceptable to have multiple readers at the same time
- But while a writer is writing to (updating) the database, no one else can access the database — either to read or to write
- Two versions:
- Readers have priority over writers
- Writers have priority over readers
- In most solutions, the non-priority group can starve

(Using Semaphores)

Reader:

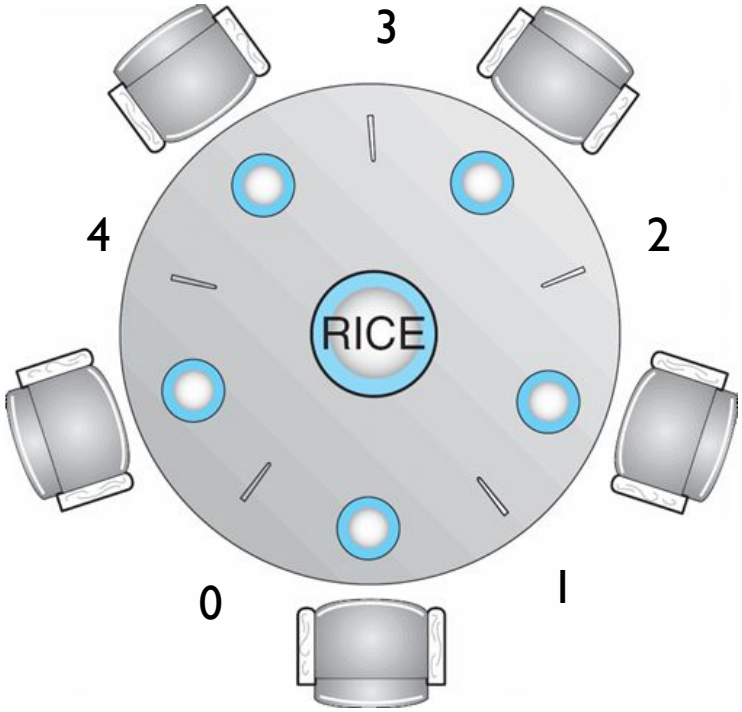
- Needs mutually exclusive access while manipulating "readers" variable
- Does not need mutually exclusive access while reading database
- If this reader is the first reader, it has to wait if there is an active writer (which has exclusive access to the database)
- First reader did a "Wait(write)"
- If other readers come along while the first one is waiting, they wait at the "Wait(mutex)"
- If other readers come along while the first one is actively reading the database, they can also get in to read the database
- When the last reader finishes, if there are waiting writers, it must wake one up

(Using Semaphores)

Writer:

- If there is an active writer, this writer has to wait (the active writer has exclusive access to database)
- If there are active readers, this writer has to wait (readers have priority)
- First reader did a "Wait(write)"
- The writer only gets in to write to the database when there are no other active readers or writers
- When the writer finishes, it wakes up someone (either a reader or a writer —it's up to the CPU scheduler)
- If a reader gets to go next, then once it goes through the "V(mutex)" and starts reading the database, then all other readers waiting at the top "P(mutex") get to get in and read the database as well

Dining-Philosophers Problem



- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem (Cont.)

- The structure of Philosopher i:

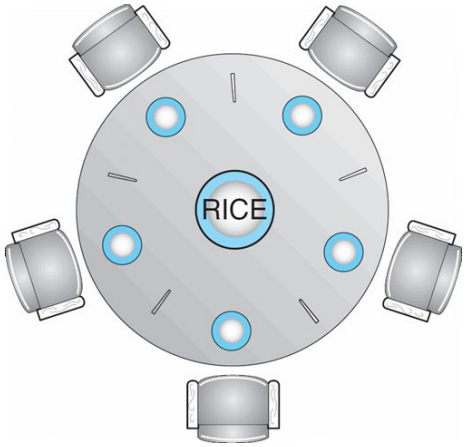
```
do {
        wait ( chopstick[i] );
        wait ( chopStick[ (i + 1) % 5] );

            //  eat

        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );

            //  think

} while (TRUE);
```



Deadlock?
Starvation?

What happens if all pick their left chopsticks?

Problems with semaphore

- Correct use of semaphore operations:
  - signal (mutex) …. wait (mutex)
    - Can violate mutual exclusion
  - wait (mutex) … wait (mutex)
    - Can lead to deadlock!
  - Omitting of wait (mutex) or signal (mutex)
    - either suffer mutual exclusion or deadlock.

