



Module-3: Process Synchronization, Deadlocks



Module 3: Process Synchronization

- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization



Objectives

- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To introduce the concept of an atomic transaction and describe mechanisms to ensure atomicity



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



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
```

- 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}
```



Solution to Critical-Section Problem

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
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



Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
 - int **turn**;
 - Boolean **flag[2]**
- The variable **turn** indicates whose turn it is to enter the critical section.
- The **flag** array is used to indicate if a process is ready to enter the critical section. **flag[i]** = true implies that process P_i is ready!



Algorithm for Process P_i

do {

```
flag[i] = TRUE;  
turn = j;  
while (flag[j] && turn == j);
```

critical section

```
flag[i] = FALSE;
```

remainder section

} while (TRUE);



Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
 - Currently running code would execute without preemption
 - Generally too inefficient on multiprocessor systems
 - 4 Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
 - 4 Atomic = non-interruptable
 - Either test memory word and set value
 - Or swap contents of two memory words



Solution to Critical-section Problem Using Locks

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



TestAndndSet Instruction

- Definition:

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



Solution using TestAndSet

- Shared boolean variable lock., initialized to false.
- Solution:

```
do {  
    while ( TestAndSet (&lock ))  
        ; // do nothing  
  
    // critical section  
  
    lock = FALSE;  
  
    // remainder section  
  
} while (TRUE);
```



Swap Instruction

- Definition:

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



Solution using Swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key

- Solution:

```
do {  
    key = TRUE;  
    while ( key == TRUE)  
        Swap (&lock, &key );  
  
        // critical section  
  
    lock = FALSE;  
  
        // remainder section  
  
} while (TRUE);
```




Bounded-waiting Mutual Exclusion with TestAndSet()

```
do {  
    waiting[i] = TRUE;  
    key = TRUE;  
    while (waiting[i] && key)  
        key = TestAndSet(&lock);  
    waiting[i] = FALSE;  
    // critical section  
    j = (i + 1) % n;  
    while ((j != i) && !waiting[j])  
        j = (j + 1) % n;  
    if (j == i)  
        lock = FALSE;  
    else  
        waiting[j] = FALSE;  
    // remainder section  
} while (TRUE);
```



Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore S – integer variable
- Two standard operations modify S : `wait()` and `signal()`
 - Originally called `P()` and `V()`
- Less complicated
- Can only be accessed via two indivisible (atomic) operations
 - `wait (S) {`
 - `while S <= 0`
 - `; // no-op`
 - `S--;`
 - `}`
 - `signal (S) {`
 - `S++;`
 - `}`



Semaphore as General Synchronization Tool

- 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
 - Also known as mutex locks
- Can implement a counting semaphore **S** as a binary semaphore
- Provides mutual exclusion

```
Semaphore mutex; // initialized to 1
do {
    wait (mutex);
    // Critical Section
    signal (mutex);
    // remainder section
} while (TRUE);
```



Semaphore Implementation

- Must guarantee that no two processes can execute `wait ()` and `signal ()` on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
 - Could now have `busy waiting` in critical section implementation
 - 4 But implementation code is short
 - 4 Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.



Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
 - value (of type integer)
 - pointer to next record in the list
- Two operations:
 - **block** – place the process invoking the operation on the appropriate waiting queue.
 - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue.



Semaphore Implementation with no Busy waiting (Cont.)

- Implementation of wait:

```
wait(semaphore *S) {  
    S->value--;  
    if (S->value < 0) {  
        add this process to S->list;  
        block();  
    }  
}
```

- Implementation of signal:

```
signal(semaphore *S) {  
    S->value++;  
    if (S->value <= 0) {  
        remove a process P from S->list;  
        wakeup(P);  
    }  
}
```



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

wait (S);
wait (Q);

.
. .
. .

signal (S);
signal (Q);

P_1

wait (Q);
wait (S);

.
. .
. .

signal (Q);
signal (S);

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** - Scheduling problem when lower-priority process holds a lock needed by higher-priority process



Classical Problems of Synchronization

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



Bounded-Buffer Problem

- N buffers, each can hold one item
- Semaphore **mutex** initialized to the value 1
- Semaphore **full** initialized to the value 0
- Semaphore **empty** initialized to the value N .



Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {  
  
    // produce an item in nextp  
  
    wait (empty);  
    wait (mutex);  
  
    // add the item to the buffer  
  
    signal (mutex);  
    signal (full);  
} while (TRUE);
```



Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
do {  
    wait (full);  
    wait (mutex);  
  
    // remove an item from buffer to nextc  
  
    signal (mutex);  
    signal (empty);  
  
    // consume the item in nextc  
  
} while (TRUE);
```



Readers-Writers Problem

- A data set is shared among a number of concurrent processes
 - Readers – only read the data set; they do **not** perform any updates
 - Writers – can both read and write
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time
- Shared Data
 - Data set
 - Semaphore **mutex** initialized to 1
 - Semaphore **wrt** initialized to 1
 - Integer **readcount** initialized to 0



Readers-Writers Problem (Cont.)

- The structure of a writer process

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



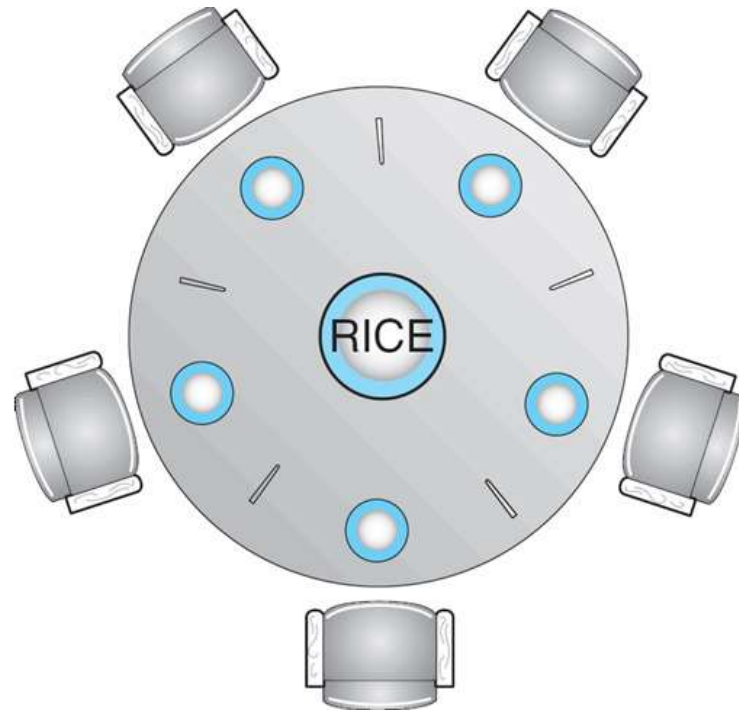
Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {  
    wait (mutex) ;  
    readcount ++ ;  
    if (readcount == 1)  
        wait (wrt) ;  
    signal (mutex)  
  
    // reading is performed  
  
    wait (mutex) ;  
    readcount - - ;  
    if (readcount == 0)  
        signal (wrt) ;  
    signal (mutex) ;  
} while (TRUE);
```



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);
```




Problems with Semaphores

- Incorrect use of semaphore operations:
 - signal (mutex) wait (mutex)
 - wait (mutex) ... wait (mutex)
 - Omitting of wait (mutex) or signal (mutex) (or both)



