

Operating Systems – 1BCS304

Module -III

Course Coordinator:
Mrs. Pallavi A R
Assistant professor
Dept. of CSE-DS
ATMECE, Mysuru

Background

- A cooperating process is one that can affect or be affected by other processes executing in the system.

Cooperating processes can either

directly share a logical address space
(that is, both code and data)

be allowed to share data only
through files or messages.

- Concurrent access to shared data may result in data inconsistency.
- Various mechanisms is required to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

Critical section problem

- Consider a system consisting of n processes $\{P_0, P_1, \dots, P_{n-1}\}$.
- Each process has a segment of code, called a **critical section**
- in which the process may be changing common variables, updating a table, writing a file, and so on.
- The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section.
- **The critical-section problem is to design a protocol that the processes can use to cooperate.**

Critical section problem

- The general structure of a typical process P_i is shown in below figure.
- Each process must request permission to enter its **critical section**.
- The section of code implementing this request is the **entry section**.
- The critical section may be followed by an **exit section**.
- The remaining code is the **reminder section**.

Critical section

- General structure of typical process p_i :
do{
 entry section

 critical section

 exit section

 remainder section
} while(TRUE);

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

Peterson's Solution

- This is a classic software-based solution to the critical-section problem. There are no guarantees that Peterson's solution will work correctly on modern computer architectures
- Peterson's solution provides a good algorithmic description of solving the critical-section problem.
- It illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting.

Peterson's Solution

- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.
- The processes are numbered P_0 and P_1 or P_i and P_j , where $j = 1-i$
- Peterson's solution requires the two processes to share two data items:
 int turn;
 boolean flag[2];

Peterson's Solution

- turn: The variable turn indicates whose turn it is to enter its critical section.
- Ex: if $\text{turn} == i$, then process P_i is allowed to execute in its critical section
- flag: The flag array is used to indicate if a process is ready to enter its critical section.
- Ex: if $\text{flag}[i]$ is true, this value indicates that P_i is ready to enter its critical section.

Structure of process P_i and P_j

```
do {  
    flag[i] = TRUE;  
    turn = j;  
    while (flag[j] && turn == j);  
  
    critical section  
    flag[i] = false;  
  
    remainder section  
} while (TRUE);
```

```
do {  
    flag[j] = TRUE;  
    turn = i;  
    while (flag[i] && turn == i);  
  
    critical section  
    flag[j] = false;  
  
    remainder section  
} while (TRUE);
```

Peterson's Solution

- To prove that solution is correct, then we need to show that
 - 1. Mutual exclusion is preserved
 - 2. Progress requirement is satisfied
 - 3. Bounded-waiting requirement is met

Synchronization hardware

- Software-based solutions such as Peterson's are not guaranteed to work on modern computer architectures
- Hardware features can make any programming task easier and improve system efficiency To generalize the solution(s) expressed, each process when entering their critical section must set some sort of lock, to prevent other processes from entering their critical sections simultaneously, and must release the lock when exiting their critical section, to allow other processes to proceed
- Obviously it must be possible to attain the lock only when no other process has already set a lock.
- Specific implementations of this general procedure can get quite complicated, and may include hardware solutions as outlined in this section.
- One simple solution to the critical section problem is to simply prevent a process from being interrupted while in their critical section, which is the approach taken by non preemptive kernels. Unfortunately this does not work well in multiprocessor environments, due to the difficulties in disabling and the re-enabling interrupts on all processors.
- There is also a question as to how this approach affects timing if the clock interrupt is disabled.

Synchronization hardware

- Another hardware approach to is for provide
- These atomic operations. operations are guaranteed to operate as a single instruction, without interruption. One such operation is the "Test and Set", which simultaneously sets a boolean lock variable and returns its previous value, as shown in Figures 5.3 and 5.4:

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

Figure 5.3 The definition of the TestAndSet () instruction.

```
do {
    while (TestAndSetLock(&lock))
        ; // do nothing

    // critical section

    lock = FALSE;

    // remainder section
}while (TRUE);
```

Figure 5.4 Mutual-exclusion implementation with TestAndSet ().

Synchronization hardware

- The above examples satisfy the mutual exclusion requirement, but unfortunately do not guarantee bounded waiting. If there are multiple processes trying to get into their critical sections, there is no guarantee of what order they will enter, and any one process could have the bad luck to wait forever until they got their turn in the critical section.
- Figure 5.7 illustrates a solution using test-and-set that does satisfy this requirement, using two shared data structures, boolean lock and boolean waiting[N], where N is the number of processes in contention for critical sections:

Synchronization hardware

```
int compare_and_swap(int *value, int expected, int new_value) {  
    int temp = *value;  
  
    if (*value == expected)  
        *value = new_value;  
  
    return temp;  
}
```

Figure 5.5 The definition of the `compare_and_swap()` instruction.

```
do {  
    while (compare_and_swap(&lock, 0, 1) != 0)  
        ; /* do nothing */  
  
    /* critical section */  
  
    lock = 0;  
  
    /* remainder section */  
} while (true);
```

Figure 5.6 Mutual-exclusion implementation with the `compare_and_swap()` instruction.