Monitors

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

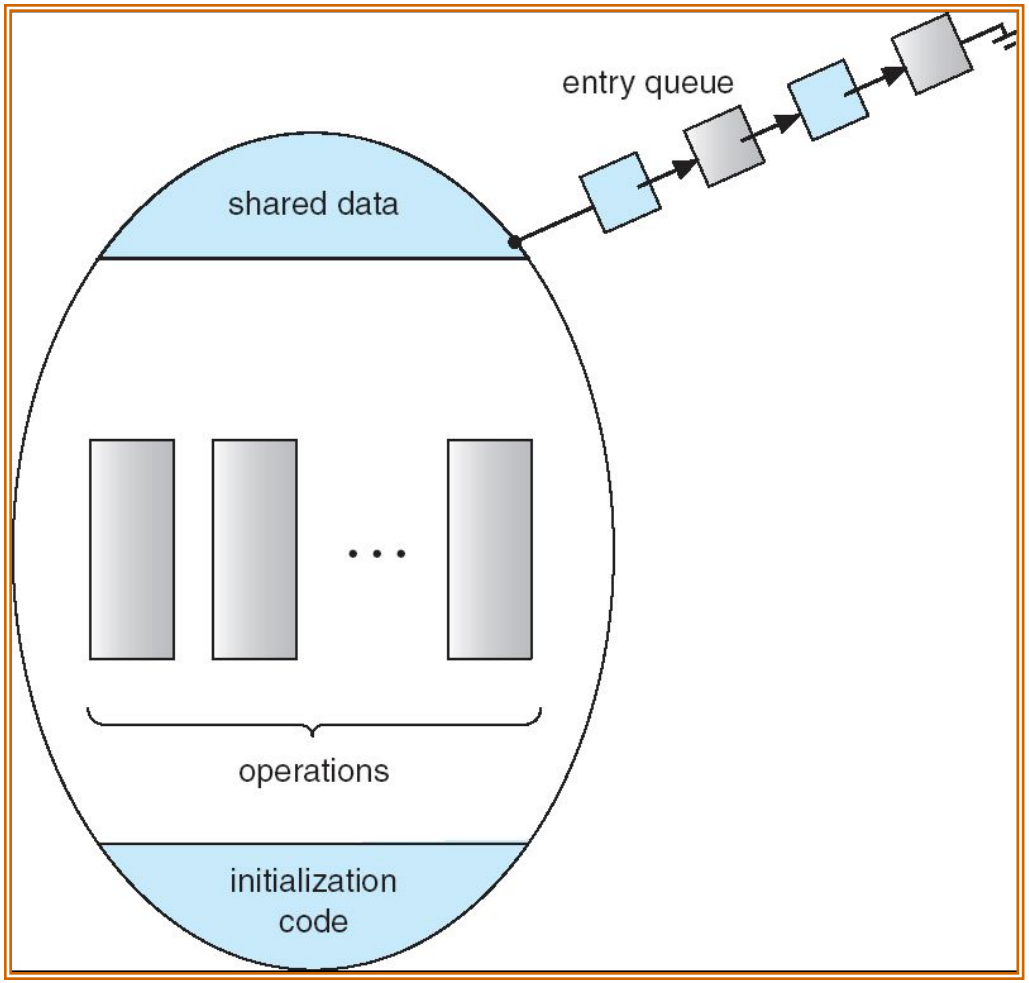
```
monitor monitor-name
{
   // shared variable declarations
   procedure P1 (…) { …. }
   …

   procedure Pn (…) {……}

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

Procedure defined within a monitor can access only those variables declared locally within monitor.