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

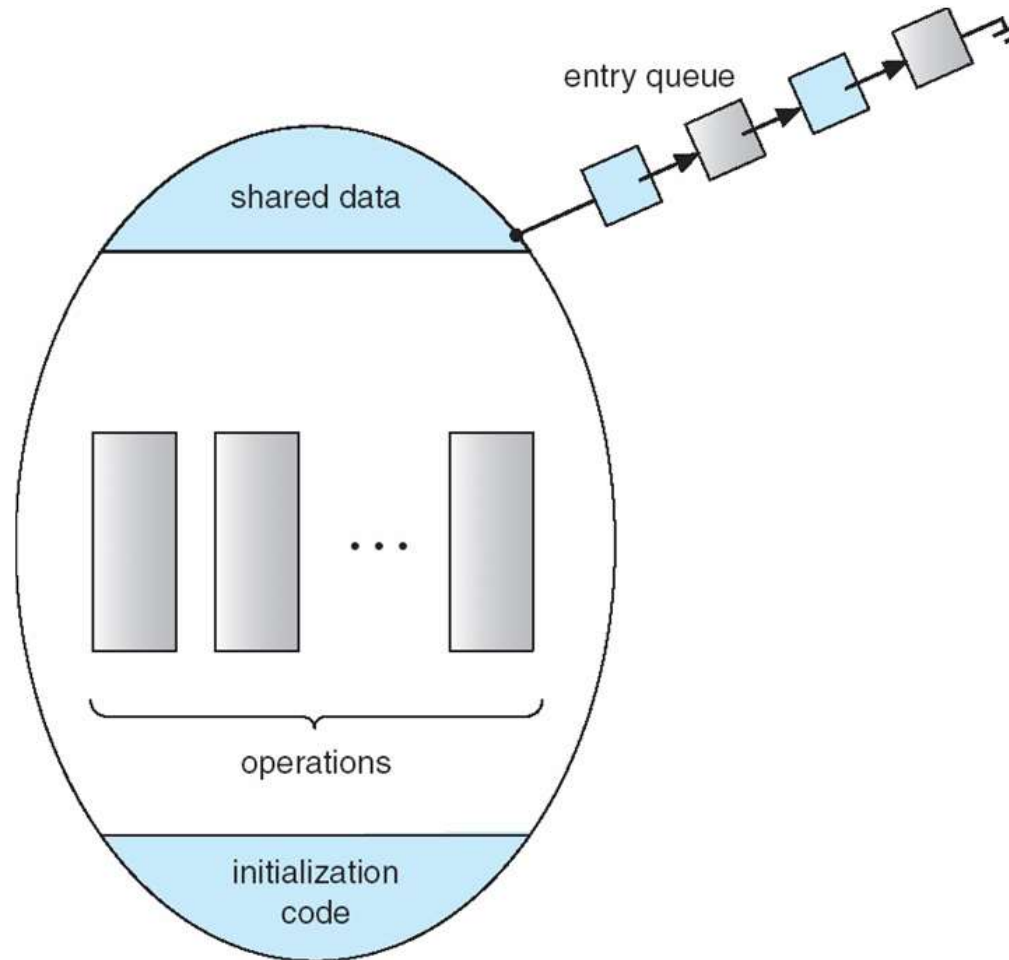
```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { .... }
    ...

    procedure Pn (...) {.....}

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



Schematic view of a Monitor



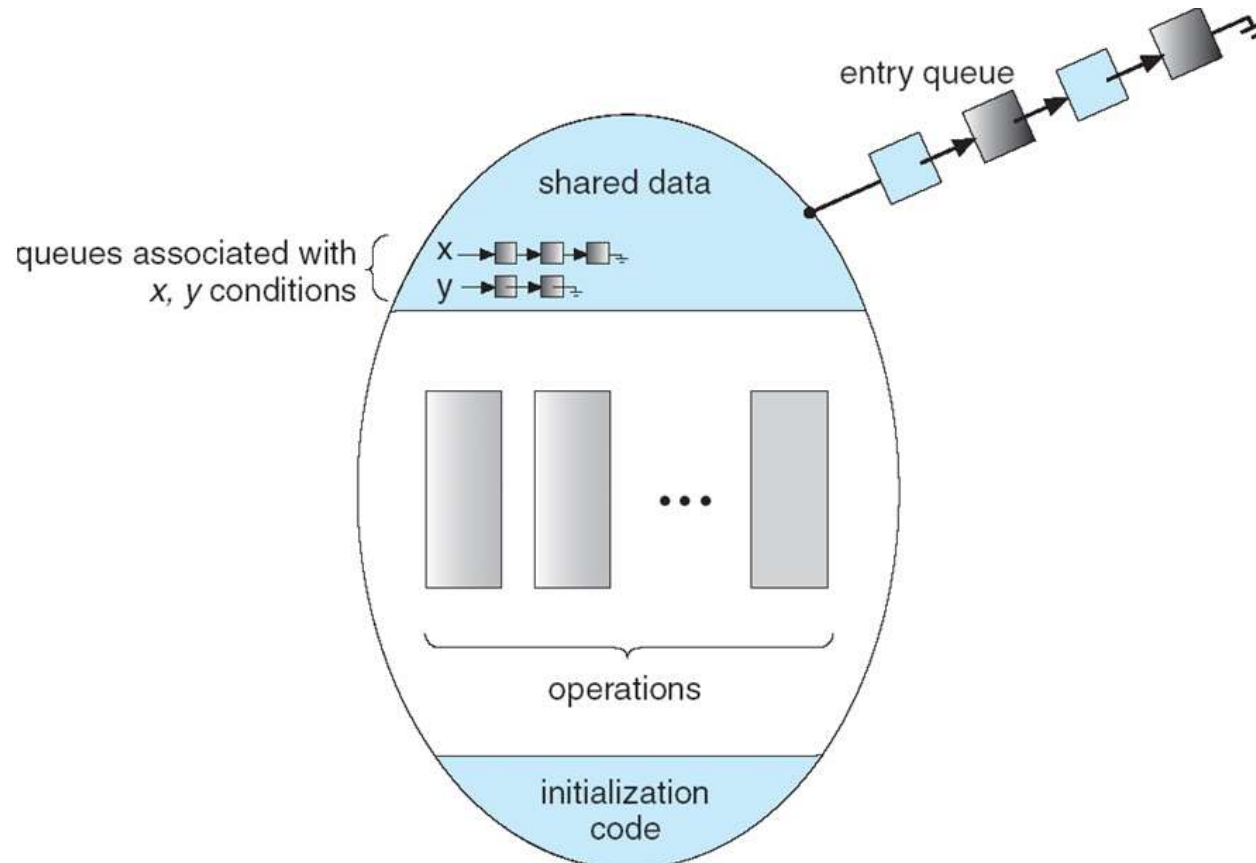


Condition Variables

- condition `x, y`;
- 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





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



Solution to Dining Philosophers (cont)

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

