CONCURRENCY CONTROL / PROCESS SYNCRONIZATION

SCHOOL OF COMPUTER ENGINEERING AND TECHNOLOGY

Syllabus

Concurrency Control

Process Synchronization: Principles of Concurrency, Requirements for Mutual Exclusion, Mutual Exclusion: Hardware Support, Operating System Support

Classical synchronization problems: Readers/Writers Problem, Producer and Consumer problem.

Deadlock: Principles of Deadlock, Deadlock Modeling, Deadlock Prevention, Deadlock Avoidance, Deadlock detection and recovery.

Concurrency

- Multiprogramming: Management of multiple processes within a uni-processor system
- Multiprocessing: Managing multiple processes within multiprocessor.

Design issues in concurrency

- Communication among processes
- Sharing & competition for resources
- Synchronization of activities of multiple processes
- Allocation of processor time to processes

Concurrency

3 different contexts of concurrency

- Multiple applications

  - Multiprogramming

- Structured applications

  - Some applications can be effectively programmed as a set of concurrent processes(Principles of modular design & structured programming)

- OS structure

  - OS often implemented as a set of processes or threads

Key terms related to concurrency

Atomic operation:

A sequence of one or more statements that appear to be indivisible, i.e., no other process can see an intermediate state or interrupt the operation

Critical section:

A section of code within a process that requires access to shared resources & that must not be executed while another process is in a corresponding section of code

Deadlock:

A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something

Key terms related to concurrency

Mutual exclusion:

The requirement that when one process is in Critical Section that accesses shared resources no other process may be in critical section that accesses any of those shared resources.

Race condition:

A situation in which multiple threads/processes read & write a shared data item and final result depends on relative timing of their execution.

Starvation:

A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

Difficulties due to concurrency

Sharing of global resources: Eg. Two processes both make use of global variable & both perform read & write on that variable. Order in which read & write are done is critical.

Management of resources optimally: Eg. Process has gained the ownership of i/o device but is suspended before using it, thus locking i/o device and preventing its use by other processes.

Error locating in program: Results are not deterministic and reproducible.

Example

```
void echo() { chin = getchar();
              chout = chin;
              putchar(chout);
             }
```

- Uniprocessor multiprogramming , single user environment
- Many applications can call this procedure repeatedly to accept user i/p & display on screen
- User can jump from one application to other.
- Each application needs/ uses procedure echo.
- Echo is made shared procedure and loaded into a portion of memory, global to all applications
- Single copy of echo procedure is used, saving space

Problem and solution

Sequence of execution

1. Process p1 invokes echo & gets a character i/p from keyboard say x. At this point it gets interrupted.
2. Process P2 invokes echo accepts character as y and displays it on screen.
3. Process p1 resumes, value of chin was x but now overwritten as y, thus chin has lost value which it had.

Solution: When echo procedure is invoked by any process and if that process gets suspended for any reason before completing it, then no other process can invoke echo till process that was suspended is resumed & completes echo. Thus other processes are blocked from entering into echo.

Same is applicable to multiprocessor systems

Race condition

- A race condition occurs when multiple competing processes or threads read and write data items so that final result depends on the order of execution of instructions in multiple processes.
- Two processes p1 & p2, share global variable ‘a’. P1 updates a to 1 and then p2 updates it to 2. Thus two tasks are in race to write variable ‘a’. Loser of race (the one who updates last) determines the value of ‘a’

Race condition

Another eg.

Initially, shared global variables have values b=1 & c=2

P1 executes b=b+c

P2 executes c=b+c

If p1 executes first, then b=? & c=?

If p2 executes first, then b=? & c=?

Race condition

Another eg.

Initially, shared global variables have values b=1 & c=2

P1 executes b=b+c

P2 executes c=b+c

If p1 executes first, then b=3 & c=5

If p2 executes first, then b=4 & c=3

How Processes interact with each other

- Processes unaware of each other
- Processes indirectly aware of each other
- Processes directly aware of each other

How Processes interact with each other

Processes unaware of each other **(Competition)**

- E.g. Multiprogramming of multiple independent processes
- OS need to know about competition for resources such as printer, disk, file, etc
- Potential problems : mutual exclusion, deadlock, starvation

Processes indirectly aware of each other **(Cooperation by sharing)**

- Shared access to some object such as shared variable
- Cooperation by sharing
- Potential problem: mutual exclusion, deadlock, starvation, data coherence

How Processes interact with each other

Processes directly aware of each other

(Cooperation by communication)

- Cooperation by communication, communication primitives available
- Potential control Problems: Deadlock and starvation
- Mutual exclusion not a problem
- Deadlock possible
- Starvation possible

Three Control Problems: Competition

- Need for mutual exclusion: Two or more processes require access to single non-sharable resource such as printer. Such a resource is called as Critical resource and portion of code using it is called as critical section(CS) of program.
- Mutual Exclusion - Only one process should be allowed in its critical section.
- Deadlock
- Starvation

Control problems: Competition

Deadlock: Mutual exclusion creates a problem of deadlock.

p1← printer and p2← file

both p1 and p2 require printer and file to complete the task but printer is blocked from using by p1 and file is blocked from using by p2. Thus leading a deadlock

Control problems: Competition

Starvation:

- Three processes p1, p2, p3 require a periodic access to resource type R.
- Process p1 currently using R, so p2 and p3 must wait.
- When process p1 exits critical section, suppose p3 gains the control, and suppose p1 again needs resource R and OS assigns it to p1 when p3 exits the CS.
- This situation may continue and p2 never gets ownership of R.