- Another variation on the test-and-set is an atomic swap of two boolean, as shown in Figures 5.5 and 5.6:

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);
```

Semaphore

- Semaphore is a technique to manage concurrent process by using a simple integer value known as semaphore.
- Semaphore is simply a variable which is non-negative and shared between threads. The variable is used to solve the critical section problem and to achieve process synchronization in multiprogramming environment.
- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations:
wait() and signal().

Semaphore

- **Definition of wait()**
- **wait()** -> P [from Dutch word *proberen*, which means “to test”]

```
wait (S) {  
    while S <= 0  
        ; // no-op  
        S--;  
}
```

- **Definition of signal ():**
- **signal()** -> V [from Dutch word *verhogen*, which means “to increment”]

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

Semaphore

- All modifications to the integer value of the semaphore in the wait () and signal() operations must be executed indivisibly.
- When one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

Binary Semaphore

- The value of a binary semaphore can range only between 0 and 1.
- In some systems, binary semaphores are known as mutex locks, as they are locks that provide mutual exclusion
- Binary semaphores to deal with the critical-section problem for multiple processes.

Counting semaphore

- The value of a counting semaphore can range over an unrestricted domain.
- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.
- The semaphore is initialized to the number of resources available.
- Each process that wishes to use a resource performs a wait() operation on the semaphore.
- When a process releases a resource, it performs a signal() operation.
- When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

Semaphore Implementation

- 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

- Implementation of wait:
 - wait(S){
 - value--;
 - if (value < 0) {
 - add this process to waiting queue
 - block(); }
 - }
- Implementation of signal:
 - Signal(S){
 - value++;
 - if (value <= 0) {
 - remove a process P from the waiting
 - queue
 - wakeup(P); }
 - }

Disadvantages of Semaphores

- The main disadvantage of the semaphore definition requires busy waiting.
- While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code.
- Busy waiting wastes CPU cycles that some other process might be able to use productively.
- This type of semaphore is also called a spinlock because the process "spins" while waiting for the lock.

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 P1
 - wait (Q);
 - wait (S);
 - ...
 - signal (Q); signal (S);
 - P0
 - wait (S);
 - wait (Q);
 - ...
 - signal (S); signal (Q);
- Starvation –indefinite blocking.
 - A process may never be removed from the semaphore queue in which it is suspended.

Classical Problems of Synchronization

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

Solution to BB Problem

- We will make use of 3 semaphores:
 1. m (**mutex**), a binary semaphore which is used to acquire and release the lock, initialized to the value 1
 2. **empty**, a counting semaphore whose initial value is the number of slots in the buffer, since, initially all slots are empty. i.e. initialized to the value N.
 3. **full**, a counting semaphore whose initial value is 0

Bounded-Buffer Problem

- There is a buffer of n slots and each slot is capable of storing one unit of data.
- There are two processes running, namely, Producer and Consumer, which are operating on the buffer.



- The producer tries to insert data into an empty slot of buffer.
- The consumer tries to remove data from filled slot in the buffer.
- The producer must not insert data when the buffer is full.
- The consumer must not remove data when the buffer is empty.
- Producer and Consumer should not insert and remove data simultaneously

Bounded Buffer Solution(Cont.)

- The structure of the producer process

```
do {  
wait (empty);//until empty>0  
and then decre empty  
    wait (mutex);//acquire lock  
// add the item to the buffer  
signal (mutex);//release lock  
signal (full);//incr full  
} while (true)
```

- The structure of the consumer process

```
do {  
    wait (full);//until full>0  
then decre full  
    wait (mutex);//acquire lock  
// remove an item from buffer  
    signal (mutex);//release lock  
    signal (empty);//incr empty  
  
// consume the removed item  
} while (true)
```

Readers-Writers Problem

- A database is to be shared among several concurrent processes.
- Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database.
- We distinguish between these two types of processes by referring to the former as **Readers** and to the latter as **Writers**.
- Obviously, if two readers access the shared data simultaneously, no adverse affects will result.
- However, if a writer and some other thread (either a reader or a writer) access the database simultaneously, problems may ensue.

- To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database.
- This synchronization problem is referred to as the readers-writers problem.

Solution to the Readers-Writers Problem using Semaphores

- We will make use of two semaphores and an integer variable:
 1. **mutex**, a semaphore (initialized to 1) which is used to ensure mutual exclusion when readcount is updated i.e. when any reader enters or exit from the critical section.
 2. **wrt**, a semaphore (initialized to 1) common to both reader and writer processes.
 3. **readcount**, an integer variable (initialized to 0) that keeps track of how many processes are currently reading the object.

Readers-Writers Problem (Cont.)

Copyright 2011, Morgan Kaufmann Publishers, Inc. All rights reserved. This work is derived from the book "Operating Systems: Three Easy Pieces" by Remar, et al. (2011). The book is available for free at <http://www.easypieces.org/>.

Writer Process