```
initialization_code() {  
    for (int i = 0; i < 5; i++)  
        state[i] = THINKING;  
}
```



Solution to Dining Philosophers (cont)

- Each philosopher / invokes the operations `pickup()` and `putdown()` in the following sequence:

`DiningPhilosophers.pickup (i);`

EAT

`DiningPhilosophers.putdown (i);`



Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex; // (initially = 1)
semaphore next;  // (initially = 0)
int next-count = 0;
```

- Each procedure ***F*** will be replaced by

```
wait(mutex);
...
body of F;

...
if (next_count > 0)
    signal(next)
else
    signal(mutex);
```

- Mutual exclusion within a monitor is ensured.



Monitor Implementation

- For each condition variable **x**, we have:

```
semaphore x_sem; // (initially = 0)  
int x-count = 0;
```

- The operation **x.wait** can be implemented as:

```
x-count++;  
if (next_count > 0)  
    signal(next);  
else  
    signal(mutex);  
wait(x_sem);  
x-count--;
```



Monitor Implementation

- The operation `x.signal` can be implemented as:

```
if (x-count > 0) {  
    next_count++;  
    signal(x_sem);  
    wait(next);  
    next_count--;  
}
```



A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;
    void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = TRUE;
    }
    void release() {
        busy = FALSE;
        x.signal();
    }
    initialization code() {
        busy = FALSE;
    }
}
```



End



Module 3 :

Deadlocks



Deadlocks