Co-operation among processes by sharing

- Processes that interact with other processes without being explicitly aware of them. Access to shared variables/files/databases
- Processes may use & update shared data without reference to other processes but know that other processes may have access to same data
- Processes must co-operate to ensure that the data that they share are properly managed.
- Control problems of mutual exclusion, deadlock & starvation are again present.
- Data items are accessed in 2 modes: reading & writing
- Writing operations must be mutually exclusive.

Co-operation among processes by sharing

- New requirement introduced i.e. Data coherence
- Suppose 2 data items a & b are to be maintained in the relationship a=b i.e. any program that updates one value must update other to maintain the relationship
- Consider 2 processes

p1: a = a+1

b = b+1;

p2: b = 2*b

a = 2*a

Requirement is always a=b

Initially a=b=1

Concurrent execution

a = a + 1

b = 2 * b

b= b + 1;

a = 2 * a

This leads to a=4 & b=3

Co-operation among processes by sharing

- New requirement introduced i.e. Data coherence
- Suppose 2 data items a & b are to be maintained in the relationship a=b i.e. any program that updates one value must update other to maintain the relationship
- Consider 2 processes

p1: a = a+1

b = b+1;

p2: b = 2*b

a = 2*a

Requirement is always a=b

Initially a=b=1

Concurrent execution

a = a + 1

b = 2 * b

b= b + 1;

a = 2 * a

This leads to a=4 & b=3

Solution : Declare entire sequence in each process as CS

Co-operation among processes by communication

- Processes co-operate by communication, since they participate in common effort that links all of the processes.
- Communication provides a way to synchronize / co-ordinate various activities
- Communication is in form of sending and receiving messages.
- Mutual exclusion is not a problem, since there is no sharing.
- Deadlock possible: Two processes may be blocked, waiting for communication from each other.
- Starvation possible: (p1,p2,p3) p1 repeatedly attempts to communicate with p2 and p3. p2 & p3 attempt to communicate with p1. A sequence may arise in which p1 & p2 continuously communicate with each other while p3 is blocked & waiting for communication

Requirements for Mutual exclusion

Any facility/capability providing support for mutual exclusion should meet following requirements:

- Mutual exclusion must be **enforced**: Only one process at a time in the CS, among all processes that have CS for same resource or shared object.
- A process that **halts in its non CS** must do without interfering with other processes.
- A process requiring access to CS must not be **denied/delayed indefinitely** (no deadlock or starvation)
- When **no process in its CS**, any process that requests a entry to its CS, must be permitted to enter without delay.
- No assumptions are made about the **relative speed of processes or total no. of processes**.
- Process remains in its **CS for a finite time** only.

Approaches to satisfy the requirements of Mutual Exclusion

1. Hardware Approach
2. Support from OS or programming language
3. Software approach (No Support from OS or programming language)

Hardware Approach for Mutual Exclusion

1. Disabling Interrupts

- A process runs until it invokes an operating system service or until it is interrupted
- Disabling interrupts guarantees mutual exclusion
- Because CS cannot be interrupted, mutual exclusion is guaranteed
- Will not work in multiprocessor architecture, since computer includes more than one processor & possible for more than one process to be executing at a time
- The efficiency of execution degraded as processor is limited in its ability to interleave programs

Pseudo Code

```
while (true) {
  /* disable interrupts */;
  /* critical section */;
  /* enable interrupts */;
  /* remainder */;
}
```

Hardware Approach : Special Machine Instructions

2. Special Machine Instructions

- Processor designers proposed machine instructions that carry out two actions atomically
- Compare and exchange /compare & swap instruction or testset

Test and Set instruction

```
boolean testset(int i)//done atomically
{
  if (i==0)
     { i= 1;
     return true;
     }
  else
     return false;
}
```

```
const int n = /* no of processes */;
int bolt;
void P(int i)
{
    while(1)
     {


        while(testset(bolt) != true)
           /* do nothing */ ;
        /* critical section */ ;
         bolt = 0;
       /* remainder * /;

    }
```

```
main()
{
  bolt =0;

 Parbegin(P(1),P(2),….P(n));


 }
```

Support 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

Support Disadvantages

- **Busy waiting or spin waiting is employed** – Thus a process waiting to access a CS continuously consumes processor time
- **Starvation is possible** – when a process leaves CS, selection of a waiting process is arbitrary & hence some process could be indefinitely denied access
- **Deadlock is possible** – Eg. P1 executes special instruction & enters its CS. P1 is interrupted to give processor to P2 (having higher priority). If P2 attempts to use same resource as P1, it will be denied access because of mutual exclusion mechanism. Thus it will go into busy waiting loop. However P1 will never be dispatched because it is of lower priority than P2

Mutual Exclusion: OS or programming language support

- Semaphore
- Mutex
- Monitors

Semaphore

- Provides multiple process solution for mutual exclusion
- Uses a variable which is an integer
- It is accessed only through two standard atomic(indivisible) operations: wait & signal
- Semaphores are of 2 types…

1. Binary : Variable takes values 0 or 1
2. General/Counting : Variable takes any integer value

Semaphore

Wait : Is used for acquiring a shared resource represented by the semaphore
Hence wait decrements the value of semaphore variable

- Denoted by P

```
wait(s)
 {
        while(s<=0)
                    ; // no operation or busy waiting
             s--;
 }
```

Semaphore

Signal: releases the shared resource represented by the semaphore

Therefore wait increments the value of semaphore variable

Denoted by V

```
signal(s)
  {
          s++;
  }
```

Semaphore : Removing busy waiting

- Semaphores suffer from busy waiting
- To overcome, modify the working of wait & signal
- When a process executes waits operation & finds semaphore value not greater than 0, it must wait
- Instead of process doing a busy wait, process is placed into a waiting queue associated with the semaphore & CPU selects another process to execute
- Waiting process is restarted, when some other process executes a signal operation. Process moves from waiting state to ready state & process is placed in ready queue

Semaphore contd.