```
do {  
    /* writer requests for critical  
       section*/  
    wait(wrt);  
    /* performs the write */  
    // leaves the critical section  
    signal(wrt);  
} while(true);
```

Reader Process

```
do {  
wait (mutex);  
readcount++;  
if (readcount 1)  
wait (wrt);  
signal(mutex);  
  
/* current reader performs reading here*/  
wait (mutex);  
readcnt--; // a reader wants to leave  
if (readcnt == 0)//no reader is left in the critical section  
signal (wrt);// writers can enter  
signal (mutex); // reader leaves  
} while(true);
```

Dining-Philosophers Problem

Philosopher either

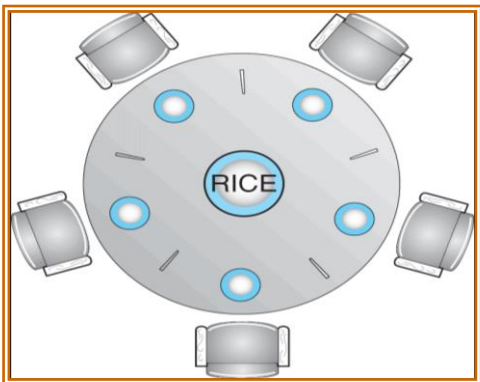
Thinks

When a philosopher thinks
He does not interact with his

Eats

When the philosopher gets hungry he
tries to pick up the 2 forks that are
colleagues
closest to him(left or right). He may pick
up only one fork at a time.
Cannot pick up a fork that is already in
the hand of a neighbor.

When a hungry philosopher has both his
forks at the same time, he eats without
releasing his forks. When he has finished
eating, he puts down both of his forks
and starts thinking again.



One simple solution is to represent each fork/chopstick with a semaphore.

- A philosopher tries to grab a fork/chopstick by executing a wait () operation on that semaphore.
- He releases his fork/chopsticks by executing the signal () operation on the appropriate semaphores.
- Thus, the shared data are :

semaphore chopstick[5];

- where all the elements of chopstick are initialized to 1.

Dining-Philosophers Problem (Cont.)

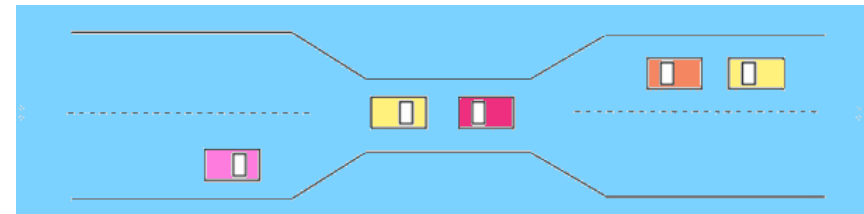
- The structure of Philosopher *i*:

```
do {  
    wait ( chopstick[i] );  
    wait ( chopstick[ (i + 1) % 5] );  
  
    // eat  
  
    signal ( chopstick[i] );  
    signal ( chopstick[ (i + 1) % 5] );  
  