Chapter 6: Deadlocks

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



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



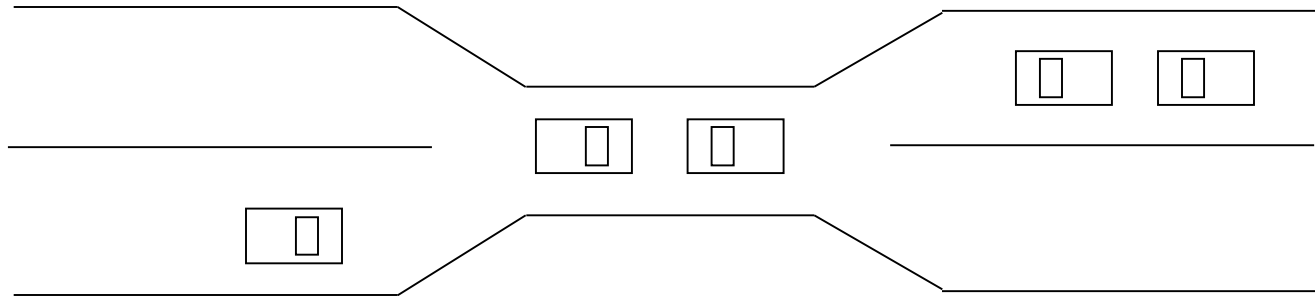
The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set
- Example
 - 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)



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
- Note – Most OSes do not prevent or deal with deadlocks



System Model

- Resource types R_1, R_2, \dots, 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 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 $\{P_0, P_1, \dots, P_n\}$ 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

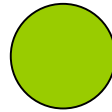
A set of vertices V and a set of edges
 E .

- V is partitioned into two types:
 - $P = \{P_1, P_2, \dots, P_n\}$, the set consisting of all the processes in the system
 - $R = \{R_1, R_2, \dots, R_m\}$, 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 (Cont.)

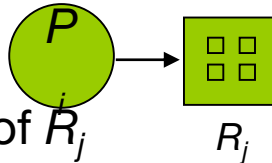
- Process



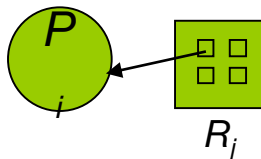
- Resource Type with 4 instances



- P_i requests instance of R_j

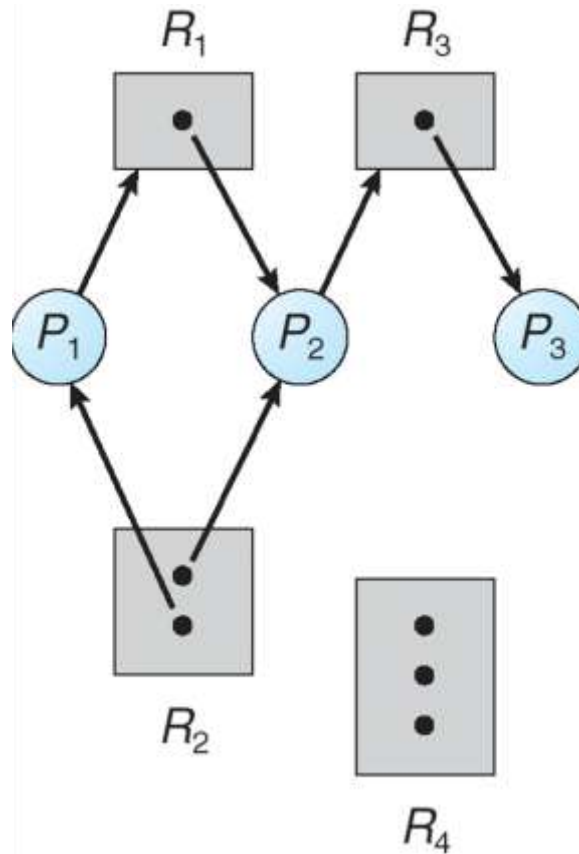


- P_i is holding an instance of R_j



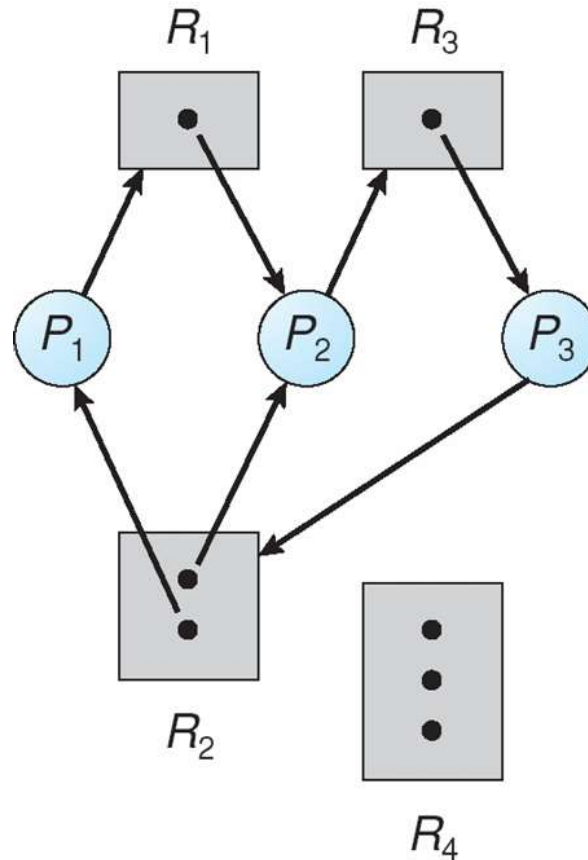


Example of a Resource Allocation Graph



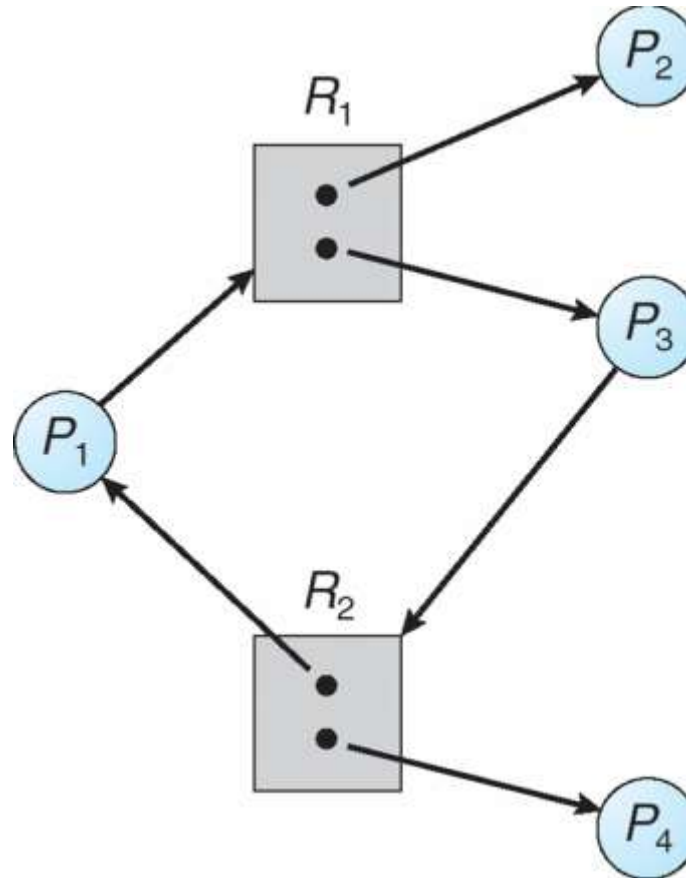


Resource Allocation Graph With A Deadlock





Graph With A Cycle But No Deadlock





Basic Facts

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