For both counting semaphores and binary semaphores, a queue is used to hold processes waiting on the semaphore. The fairest removal policy is

- first-in-first-out (FIFO): The process that has been blocked the longest is released from the queue first;
- A semaphore whose definition includes this policy is called a **strong semaphore**.
- A semaphore that does not specify the order in which processes are removed from the queue is a **weak semaphore**

Example: Semaphore usage

- initially $m = 1$

Process P_i

{

wait(*m*);

critical section

signal(*m*);

remainder section

}

Semaphore example

Suppose we want to synchronize two concurrent processes P1 and P2 using binary semaphores s and t. The code for the processes P1 and P2 is shown below-

Process P1:

```
while(1)
{
w: __________
  print '0';
  print '0';
x:___________
}
```

Process P2:

```
while(1)
{
y: __________
  print '1';
  print '1';
z:___________
}
```

The required output is "001100110011 …….". Processes are synchronized using P & V operations on semaphores s & t. Choose the correct options from the following at points w, x, y & z. Which of the following option is correct?

1. P(s) at w, V(s) at x, P(t) at y, V(t) at z, s and t initially 0
2. P(s) at w, V(t) at x, P(t) at y, V(s) at z, s initially 1 and t initially 0
3. P(s) at w, V(t) at x, P(t) at y, V(s) at z, s and t initially 0
4. P(s) at w, V(s) at x, P(t) at y, V(t) at z, s initially 1 and t initially 1

Producer-Consumer problem

- It is one of the classic problems of synchronization
- Producer produces an item and adds to a buffer of limited size(bounded buffer)
- Consumer takes out an item from buffer and consumes it.
- Buffer is a shared resource and must be used in a mutual exclusion manner by both processes.
- Producers to be prevented from adding into a full buffer.
- Consumers to be stopped from taking out an item from an empty buffer

Producer-Consumer problem

General Situation:

- One or more producers are generating data and placing these in a buffer
- One or more consumers are taking items out of the buffer

The Problem:

- Only one producer or consumer may access the buffer at any one time

- Ensure that the producer can't add data into full buffer and consumer can't remove data from an empty buffer

Solution to producer-consumer problem using semaphore

With a bounded buffer

The bounded buffer producer problem assumes that there is a fixed buffer size.

In this case, the consumer must wait if the buffer is empty and the producer must wait if the buffer is full.

Initialization

char item;   //could be any data type

char   buffer[n];

semaphore full = 0;   //counting semaphore for full slots

semaphore empty = n;   //counting semaphore for empty slots

semaphore mutex = 1;   //binary semaphore for mutual exclusion of buffer

char nextp, nextc;

Solution to producer-consumer problem using semaphore

Producer Process

do

{

produce an item in nextp

wait (empty); // wait until empty>0 and then decrement 'empty'

wait (mutex); //acquire lock

add nextp to buffer //add data to buffer

signal (mutex); //releaser lock

signal (full); //increment 'full'

} while (true)

Consumer Process

do {

wait(full); // wait until full>0 and then decrement 'full'

wait(mutex); //acquire lock

remove an item from buffer to nextc // //remove data from buffer

signal(mutex); //releaser lock

signal(empty); //increment 'empty'

consume the item in nextc;

} while (true)

Reader Writer Problem

- There is a data area shared among a number of processes.
- The data area could be a file or record
- There are number of processes that only read the data area(readers) and a number of processes that only write the data area (writers).
- Conditions that must be satisfied are as follows:
  - 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.

Pseudo Code reader writer: readers have priority

int readcount = 0; // keeps track of number of readers

semaphore mutex = 1, //binary, used for updating reader count

semaphore wrt = 1; // binary, common to readers & writers. Mutual exclusion for writers & is used by 1st & last reader that enters or exits CS. Not used by readers who enter or exit while other readers are in their CS

Pseudo Code readers-writers

```
void reader()
{while(true)
  {
     wait(mutex);
                                                   readcount++;
           if(readcount == 1)
wait(wrt);
      signal(mutex);
                  ………                                          reading is
performed
             ………..
     wait(mutex);
               readcount--;
           if (readcount == 0)
             signal(wrt);
      signal(mutex);
    }
 }
```

```
void writer()
{
  while(true)
  {
     wait(wrt);
                         ………
                                    writing is performed
                              ………..

     signal(wrt);
    }
 }
```

Semaphore example

The following program consists of 3 concurrent processes and 3 binary semaphores. The semaphores are initialized as S0 = 1, S1 = 0 and S2 = 0.

P0	P1	P2

while (true)
{
wait(S0);
print ‘0’
signal(S1);
signal(S2);
}
	wait (S1);
	
	signal(S0);
		wait (S2);
		
		signal(S0);

How many times will process P0 print ‘0’?

1. At least twice
2. Exactly twice
3. Exactly thrice
4. Exactly once

Semaphore example

Suppose we want to synchronize two concurrent processes P and Q using binary semaphores S1 and S2. The code for the processes P and Q is shown below-

P	**Q**

while(1)
{
P(S1);
P(S2);
Critical Section
V(S1);
V(S2);
}
	while(1)
	{
	P(S2);
	P(S1);
	Critical Section
	V(S1);
	V(S2);
	}

This leads to -

1. Mutual Exclusion
2. Deadlock
3. Both (1) and (2)
4. None of these

Synchronization with Mutex

- Mutex allows the programmer to “lock” an object so that only one thread can access it.
- To control access to a critical section of the code, programmer is required to lock a mutex before entering into a CS and then unlock the mutex while leaving the CS.
- Mutex is like a binary semaphore, but the thread which locks the mutex, can only unlock the mutex.