Schematic view of a Monitor

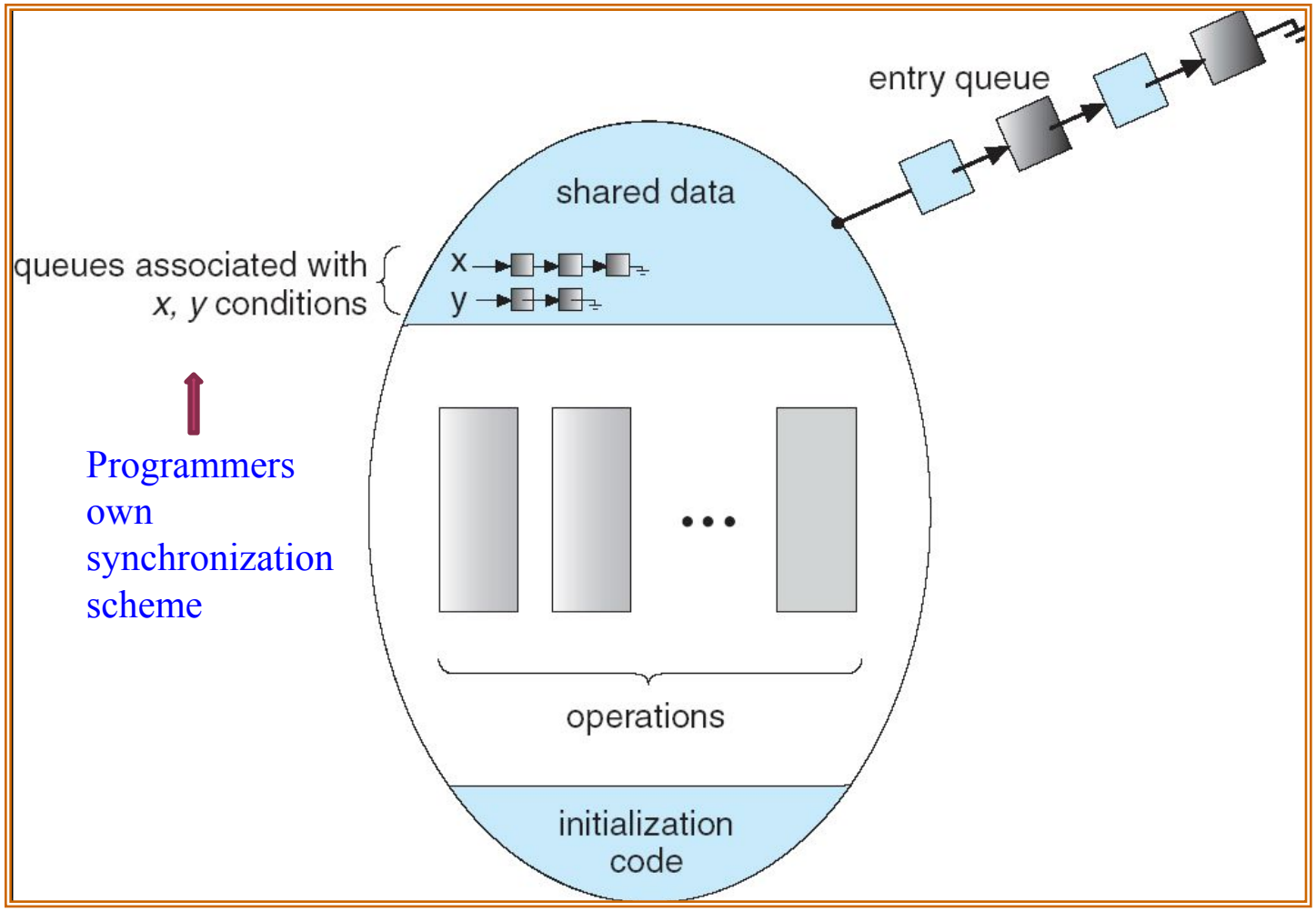


Condition Variables

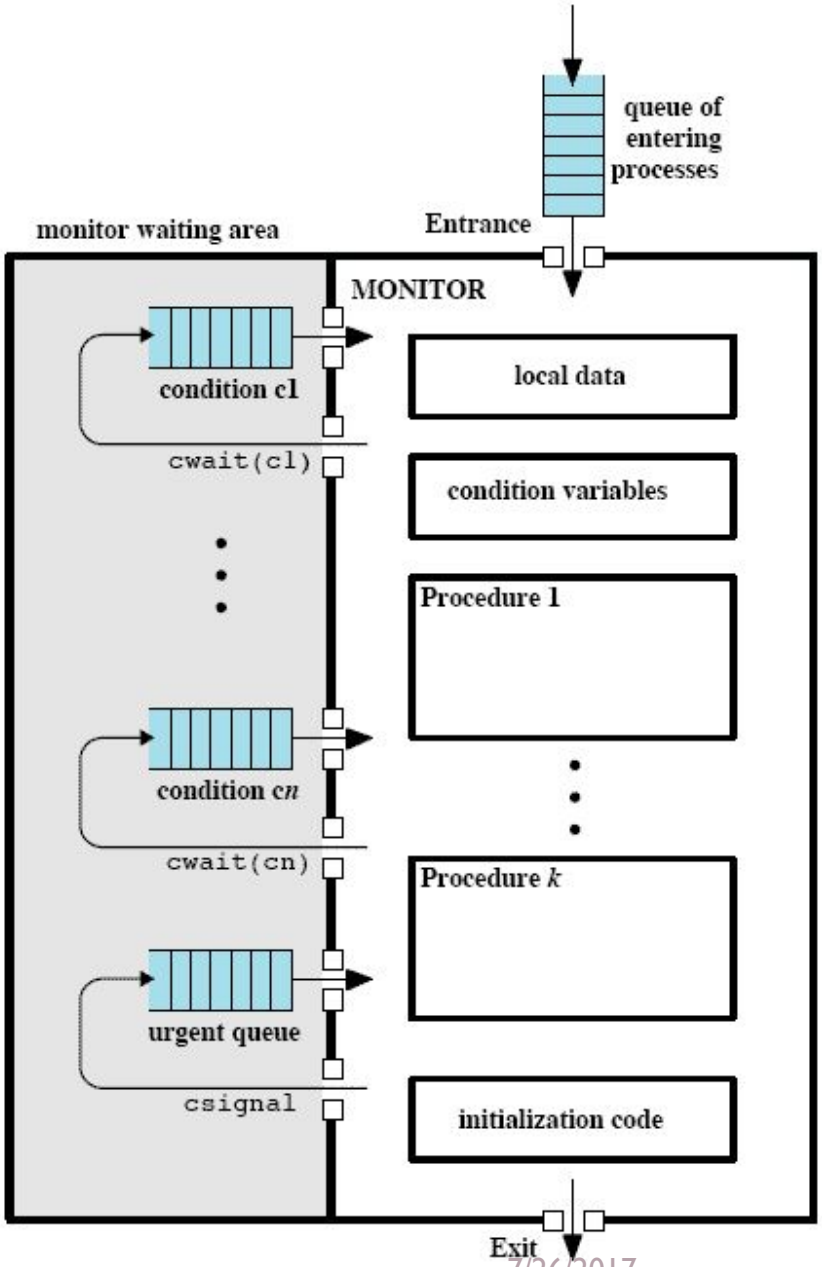
- condition x, y;

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

Monitor with Condition Variables



Structure of a Monitor



Solution to Dining Philosophers