Methods for Handling Deadlocks

- 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 Prevention

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 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 sequence $\langle P_1, P_2, \dots, P_n \rangle$ of ALL the processes is the systems such that for each P_i , the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j , with $j < i$
- That is:
 - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
 - 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

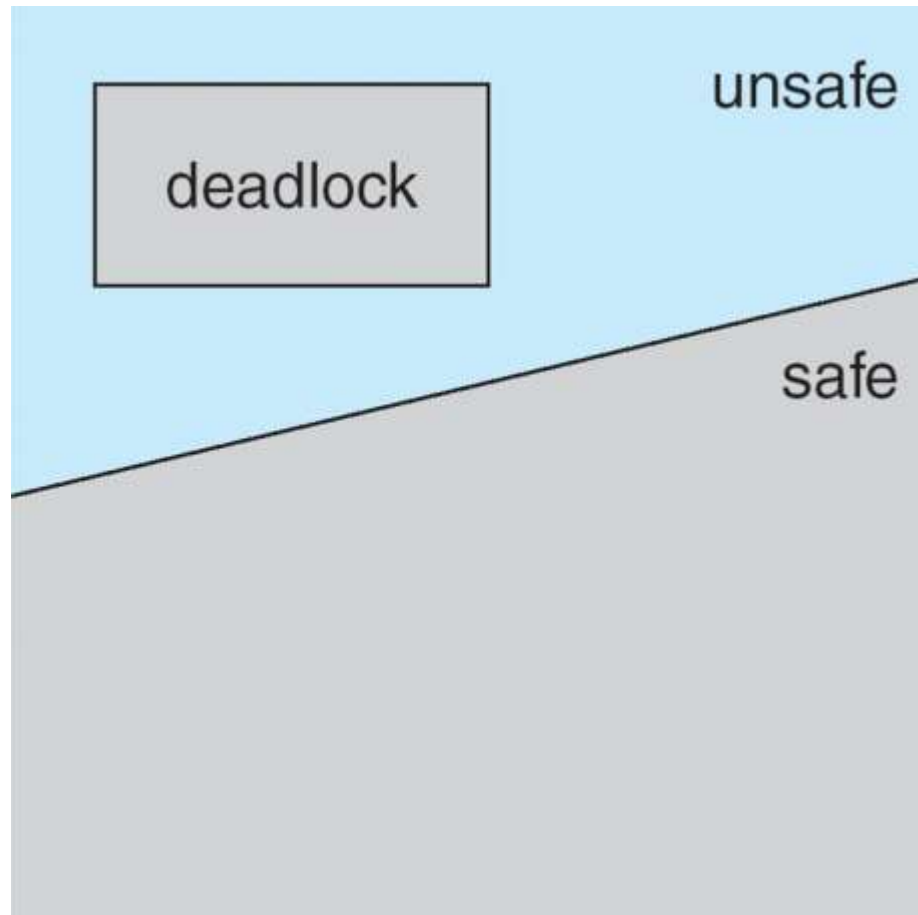


Basic Facts

- 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.



Safe, Unsafe , Deadlock State





Avoidance algorithms

- Single instance of a resource type
 - Use a resource-allocation graph
- Multiple instances of a resource type
 - Use the banker's algorithm

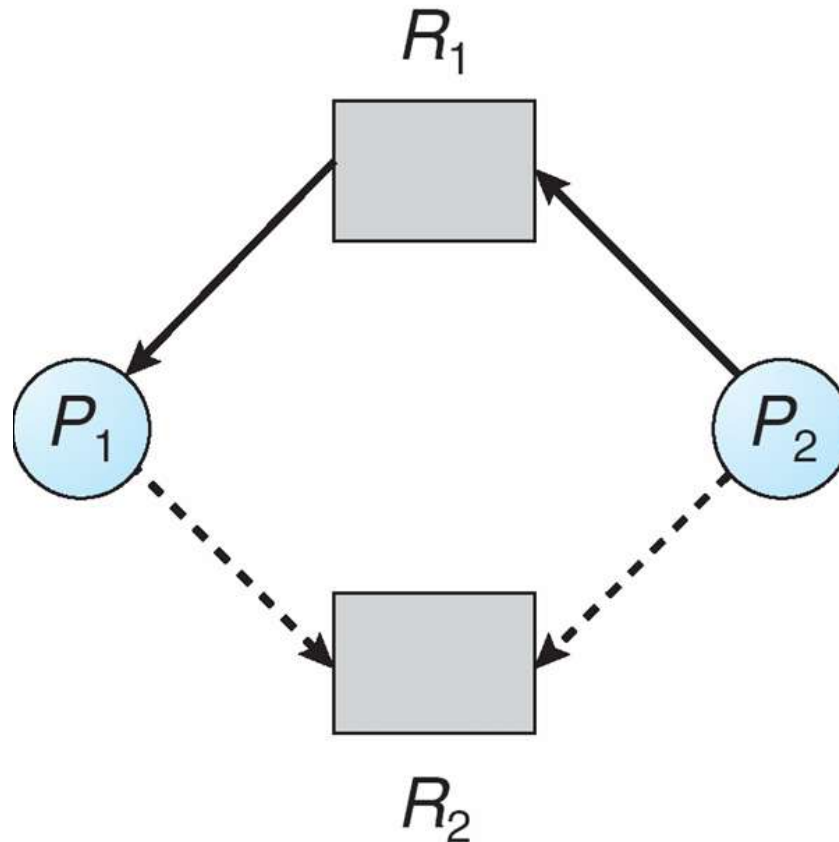


Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_i may request resource R_j ; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system

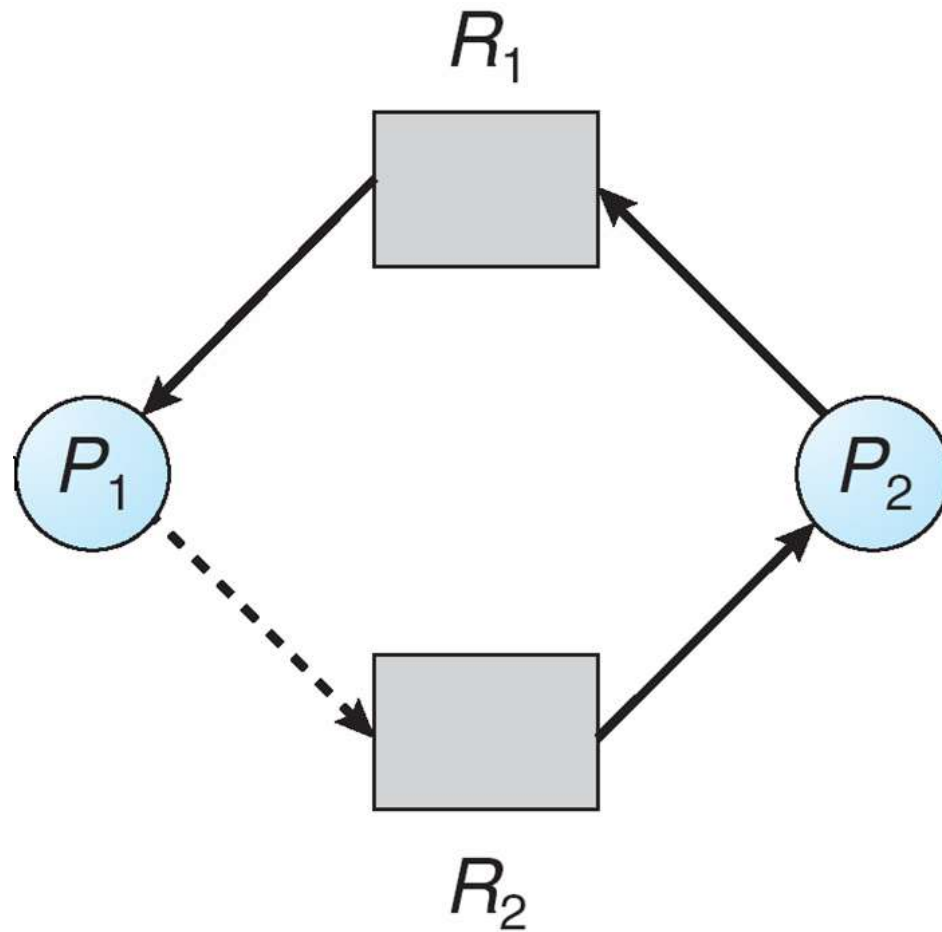


Resource-Allocation Graph





Unsafe State In Resource-Allocation Graph





Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph



Banker's Algorithm

- 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:** $n \times m$ matrix. If $Max[i, j] = k$, then process P_i may request at most k instances of resource type R_j