Synchronization with Mutex

- Strictly speaking, a mutex is **locking mechanism** used to synchronize access to a resource. Only one task (can be a thread or process) can acquire the mutex.
- It means there is ownership associated with mutex, and only the owner can release the lock (mutex)
- Semaphore is **signaling mechanism** (“I am done, you can carry on” kind of signal).

Synchronization with Monitors

- Monitors are a synchronization construct.
- Monitors contain data variables and procedures.
- Data variables cannot be directly accessed by a process.
- Monitors allow only a single process to access the variables at a time.

Dr. Vishwanath Karad
MIT WORLD PEACE UNIVERSITY | PUNE
TECHNOLOGY, RESEARCH, SOCIAL INNOVATION & PARTNERSHIPS

Deadlocks

Syllabus

Deadlock:

Principles of Deadlock, Deadlock Modeling, Deadlock Prevention, Deadlock Avoidance, Deadlock detection and recovery.

References

1. Abraham Silberschatz, Peter Baer Galvin and Greg Gagne, Operating System Concepts, WILEY,ISBN 978-1-118-06333-0, 9th Edition

Deadlocks

Finite number of resources to be distributed among a number of competing processes

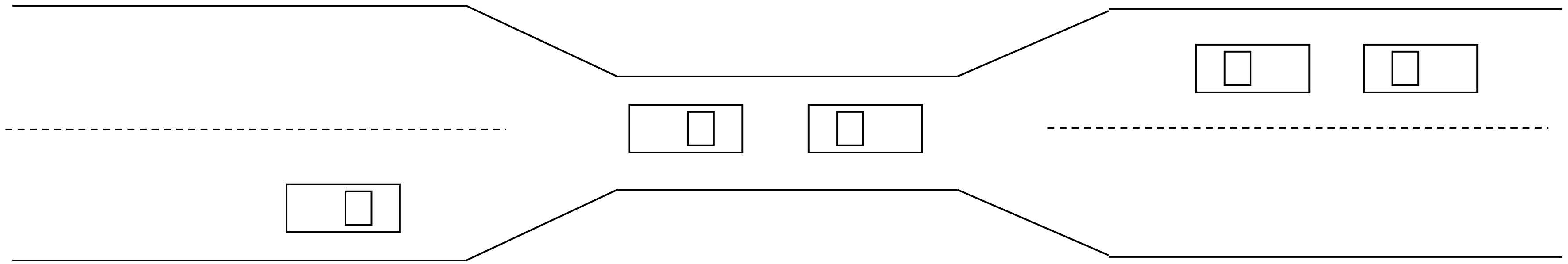
A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.

The Deadlock Problem

Example

- System has 2 tape drives.
- P_1 and P_2 each hold one tape drive and each needs another one.

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, . . ., R_m

CPU , 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 iff four conditions hold simultaneously.

Mutual exclusion: resource must be held in a nonshareable mode

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 $\{P_0, P_1, \ldots, P_n\}$ 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:

- $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.

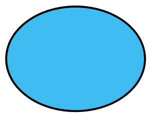
- $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.

request edge – directed edge $P_i \rightarrow R_j$

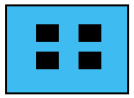
assignment edge – directed edge $R_j \rightarrow P_i$

Resource-Allocation Graph (Cont.)