    // think  
} While (true)
```

- Although this solution guarantees that no two neighbors are eating simultaneously, it could still create a deadlock.
- Suppose that all five philosophers become hungry simultaneously and each grabs their left chopstick. All the elements of chopstick will now be equal to 0.
- When each philosopher tries to grab his right chopstick, he will be delayed forever.

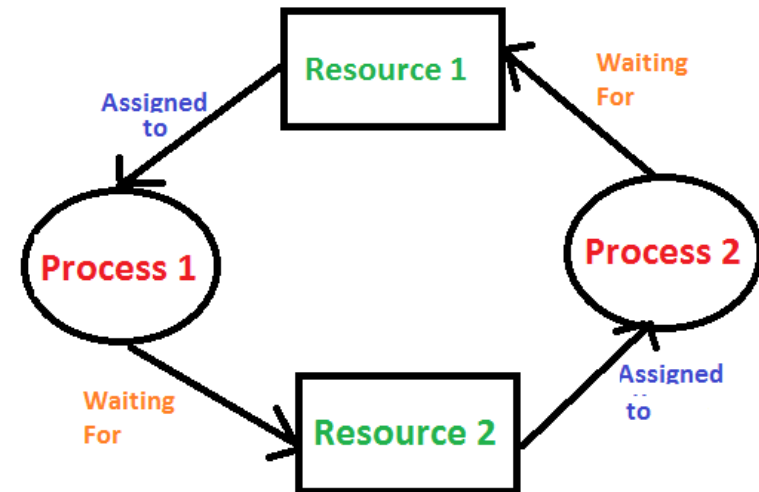
Introduction

- Deadlock is a situation where a set of processes are blocked because each
- process is holding a resource and waiting for another resource acquired by some other process.
- Consider an example when two trains are coming toward each other on the same track and there is only one track, none of the trains can move once they are in front of each other.



Introduction

- A similar situation occurs in operating systems when there are two or more processes
- that hold some resources and wait for resources held by other(s). For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.



System Model

- System consists of resources
- 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

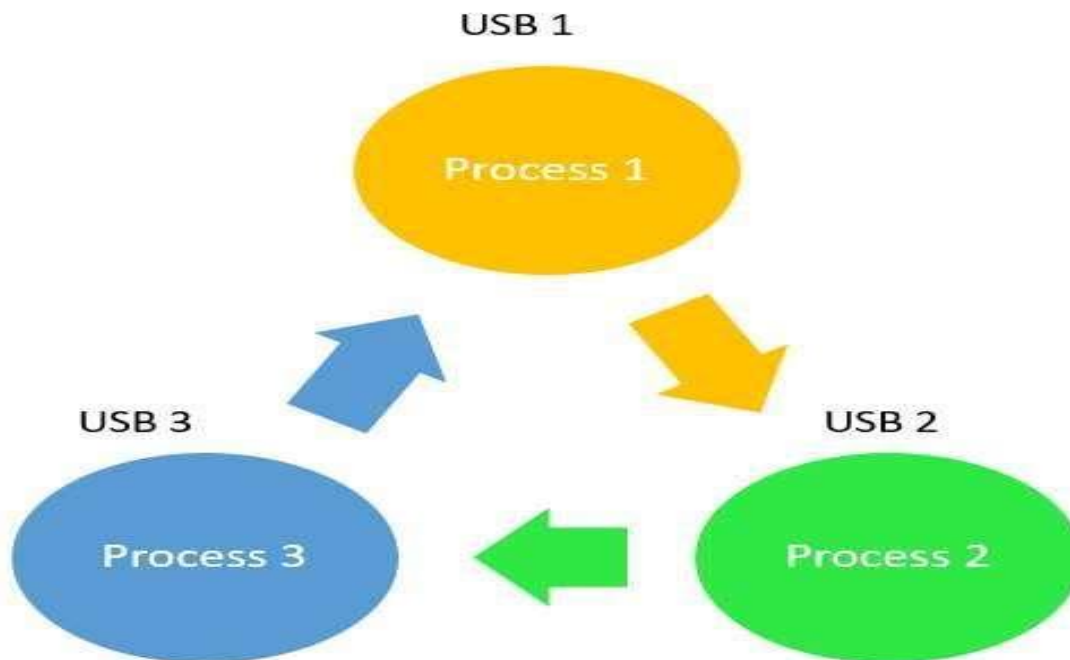
Deadlock with Semaphores

- Data:
 - A semaphore S1 initialized to 1
 - A semaphore S2 initialized to 1
- Two processes P1 and P2
- P1:
 - wait(s1)
 - wait(s2)
- P2:
 - wait(s2)
 - wait(s1)

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 .

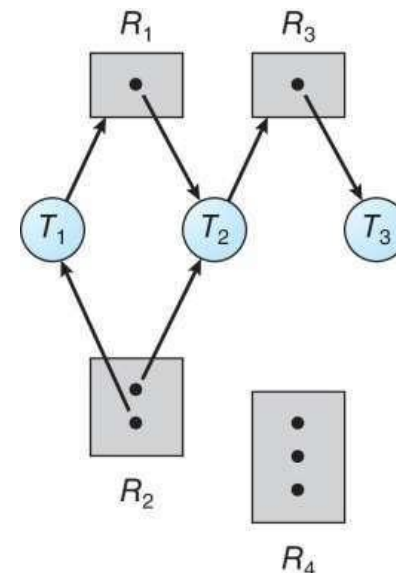
Deadlock Characterization



Resource Allocation Graph Example

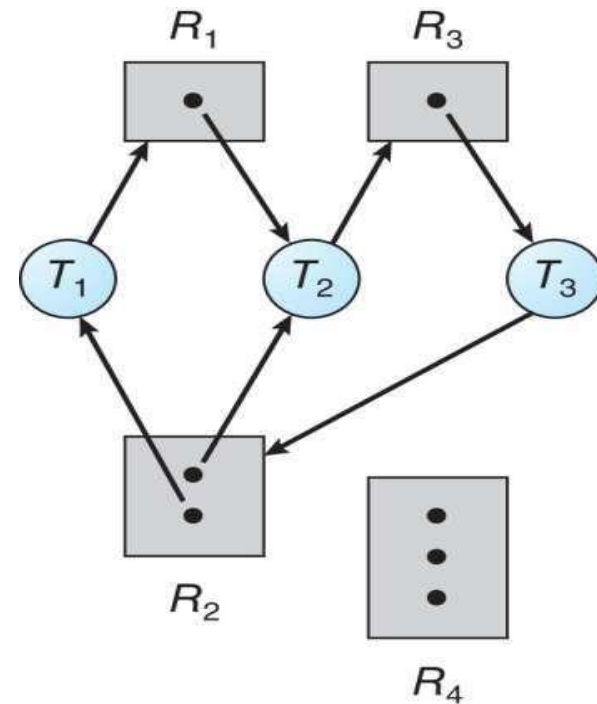
One instance of R1

- Two instances of R2
- One instance of R3
- Three instance of R4
- T1 holds one instance of R2 and is waiting for an instance of R1
- T2 holds one instance of R1, one instance of R2, and is waiting for an instance of R3
- T3 is holds one instance of R3



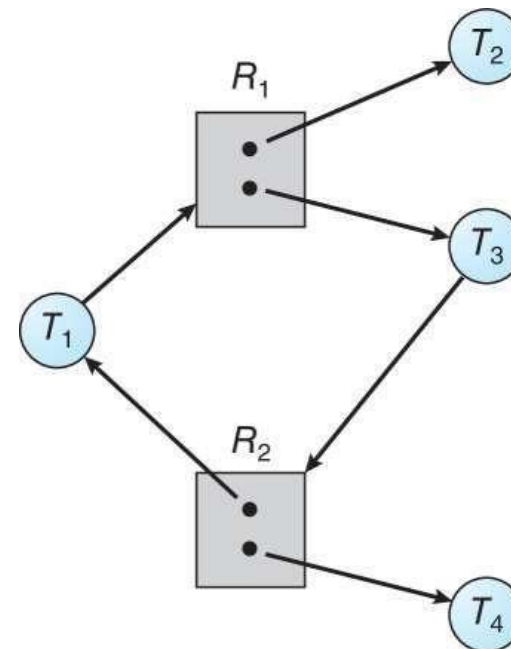
Resource Allocation Graph with a Deadlock

- $T_1 \rightarrow R_1 \rightarrow T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow P_1(T_1)$
- $T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow P_2(T_2)$



Graph with a Cycle But no Deadlock

- $T_1 \rightarrow R_1 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$



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:
 - – Deadlock prevention
 - – Deadlock avoidance
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system.