```
monitor DP
  {
   enum { THINKING; HUNGRY, EATING} state [5] ;
   condition self [5];

     void pickup (int i)
     {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING) self [i].wait;
     }
     void test (int i)
     {
         if ( (state[(i + 4) % 5] != EATING) && (state[i] == HUNGRY)
               && (state[(i + 1) % 5] != EATING) )
          {
                state[i] = EATING ;
           self[i].signal () ;
           }
     }
```

Solution to Dining Philosophers (cont)

```
void putdown (int i)
     {
          state[i] = THINKING;
                // test left and right neighbors
           test((i + 4) % 5);
           test((i + 1) % 5);
       }
initialization_code( )
    {
          for (int i = 0; i < 5; i++)
          state[i] = THINKING;
    }
}
```

Solution to Dining Philosophers (cont)

- Each philosopher *I* invokes the operations pickup()

  and putdown() in the following sequence:

```
dp.pickup (i)

    EAT

dp.putdown (i)
```

Deadlock

- Deadlock
- Characterization
- Detection
- Recovery
- Avoidance and Prevention

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 disk drives.
  - P_1 and P_2 each hold one disk drive and each needs another one.
- Example
  - 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 the resource.**
  - **use the resource.**
  - **Release the resource.**

Deadlock Characterization

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.