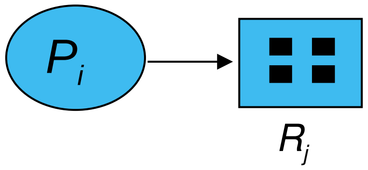
Process



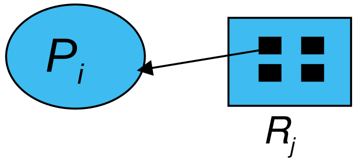
Resource Type with 4 instances



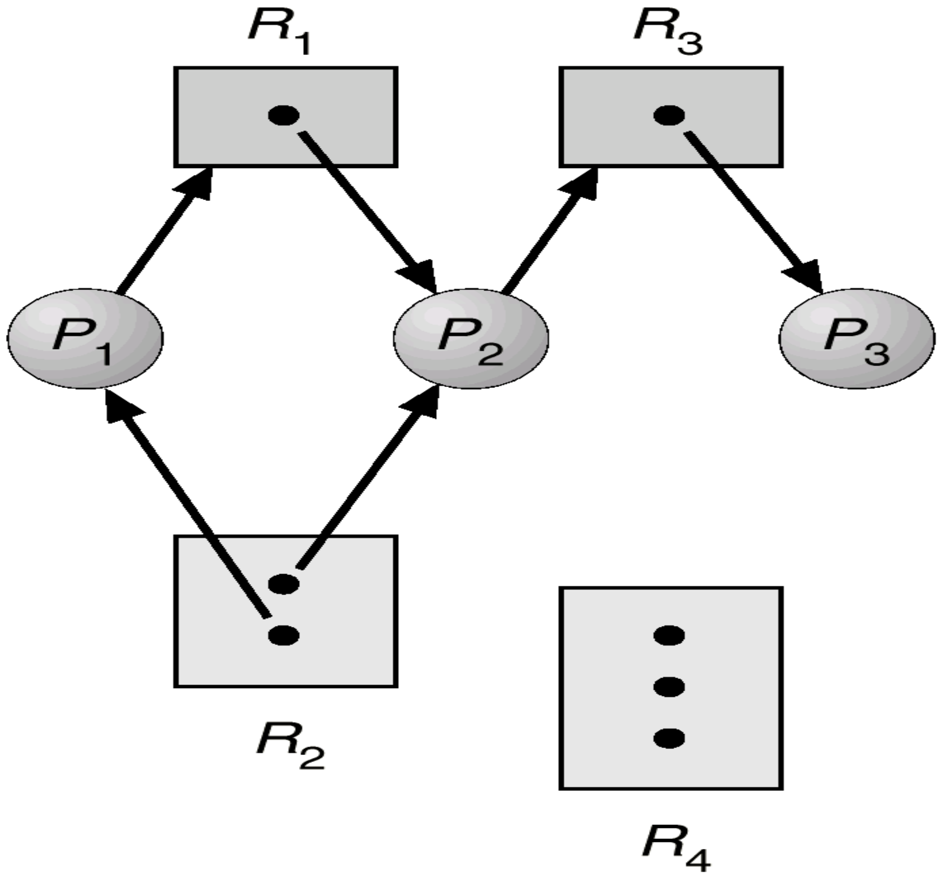
P_i requests instance of R_j



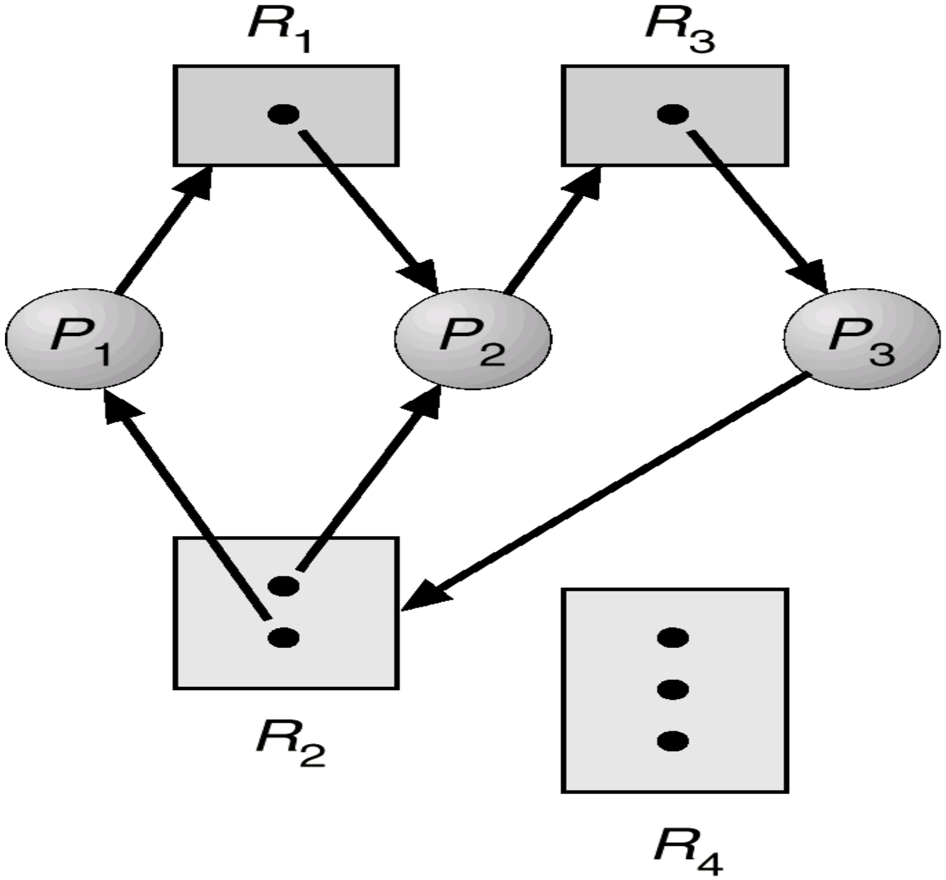
P_i is holding an instance of R_j



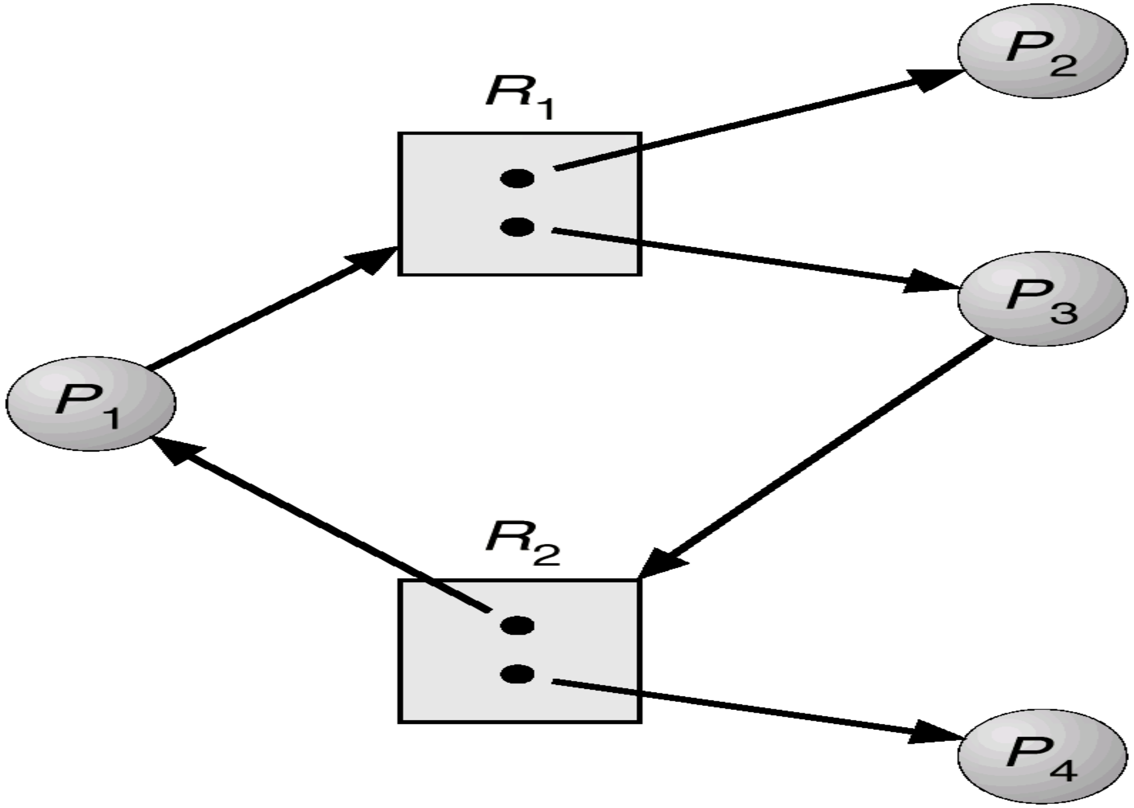
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Resource Allocation Graph With A Cycle But No Deadlock



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.
- Allow the system to enter a deadlock state and then recover
- Ignore

Deadlock Prevention

By ensuring that atleast one of the conditions cannot hold, we can prevent deadlock

1. Mutual Exclusion – some resources are inherently nonsharable eg. printer.

2. Hold and Wait –

- Require process to request and be allocated all its resources before it begins execution
- Low resource utilization; starvation possible.