- **Allocation:** $n \times m$ matrix. If $Allocation[i, j] = k$ then P_i is currently allocated k instances of R_j
- **Need:** $n \times m$ matrix. If $Need[i, j] = k$, then P_i may need k more instances of R_j to complete its task

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



Safety Algorithm

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

Work = *Available*

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

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 \Rightarrow the resources are allocated to P_i
- If unsafe $\Rightarrow P_i$ 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 (5 instances), and C (7 instances)

Snapshot at time T_0 :

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>
	$A \ B \ C$	$A \ B \ C$	$A \ B \ C$
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	



Example (Cont.)

- The content of the matrix *Need* is defined to be *Max – Allocation*

	<u>Need</u>
	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 $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies safety criteria



Example: P_1 Request (1,0,2)

- Check that Request \leq Available (that is, $(1,0,2) \leq (3,3,2) \Rightarrow$ true

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	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 $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ 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

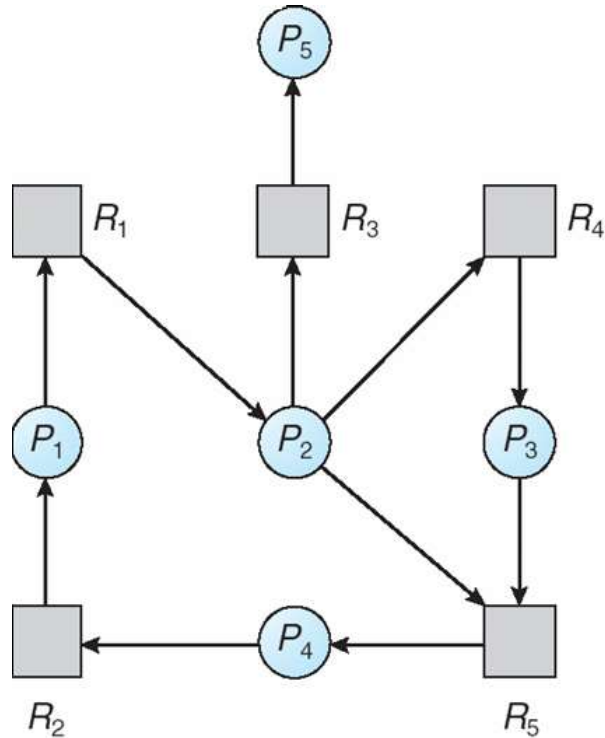


Single Instance of Each Resource Type

- 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