- **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 $\{P_0, P_1, \ldots, P_0\}$ 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

  $P_2, \ldots, P_{n-1}$ is waiting for a resource that is held by

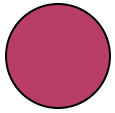
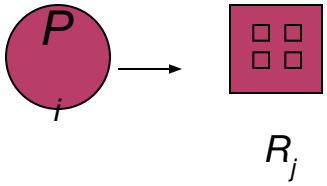
  P_n, and P_0 is waiting for a resource that is held by P_0.

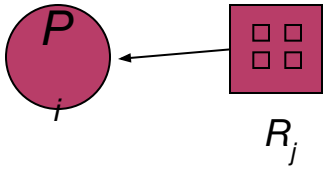
Resource-Allocation Graph

A set of vertices V and a set of edges E.

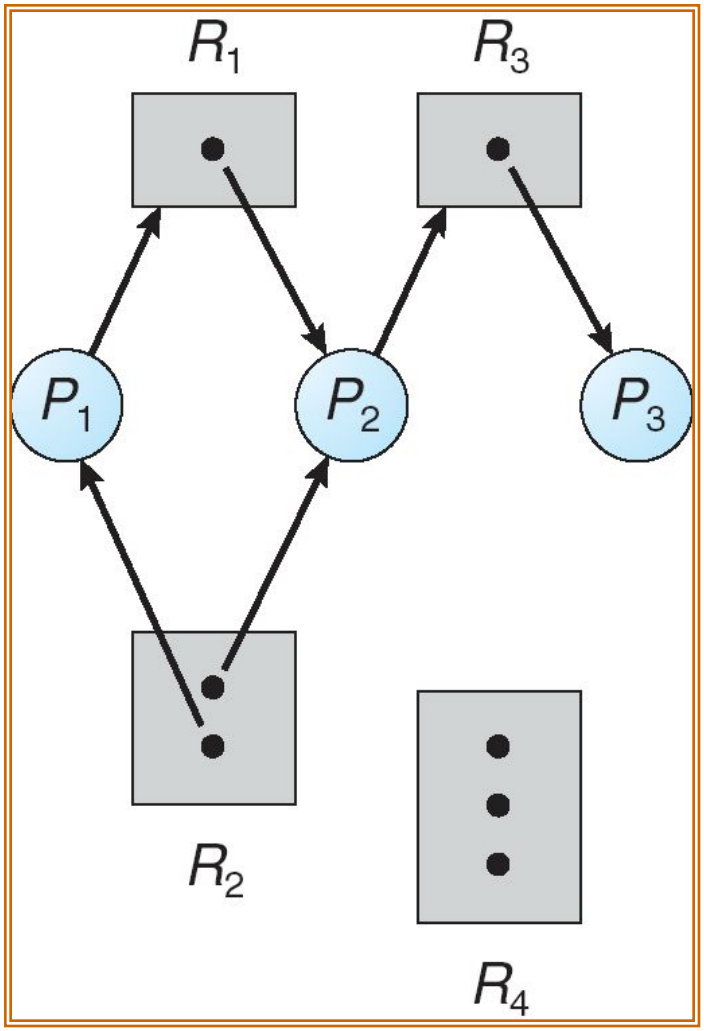
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the multi-set consisting of all resource types in the system.
- request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed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

Resource-Allocation Graph (Cont.)

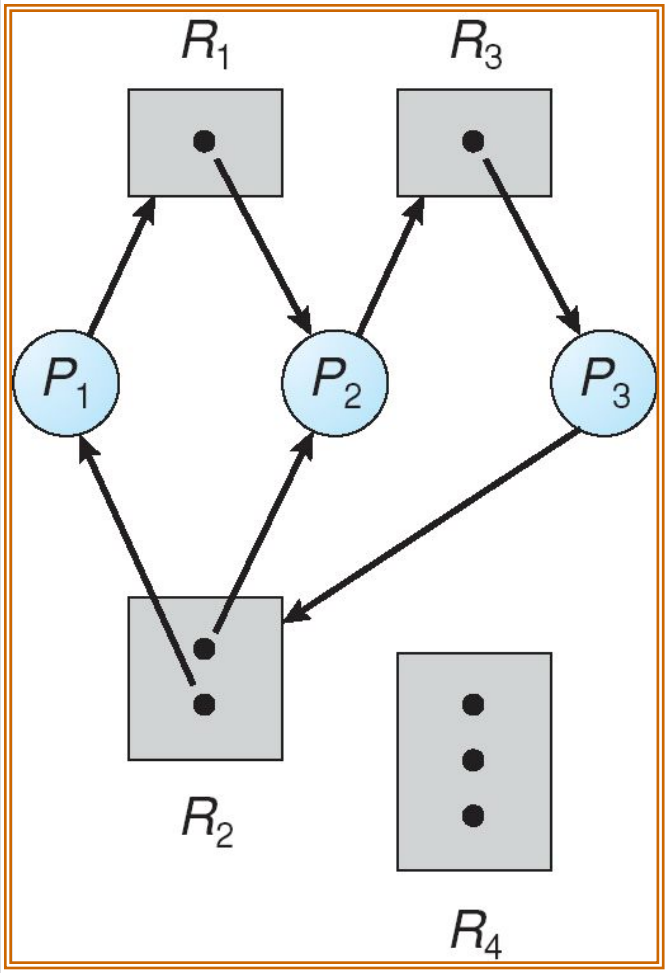
- Process 
- Resource Type with 4 instances 
- P_i requests instance of R_j 

- P_i is holding an instance of R_j 

Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock

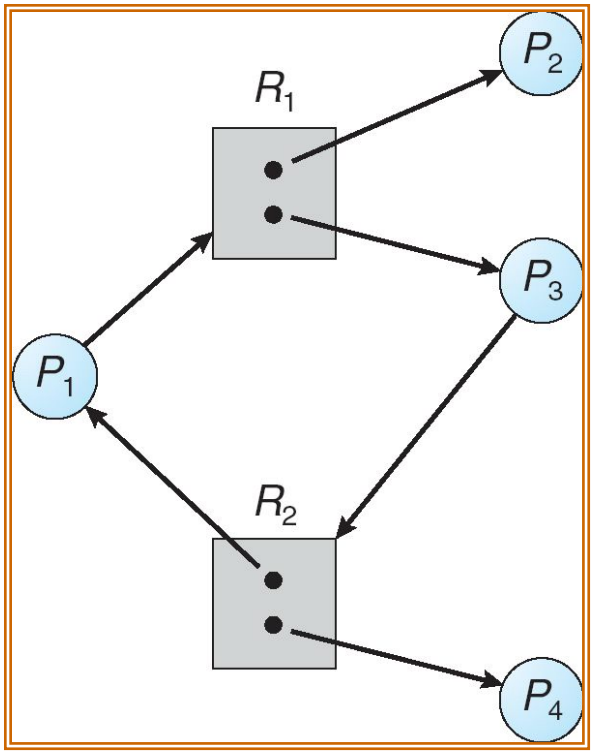


Two cycles:

1) P1 → R1 → P2 → R3 → P3 → R2 → R1

2) P2 → R3 → P3 → R2 → P2

Graph With A Cycle But No Deadlock



1) P1 → R1 → P3 → R2 → P1

Process P4 may release its instances of resource type R2.

That resource can be allocated to P3, breaking the cycle.

Basic Facts

- 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.

Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state.