Deadlock Prevention (Cont.)

3. 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.

4. Circular Wait – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

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 (eg. Disk needed before printer, so *Number assigned to printer* > *Number assigned to disk*)

$R = \{R_1, R_2, \ldots, R_m\}$ the set consisting of all resource types in the system

Assign a unique integer number to each resource type

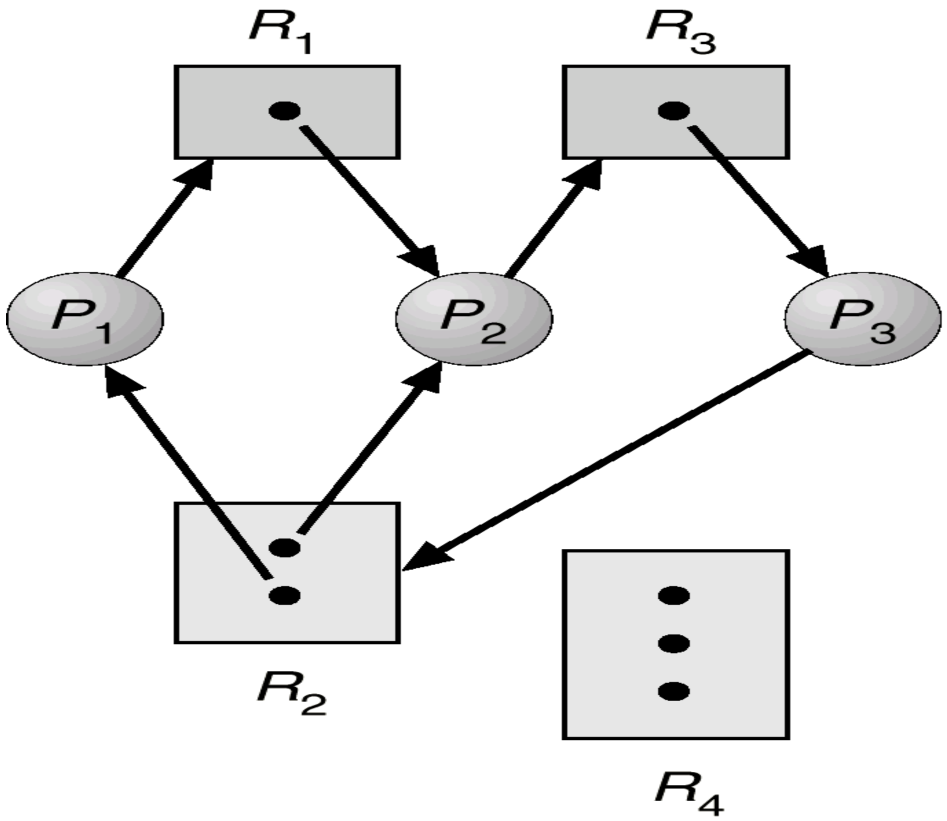
A process can initially request any number of instances of a resource type R_i

After that process can request instances of a resource type R_j iff

Number of R_j > *Number of* R_i

If several instances of some resource type are needed, a single request for all must be issued

Deadlock Prevention (Cont.)



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.