Deadlock Prevention

- Invalidate one of the four necessary conditions for deadlock:
- **Mutual Exclusion**
 - –not required for sharable resources (e.g., read-only files); must hold for non-sharable 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 allocated to it.
 - – 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

- **CircularWait:**

- – Impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration
- Invalidating the circular wait condition is most common.
- Simply assign each resource (i.e., mutex locks) a unique number.
- Resources must be acquired in order.
- If:
 first_mutex = 1
 second_mutex = 5

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

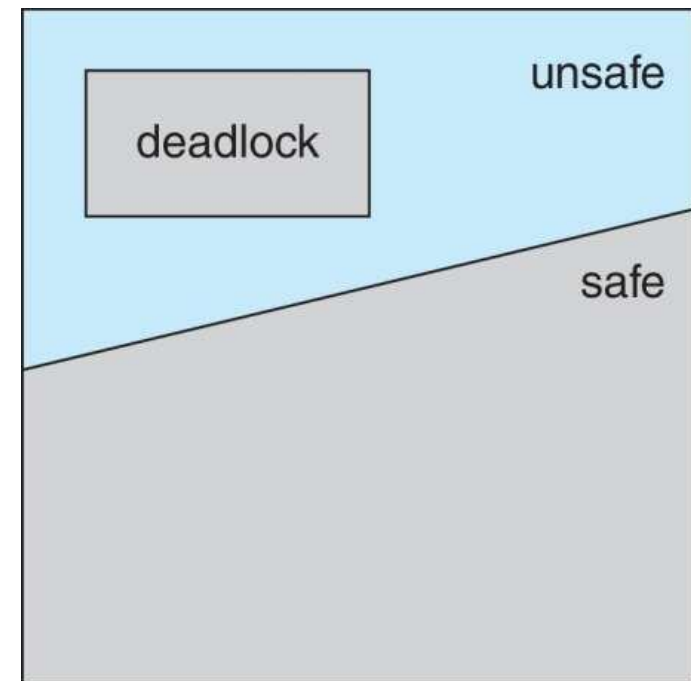
Safe State

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

5 processes P0 through P4;
3 resource types: A, B, and C • Snapshot at time T0:
Allocation Max Available
A B C A B C A B C P0 1 1 2
4 3 3 2 1 0 P1 2 1 2 3 2 2
P2 4 0 1 9 0 2 P3 0 2 0 7 5
3 P4 1 1 2 1 1 2



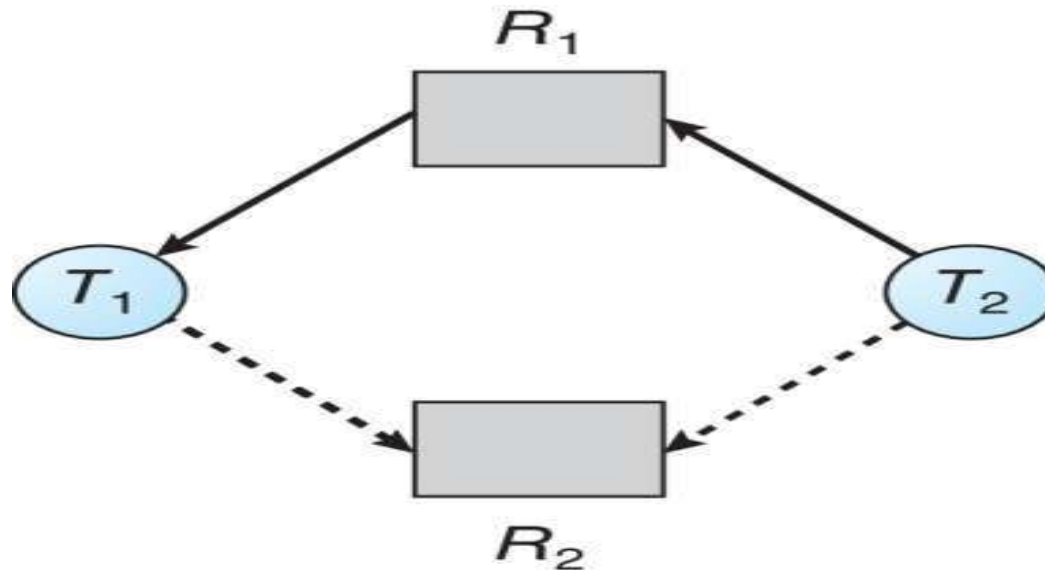
Avoidance Algorithms

- Single instance of a resource type
 - Use a resource-allocation graph
- Multiple instances of a resource type
 - Use the Banker's Algorithm

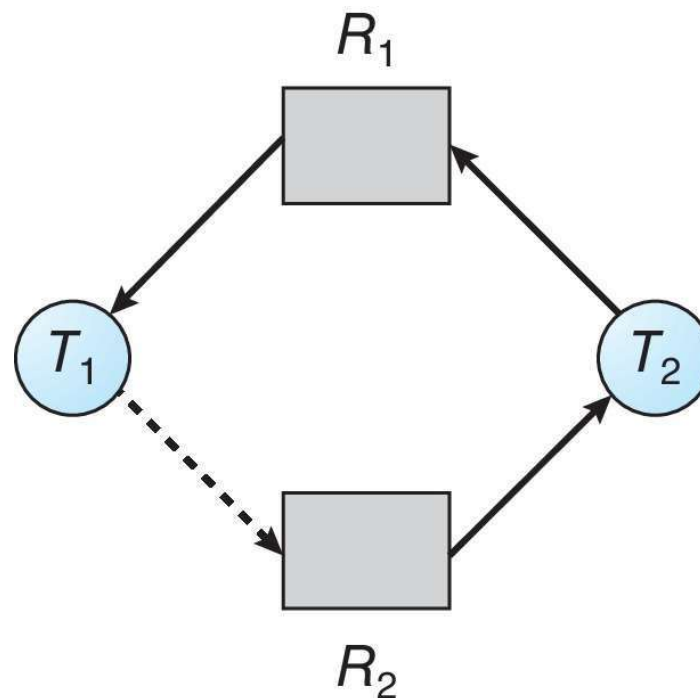
Resource-Allocation Graph Scheme

- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j 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 of resources
- 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 $\text{available}[j] = k$, there are k instances of resource type R_j available
- **Max:** $n \times m$ matrix. If $\text{Max}[i,j] = k$, then process P_i may request at most k instances of resource type R_j
- **Allocation:** $n \times m$ matrix. If $\text{Allocation}[i,j] = k$ then P_i is currently allocated k instances of R_j
- **Need:** $n \times m$ matrix. If $\text{Need}[i,j] = k$, then P_i may need k more instances of R_j to complete its task
- $\text{Need}[i,j] = \text{Max}[i,j] - \text{Allocation}[i,j]$

Safety Algorithm

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

Initialize:

Work = Available

Finish [i] = false for $i = 0, 1, \dots, n-1$

2. Find an i such that both:

(a) Finish [i] = false

(b) $\text{Need}_i \leq \text{Work}$

If no such i exists, go to step 4

3. $\text{Work} = \text{Work} + \text{Allocation}_i$

Finish[i] = true

go to step 2