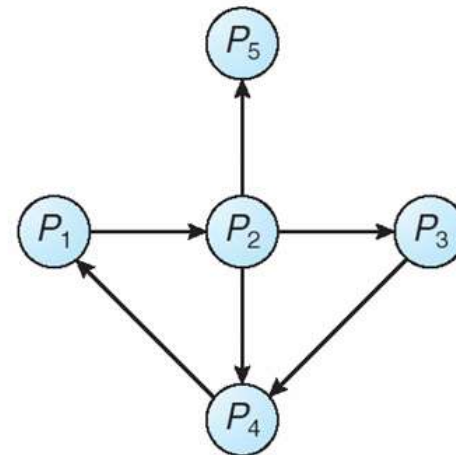


Resource-Allocation Graph and Wait-for Graph



(a)

Resource-Allocation
Graph



(b)

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 $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
- **Request:** An $n \times m$ matrix indicates the current request of each process. If $Request[i_j] = k$, then process P_i is requesting k more instances of resource type R_j .



Detection Algorithm

1. Let *Work* and *Finish* be vectors of length m and n , respectively Initialize:
 - (a) *Work* = *Available*
 - (b) For $i = 1, 2, \dots, n$, if $Allocation_i \neq 0$, then
 $Finish[i] = \text{false}$; otherwise, $Finish[i] = \text{true}$
2. Find an index i such that both:
 - (a) $Finish[i] == \text{false}$
 - (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

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 :

	<u>Allocation</u>	<u>Request</u> <u>Available</u>
	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 $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in $Finish[i] = \text{true}$ for all i



Example (Cont.)

- P_2 requests an additional instance of type C

	<u>Request</u>		
	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?
 - 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



- 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?
 - 4 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
- 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