Uses concept of **safe state**

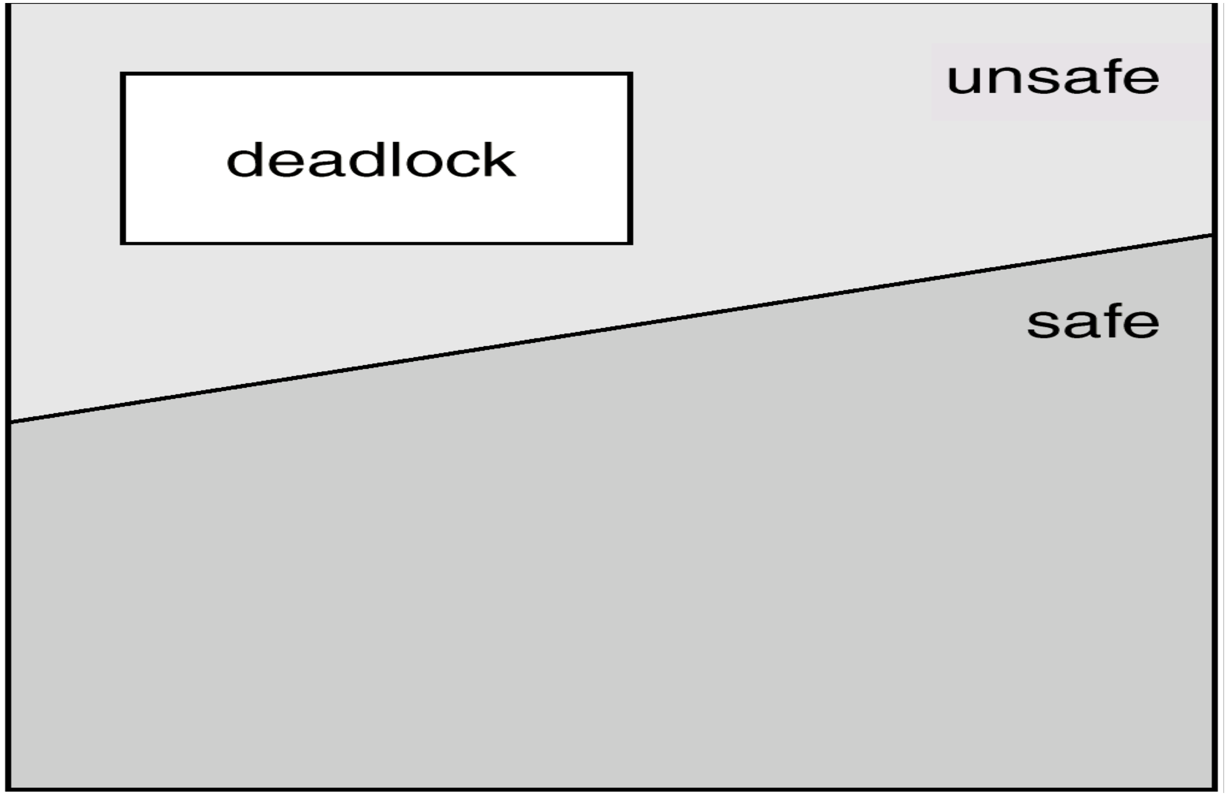
Safe State

- System is in safe state if there exists a safe sequence of all processes.
- Sequence <P_1, P_2, …, 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_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.

Safe State

- If no such sequence exists, then system is in unsafe state
- If a system is in safe state ⇒ no deadlocks.
- If a system is in unsafe state ⇒ possibility of deadlock.

Safe, unsafe , deadlock state spaces



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].

Banker's Algorithm

$Request_i$ = 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:

$Available := Available - Request_i;$

$Allocation_i := Allocation_i + Request_i;$

$Need_i := Need_i – Request_{i;;}$

- *Call safety algorithm*
- *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*

Safety Algorithm

To determine whether system is in safe state

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 an *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.

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:

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

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 $< P_1, P_3, P_4, P_0, P_2>$ or $< P_1, P_3, P_4, P_2, P_0>$ satisfies safety criteria.