4. If Finish [i] == true for all i, then the system is in a safe state

Resource-Request Algorithm for Process P_i

- $Request_i$ = request vector for process P_i .
- 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

Example: Consider a system that contains five processes P1, P2, P3, P4, P5 and the three resource types A, B and C. Following are the resources types: A has 10, B has 5 and the resource type C has 7 instances.

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P1	0	1	0	7	5	3	3	3	2
P2	2	0	0	3	2	2			
P3	3	0	2	9	0	2			
P4	2	1	1	2	2	2			
P5	0	0	2	4	3	3			

Answer the following questions using the banker's algorithm:

1. What is the reference of the need matrix?
2. Determine if the system is safe or not.
3. What will happen if the resource request (1, 0, 0) for process P1 can the system accept this request immediately?

Example of Banker's Algorithm

Ans. 2: Context of the need matrix is as follows:

Need [i] = Max [i] - Allocation [i]

Need for P1: (7, 5, 3) - (0, 1, 0) = 7, 4, 3

Need for P2: (3, 2, 2) - (2, 0, 0) = 1, 2, 2

Need for P3: (9, 0, 2) - (3, 0, 2) = 6, 0, 0

Need for P4: (2, 2, 2) - (2, 1, 1) = 0, 1, 1

Need for P5: (4, 3, 3) - (0, 0, 2) = 4, 3, 1

Process	Need		
	A	B	C
P1	7	4	3
P2	1	2	2
P3	6	0	0
P4	0	1	1
P5	4	3	1

Hence, we created the context of need matrix.

Example of Banker's Algorithm

Ans. 2: Apply the Banker's Algorithm:

Available Resources of A, B and C are 3, 3, and 2.

Now we check if each type of resource request is available for each process.

Step 1: For Process P1:

Need $\leq$ Available

7, 4, 3 $\leq$ 3, 3, 2 condition is **false**.

So, we examine another process, P2.

Step 2: For Process P2:

Need $\leq$ Available

1, 2, 2 $\leq$ 3, 3, 2 condition **true**

New available = available + Allocation

(3, 3, 2) + (2, 0, 0) $\Rightarrow$ 5, 3, 2

Example of Banker's Algorithm

Similarly, we examine another process P3.

Step 3: For Process P3:

P3 Need $\leq$ Available

6, 0, 0 $\leq$ 5, 3, 2 condition is **false**.

Similarly, we examine another process, P4.

Step 4: For Process P4:

P4 Need $\leq$ Available

0, 1, 1 $\leq$ 5, 3, 2 condition is **true**

New Available resource = Available + Allocation

5, 3, 2 + 2, 1, 1 $\Rightarrow$ 7, 4, 3

Similarly, we examine another process P5.

Example of Banker's Algorithm

Step 5: For Process P5:

$P5 \text{ Need} \leq \text{Available}$

$4, 3, 1 \leq 7, 4, 3$ condition is **true**