  - **Prevention** : prevent any one of the four conditions from happening.
  - **Avoidance** : Allow all deadlock conditions, but calculate cycles about to happen and stop dangerous operations. (Knowledge of requests & resources.)
- Allow the system to enter a deadlock state and then recover.
  - **Detection** : know a deadlock has occurred.
  - **Recovery** : regain the resources
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX. (manually system restarted)

Deadlock Prevention

- **Mutual Exclusion** –
- not required for sharable resources; must hold for non-sharable resources.
- Mutual exclusion can not be prevented, but idea here is recognize and use sharable resources as much as possible.

Deadlock Prevention

- **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,

   or

2. allow process to request resources only when the process has none.

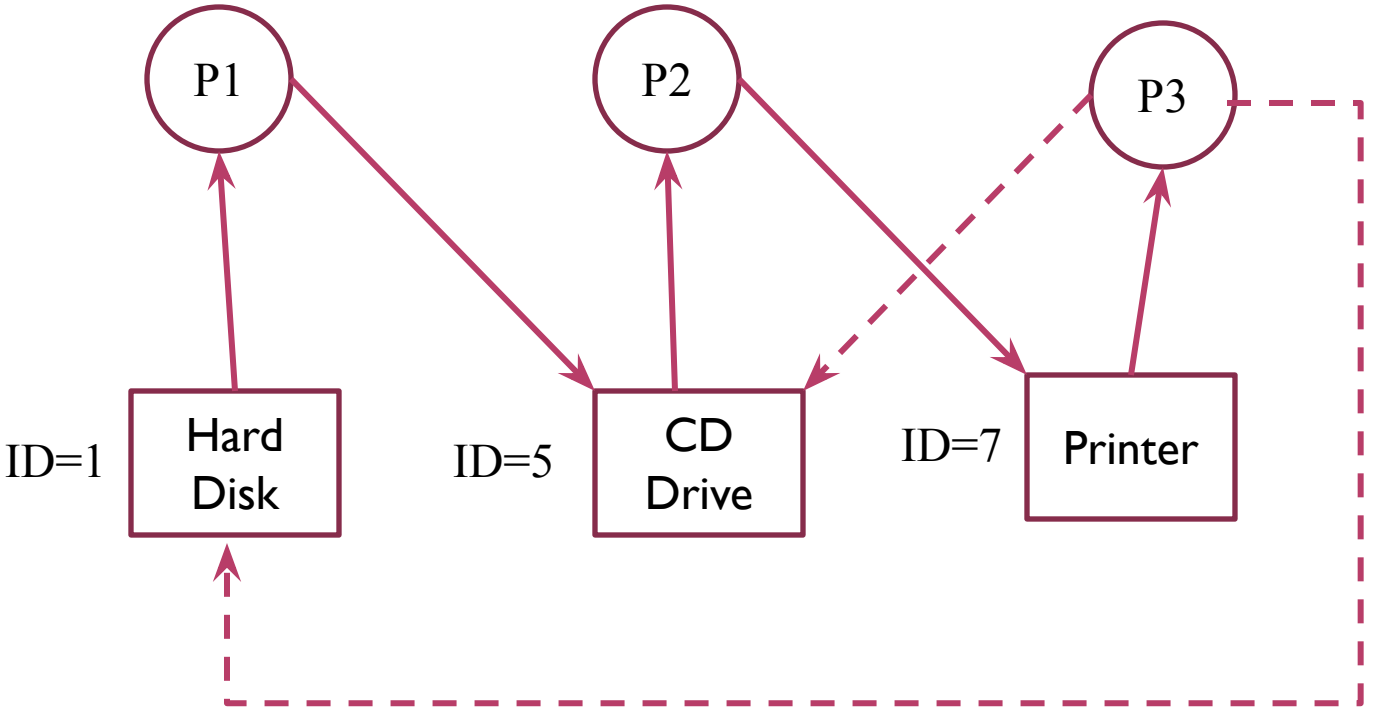
   - Low resource utilization; starvation possible.

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.

Deadlock Prevention (Cont.)

- **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 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.

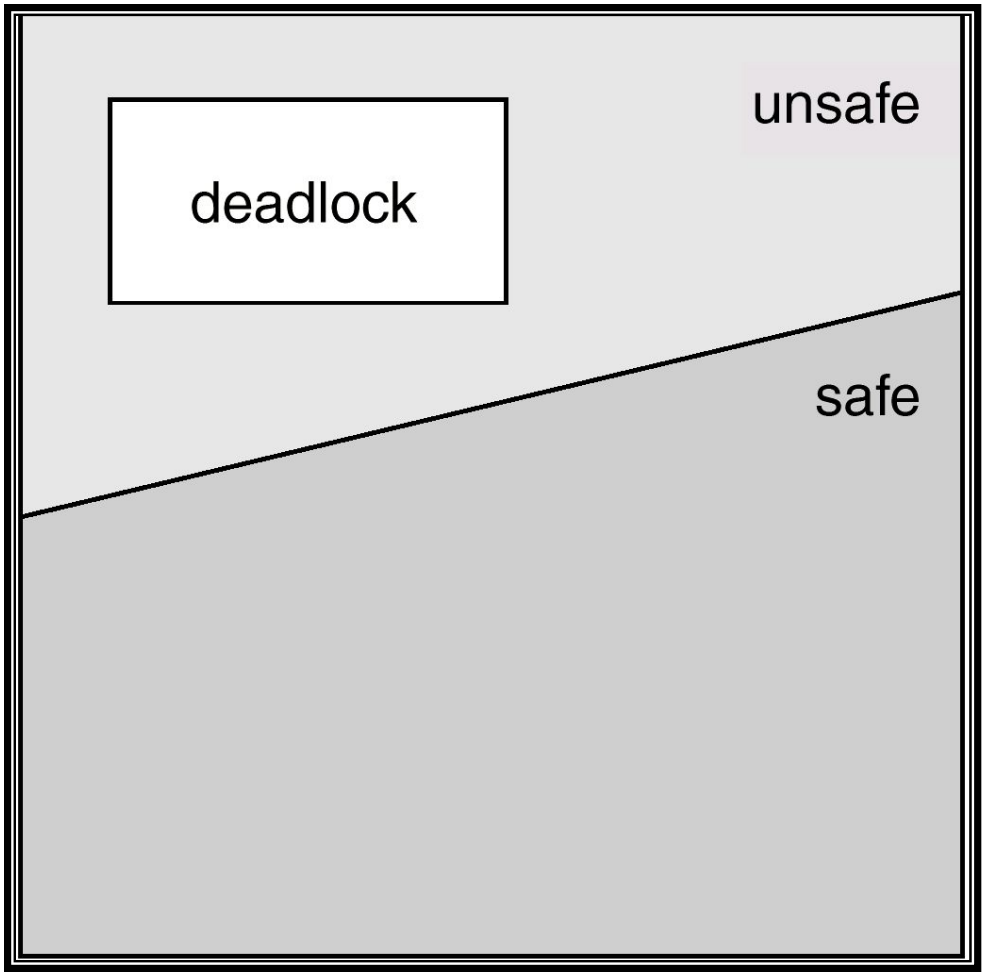
Safe State

- 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 $<P_1, P_2, \ldots, P_n>$ is safe if for each P_i, the resources that Pi can still request can be satisfied by currently available resources + resources held by all the P_j, with $j<i$.