m=3, n=5 Step 1 of Safety Algo

Work = Available

3	3	2

Work =

0	1	2	3	4
false	false	false	false	false

Finish =

For i = 0 Step 2

$Need_0$ = 7, 4, 3

Finish [0] is false and $\overset{7,4,3}{Need_0} > \overset{3,3,2}{Work}$

So P_0 must wait But Need $\leq$ Work

For i = 1 Step 2

$Need_1$ = 1, 2, 2

Finish [1] is false and $\overset{1,2,2}{Need_1} < \overset{3,3,2}{Work}$

So P_1 must be kept in safe sequence

Step 3

Work = $\overset{3,3,2}{Work} + \overset{2,0,0}{Allocation_1}$

Work =

A	B	C
5	3	2

Finish =

0	1	2	3	4
false	true	false	false	false

For i = 2 Step 2

$Need_2$ = 6 , 0, 0

Finish [2] is false and $\overset{6,0,0}{Need_2} > \overset{5,3,2}{Work}$

So P_2 must wait

For i=3 Step 2

$Need_3$ = 0, 1, 1

Finish [3] = false and $\overset{0,1,1}{Need_3} < \overset{5,3,2}{Work}$

So P_3 must be kept in safe sequence

Step 3

Work = $\overset{5,3,2}{Work} + \overset{2,1,1}{Allocation_3}$

Work =

A	B	C
7	4	3

Finish =

0	1	2	3	4
false	true	false	true	false

For i = 4 Step 2

$Need_4$ = 4, 3, 1

Finish [4] = false and $\overset{4,3,1}{Need_4} < \overset{7,4,3}{Work}$

So P_4 must be kept in safe sequence

Step 3

Work = $\overset{7,4,3}{Work} + \overset{0,0,2}{Allocation_4}$

Work =

A	B	C
7	4	5

Finish =

0	1	2	3	4
false	true	false	true	true

For i = 0 Step 2

$Need_0$ = 7, 4, 3

Finish [0] is false and $\overset{7,4,3}{Need} < \overset{7,4,5}{Work}$

So P_0 must be kept in safe sequence

Step 3

Work = $\overset{7,4,5}{Work} + \overset{0,1,0}{Allocation_0}$

Work =

A	B	C
7	5	5

Finish =

0	1	2	3	4
true	true	false	true	true

For i = 2 Step 2

$Need_2$ = 6 , 0, 0

Finish [2] is false and $\overset{6,0,0}{Need_2} < \overset{7,5,5}{Work}$

So P_2 must be kept in safe sequence

Step 3

Work = $\overset{7,5,5}{Work} + \overset{3,0,2}{Allocation_2}$

Work =

A	B	C
10	5	7

Finish =

0	1	2	3	4
true	true	true	true	true

Step 4

Finish [i] = true for $0 \leq i \leq n$

Hence the system is in Safe state

The safe sequence is P_1, P_3, P_4, P_0, P_2

Example (Cont.)……

At time T_1:

P_1 request (1,0,2)

$$\begin{array}{ll} & A\ B\ C \\ Request_1 = & 1, 0, 2 \end{array}$$

To decide whether the request is granted we use Resource Request algorithm

Step 1

$\overset{1,0,2}{Request_1} < \overset{1,2,2}{Need_1}$

Step 2

$\overset{1,0,2}{Request_1} < \overset{3,3,2}{Available}$

Step 3

$Available = Available - Request_1$

$Allocation_1 = Allocation_1 + Request_1$

$Need_1 = Need_1 - Request_1$

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

m=3, n=5 — Step 1 of Safety Algo

Work = Available

Work = 2 | 3 | 0

	0	1	2	3	4
Finish =	false	false	false	false	false

For i = 0 — Step 2

$Need_0$ = 7, 4, 3

Finish [0] is false and $\overset{7,4,3}{Need_0} > \overset{2,3,0}{Work}$

So P_0 must wait

But Need $\leq$ Work

For i = 1 — Step 2

$Need_1$ = 0, 2, 0

Finish [1] is false and $\overset{0,2,0}{Need_1} < \overset{2,3,0}{Work}$

So P_1 must be kept in safe sequence

Step 3

Work = $\overset{2,3,0}{Work}$ + $\overset{3,0,2}{Allocation_1}$

	A	B	C
Work =	5	3	2

	0	1	2	3	4
Finish =	false	true	false	false	false

For i = 2 — Step 2

$Need_2$ = 6 , 0, 0

Finish [2] is false and $\overset{6,0,0}{Need_2} > \overset{5,3,2}{Work}$

So P_2 must wait

For i=3 — Step 2

$Need_3$ = 0, 1, 1

Finish [3] = false and $\overset{0,1,1}{Need_3} < \overset{5,3,2}{Work}$

So P_3 must be kept in safe sequence

Step 3

Work = $\overset{5,3,2}{Work}$ + $\overset{2,1,1}{Allocation_3}$

	A	B	C
Work =	7	4	3

	0	1	2	3	4
Finish =	false	true	false	true	false

For i = 4 — Step 2

$Need_4$ = 4, 3, 1

Finish [4] = false and $\overset{4,3,1}{Need_4} < \overset{7,4,3}{Work}$

So P_4 must be kept in safe sequence

Step 3

Work = $\overset{7,4,3}{Work}$ + $\overset{0,0,2}{Allocation_4}$

	A	B	C
Work =	7	4	5

	0	1	2	3	4
Finish =	false	true	false	true	true

For i = 0 — Step 2

$Need_0$ = 7, 4, 3

Finish [0] is false and $\overset{7,4,3}{Need} < \overset{7,4,5}{Work}$

So P_0 must be kept in safe sequence

Step 3

Work = $\overset{7,4,5}{Work}$ + $\overset{0,1,0}{Allocation_0}$

	A	B	C
Work =	7	5	5

	0	1	2	3	4
Finish =	true	true	false	true	true

For i = 2 — Step 2

$Need_2$ = 6 , 0, 0

Finish [2] is false and $\overset{6,0,0}{Need_2} < \overset{7,5,5}{Work}$

So P_2 must be kept in safe sequence

Step 3

Work = $\overset{7,5,5}{Work}$ + $\overset{3,0,2}{Allocation_2}$

	A	B	C
Work =	10	5	7

	0	1	2	3	4
Finish =	true	true	true	true	true

Step 4

Finish [i] = true for $0 \leq i \leq n$

Hence the system is in Safe state

The safe sequence is P_1, P_3, P_4, P_0, P_2

Example (Cont.): P_1 request (1,0,2)

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

Example (Cont.)

At time T_2: Can request for (3,3,0) by P_4 be granted?

At time T_3: Can request for (0,2,0) by P_0 be granted?

2. Example of Banker's Algorithm

Consider the following snapshot of a system:

	Allocation	*Max*	*Available*
	A B C D	*A B C D*	*A B C D*
P_0	0 0 1 2	0 0 1 2	1 5 2 0
P_1	1 0 0 0	1 7 5 0	
P_2	1 3 5 4	2 3 5 6	
P_3	0 6 3 2	0 6 5 2	
P_4	0 0 1 4	0 6 5 6	

Answer the following questions using the banker's algorithm:

a. What is the content of the matrix *Need*?

b. Is the system in a safe state?

c. If a request from process P_1 arrives for (0,4,2,0), can the request be granted immediately?

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.
- etc…

Recovery from Deadlock: Resource Preemption

- Selecting a victim
- Rollback
- Starvation – same process may always be picked as victim, include number of rollbacks