$\text{New available resource} = \text{Available} + \text{Allocation}$

$7, 4, 3 + 0, 0, 2 \Rightarrow 7, 4, 5$

Now, we again examine each type of resource request for processes P1 and P3.

Step 6: For Process P1:

$P1 \text{ Need} \leq \text{Available}$

$7, 4, 3 \leq 7, 4, 5$ condition is **true**

$\text{New Available Resource} = \text{Available} + \text{Allocation}$

$7, 4, 5 + 0, 1, 0 \Rightarrow 7, 5, 5$

Example of Banker's Algorithm

So, we examine another process P2.

Step 7: For Process P3:

P3 Need $\leq$ Available

6, 0, 0 $\leq$ 7, 5, 5 condition is true

New Available Resource = Available + Allocation

7, 5, 5 + 3, 0, 2 $\Rightarrow$ 10, 5, 7

Hence, we execute the banker's algorithm to find the safe state and the safe sequence like P2, P4, P5, P1 and P3.

Ans. 3: For granting the Request (1, 0, 2), first we have to check that **Request $\leq$ Available**, that is (1, 0, 2) $\leq$ (3, 3, 2), since the condition is true. So the process P1 gets the request immediately.

Example of Banker's Algorithm

- 5 processes P0 through P4;
- 3 resource types: A, B , and C
- Snapshot at time T0:

	<u>Allocation</u>			<u>Max</u>			<u>Available</u>		
	A	B	C	A	B	C	A	B	C
P0	1	1	2	4	3	3	2	1	0
P1	2	1	2	3	2	2			
P2	4	0	1	9	0	2			
P3	0	2	0	7	5	3			
P4	1	1	2	1	1	2			

Example of Banker's Algorithm

- i) To find Need Matrix

P0 3 2 1

P1 1 1 0

P2 5 0 1

P3 7 3 3

P4 0 0 0

Example of Banker's Algorithm

- ii). Check Safe State or not Safe Sequence
- For Process P0
- Check that Need $\leq$ Available (that is, $(3, 2, 1) \leq (2, 1, 0)$ $\square$ false
- For Process P1
- Check that Need $\leq$ Available (that is, $(1, 1, 0) \leq (2, 1, 0)$ $\square$ true
- New Available = available + allocation
- = $2, 1, 0 + 2, 1, 2 = 4, 2, 2$
- For Process P2
- Check that Need $\leq$ Available (that is, $(5, 0, 1) \leq (4, 2, 2)$ $\square$ false
- For Process P3
- Check that Need $\leq$ Available (that is, $(7, 3, 3) \leq (4, 2, 2)$ $\square$ false

Example of Banker's Algorithm

- For Process P4

Check that Need $\leq$ Available (that is, $(0,0,0) \leq (4,2,2)$ $\square$ true

New Available = available + allocation

$$= 4, 2, 2 + 1, 1, 2 = 5, 3, 4$$

Now again Check for Process P2

Check that Need $\leq$ Available (that is, $(5, 0, 1) \leq (5, 3, 4)$ $\square$ true

New Available = available + allocation

$$= 5, 3, 4 + 4, 0, 1 = 9, 3, 5$$

Now again Check for Process P3