  - 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 ⇒ no deadlocks.
- If a system is in unsafe state ⇒ possibility of deadlock.
- Avoidance ⇒ ensure that a system will never enter an unsafe state.

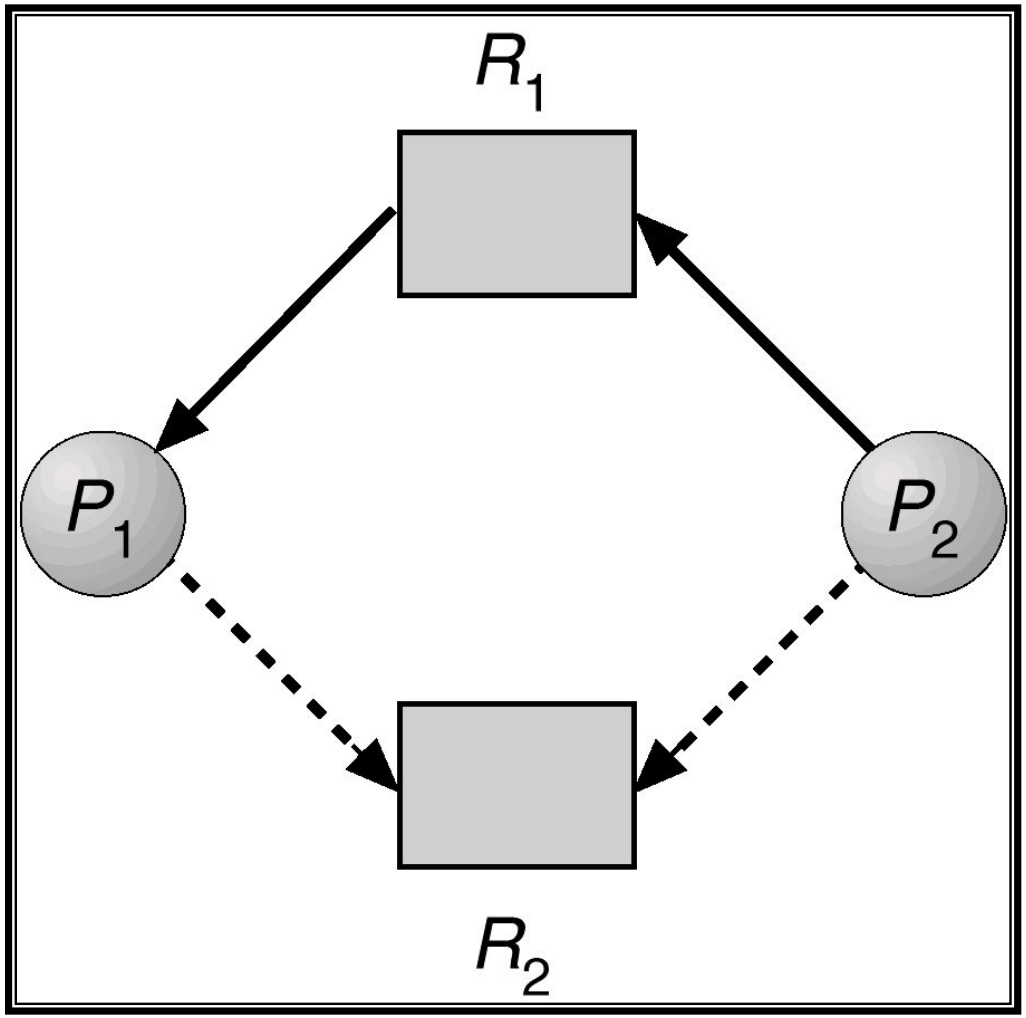
Safe, Unsafe , Deadlock State



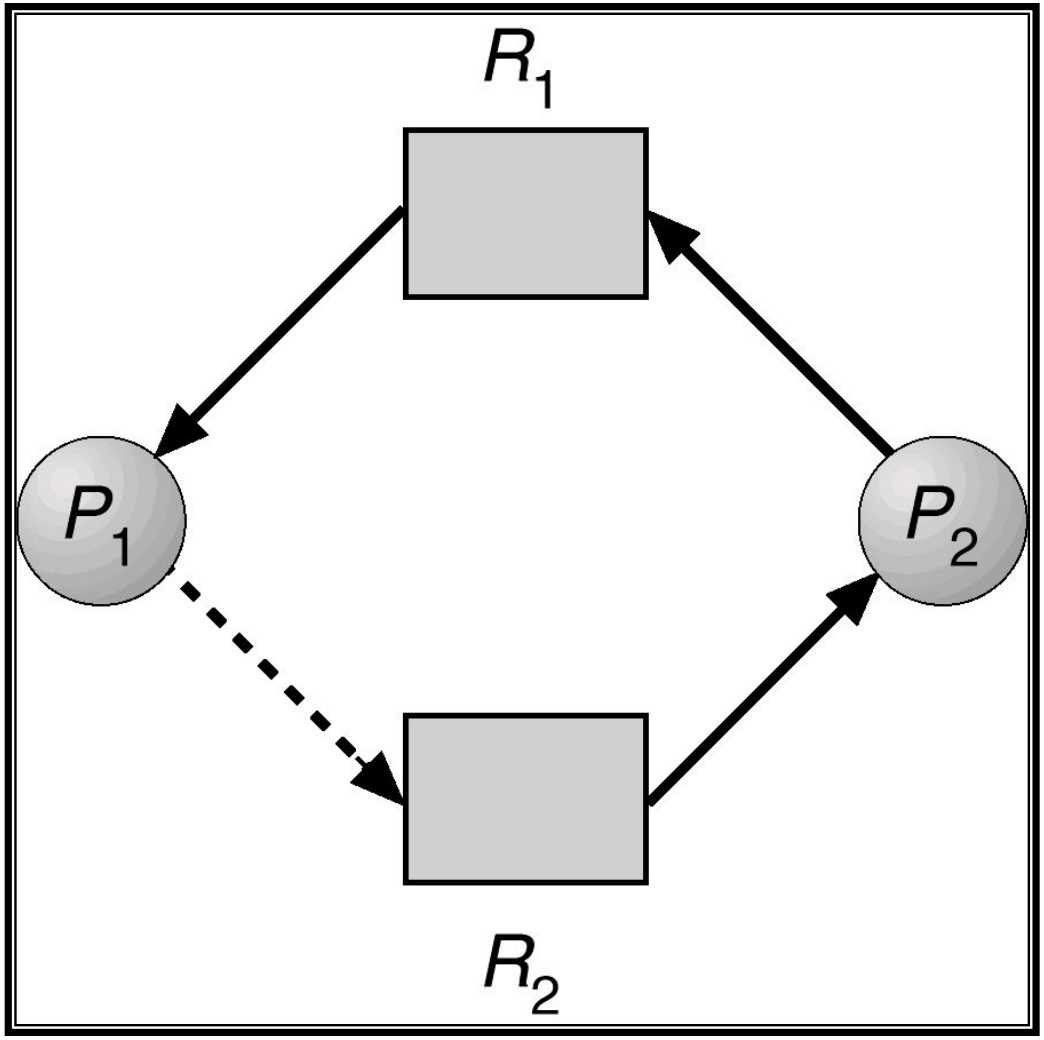
Resource-Allocation Graph Algorithm

- *Claim edge* $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line.
- Claim edge converts to request edge when a process requests a resource.
- 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 For Deadlock Avoidance



Unsafe State In Resource-Allocation Graph



Banker's Algorithm

- 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.

$$\boldsymbol{Need\ [i,j] = Max[i,j] - Allocation\ [i,j].}$$

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 = 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:

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_{i;;}$$

- ***If safe ⇒ the resources are allocated to P_i.***
- ***If unsafe ⇒ P_i 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),
- C (7 instances).

Snapshot at time T_0:

	ALLOCATION			MAX			AVAILABLE		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2
P1	2	0	0	3	2	2			
P2	3	0	2	9	0	2			
P3	2	1	1	2	2	2			
P4	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
- $< P_1, P_3, P_4, P_2, P_0 >$ satisfies safety criteria.

Example P_1 Request (1,0,2) (Cont.)

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

	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 request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

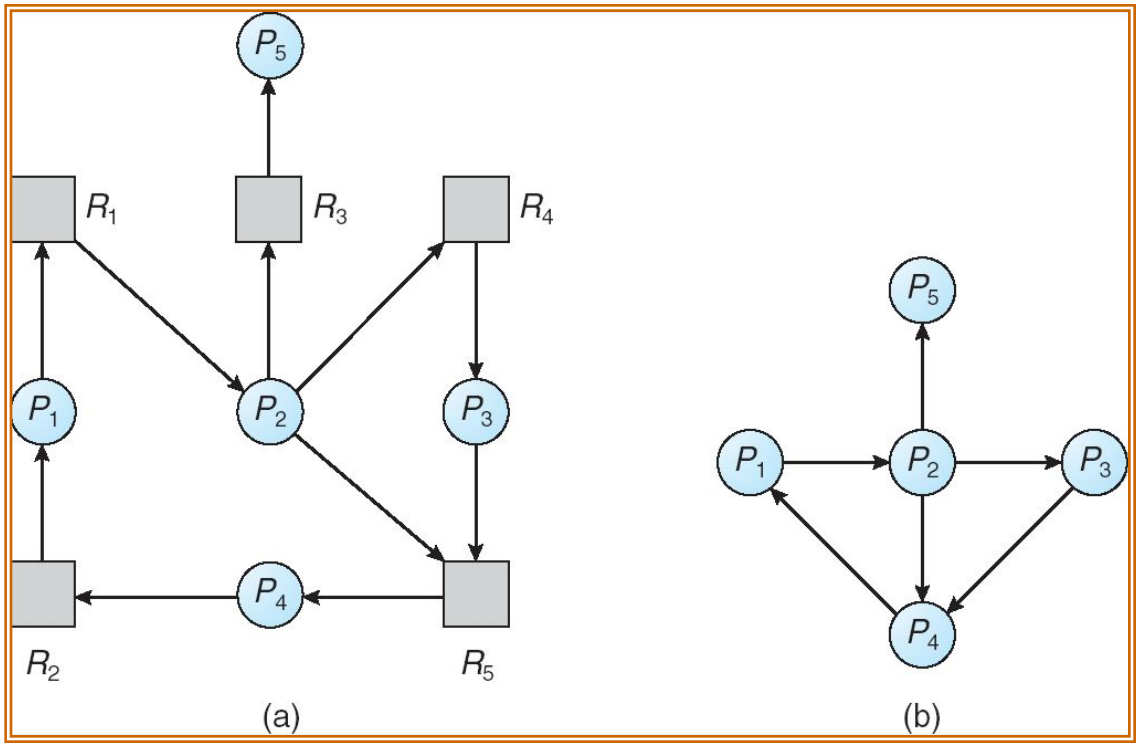
Deadlock Detection

- 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. 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

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.

Detection Algorithm (Cont.)

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.

Algorithm requires an order of O(m x $n^{2)}$ 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.)

- 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?

- if the deadlock-detection algorithm is invoked for every resource request, this will incur a considerable overhead in computation time
- If detection algorithm is invoked at random, 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.
- A less expensive alternative is simply to invoke the algorithm at less frequent intervals — for example, once per hour or whenever CPU utilization drops below 40 percent.

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 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 – To minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed during its execution.
- Rollback – return to some safe state, restart process from that state.
- Starvation – same process may always be picked as victim, include number of rollback in cost factor.