Check that Need $\leq$ Available (that is, $(7, 3, 3) \leq (9, 3, 5)$ $\square$ true

New Available = available + allocation

$$= 9, 3, 5 + 0, 2, 0 = 9, 5, 5$$

Example of Banker's Algorithm

- Now again Check for Process P0
- Check that Need $\leq$ Available (that is, $(3, 2, 1) \leq (9, 5, 5) \square \text{true}$)
- Executing safety algorithm shows that sequence $\langle P1, P4, P2, P3, P0 \rangle$ satisfies safety requirement
- iii). Determine the total amount of resources of each type
 - TotalAmount of Resources= sum of columns of Allocation + Available
 - $= 8, 5, 7 + 2, 1, 0$
 - $= 10, 6, 7$
- iV). What will happen if process P1 requests one additional instance of resource type A and two instances of resource type C?
- Check that Request $\leq$ Available (that is, $(1, 0, 2) \leq (2, 1, 0) \square \text{false}$)
- So, request cannot be granted, since the resources are not available.

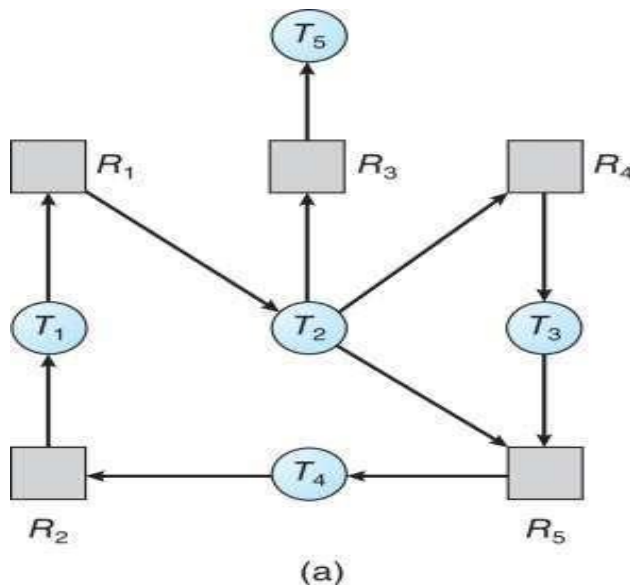
Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

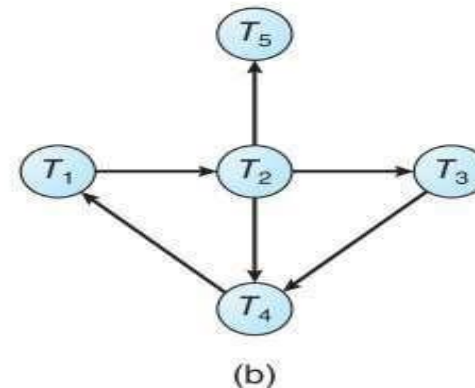
Single Instance of Each Resource Type

- Maintain wait-for graph
 - – Nodes are processes
 - – $P_i \square P_j$
 - 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] = 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

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 P0 through P4; three resource types
- A (7 instances), B (2 instances), and C (6 instances)
- Snapshot at time T0:

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	0	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

Sequence <P0, P2, P3, P1, P4> will result in Finish[i] = true for all i

Example of Detection Algorithm

P2 requests an additional instance of type C

	Request		
	A	B	C
P0	0	0	0
P1	2	0	2
P2	0	0	1
P3	1	0	0
P4	0	0	2

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
 - How often a deadlock is likely to occur?
 - How many processes will need to be rolled back?
- one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock

Recovery from Deadlock: Process Termination

Abort all deadlocked processes

- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?

1. Priority of the process
2. How long process has computed, and how much longer to completion
3. Resources the process has used
4. Resources process needs to complete
5. How many processes will need to be terminated
6. 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