SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING
Common to : CSE , IT

UNIT III

UNIT 3 SYNCHRONIZATION AND DEADLOCKS 9 Hrs.

The critical section problem - Semaphores - Classic problems of synchronization - Critical regions - Monitors-Dead locks - Deadlock characterization - Prevention - Avoidance - Detection - Recovery.

INTRODUCTION

A cooperating process is one that can affect or be affected by other processes executing in the system. Co- operating processes can either directly share a logical address space or be allowed to share data only through files or messages. The former is achieved through the use of light weight processes or threads. Concurrent access to shared data may result in data inconsistency.

A bounded buffer could be used to enable processes to share memory. While considering the solution for producer – consumer problem, it allows at most BUFFER_SIZE – 1 items in the buffer. If the algorithm can be modified to remedy this deficiency, one possibility is to add an integer variable counter initialized to 0. Counter is incremented every time a new item is added to the buffer and is decremented every time, an item is removed from the buffer. Although both the producer and consumer routines are correct separately, they may not function correctly when executed concurrently.

The code for the producer process can be modified as

```
follows: while (true)
{
/* produce an item in nextProduced */
while (counter == BUFFER.SIZE)
; /* do nothing */
buffer[in] = nextProduced;
in = (in + 1) % BUFFER-SIZE;
counter++;
```

The code for the consumer process can be modified as follows:

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

Although both the producer and consumer routines are correct separately, they may not function correctly when executed concurrently. As an illustration, suppose that the value of the variable counter is currently 5 and that the producer and consumer processes execute the statements "counter++" and "counter—" concurrently. Following the execution of these two statements, the value of the variable counter may be 4, 5, or 6! The only correct result, though, is counter == 5, which is generated correctly if the producer and consumer execute separately. We can show that the value of counter may be incorrect as follows. Note that the statement "counter++" may be implemented in machine language (on a typical machine) as

*register*1 = counter
register1 = *register1* + 1
counter = *register1*
where *register1* is a local CPU register.

Similarly, the statement "counter—" is
implemented as follows:
register2 = counter
register2 = *register2* — 1
counter = *register2*
where again *register2* is a local CPU register.

The concurrent execution of "counter++" and "counter—" is equivalent to a sequential execution where the lower-level statements presented previously are interleaved in some arbitrary order (but the order within each
high-level statement is preserved). One such interleaving is

T0: producer execute	*register1* = counter *{registeri = 5}*
T1: producer execute	*register 1* = *register1* + 1 *{registeri* = 6}
T2: consumer execute	*register2.* = counter *{register2* — 5}
T3: consumer execute	*register2* = *register2* — 1 *{register 2* — 4}
T4: producer execute	counter = *register1 {counter* = 6}
T5: consumer execute	counter = *register2 {counter* = 4}

When several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place is called **race condition**. To guard against the race condition, we need to ensure that only one process at a time can be manipulating the variable counter. Hence processes must be synchronized.

The Critical Section Problem

Each process in a system has a segment of code called a critical section in which the process may be changing common variables, updating a table, writing a file etc. 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 that is no two processes are executing in their critical sections at the same time. The critical section problem is to design a protocol that the processes can use to co- operate. 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 remainder section.

```
do
{
Entry Section
```

```
    Critical
Section
Exit Section
    Remainder Section
}while(TRUE);
```

General Structure of a typical process P_i.

A critical section is a piece of code that only one thread can execute at a time. If multiple threads try to enter a critical section, only one can run and the others will sleep. Imagine you have three threads that all want to enter a critical section. Only one thread can enter the critical section; the other two have to sleep. When a thread sleeps, its execution is paused and the OS will run some other thread. Once the thread in the critical section exits, another thread is woken up and allowed to enter the critical section.

A solution to the critical section problem must satisfy the following three requirements:

Mutual exclusion: If a process is executing in critical section, then no other process can be executing in their critical section.

Progress: If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next and this selection cannot be postponed indefinitely.

Bounded waiting: There exists a bound or limit on the number of times that other processes are allowed to enter their critical section after a process has made a request to enter its critical section and before that request is granted.

Two Process
Solutions Algorithm
1:

```
do {
```

while (turn !=1); critical

```
section turn = j;
remainder section
}while (1);
```

This Algorithm satisfies Mutual exclusion whereas it fails to satisfy progress requirement since it requires strict alternation of processes in the execution of the critical section. For example, if turn == 0 and p1 is ready to enter its critical section,p1 cannot do so, even though p0 may be in its remainder section.

Algorithm 2:

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

In this solution, the mutual exclusion is met. But the progress is not met. To illustrate this problem, we consider the following execution sequence:
To: P0 sets Flag[0] = true T1: P1
sets Flag[1] = true
Now P0 and P1 are looping forever in their respective while statements.

Mutual exclusion

P0 and P1 can never be in the critical section at the same time: If P0 is in its critical section, then flag [0] is true and either flag [1] is false (meaning P1 has left its critical section) or turn is 0 (meaning P1 is just now trying to enter the critical section, but graciously waiting). In both cases, P1 cannot be in critical section when P0 is in critical section.

Progress

A process cannot immediately re-enter the critical section if the other process has set its flag to say that it would like to enter its critical section.

Bounded waiting

A process will not wait longer than one turn for entrance to the critical section: After giving priority to the other process, this process will run to completion and set its flag to 0, thereby allowing the other process to enter the critical section.

Algorithm 3:

By combining the key ideas of algorithm 1 and 2, we obtain a correct solution.

```
do{
Flag[i] =true; Turn =
j;
While(flag[j] && turn==j);
Critical section
Flag[i] = false;
Remainder section
}while(1);
```

The algorithm does satisfy the three essential criteria to solve the critical section problem. The three criteria are mutual exclusion, progress, and bounded waiting.

Synchronization Hardware

Any solution to the critical section problem requires a simple tool called a lock. Race conditions are prevented by requiring that critical regions are protected by locks that is a process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section.

```
do

{
```

```
Acquire Lock
        Critical Section Release Lock
Remainder Section
}while(TRUE);
```

Solution to the critical-section problem using locks.

Hardware features can make any programming task easier and improve system efficiency. The critical section problem can be solved simply in a uni processor environment if we could prevent interrupts from occurring while a shared variable was being modified. Then we can ensure that the current sequence of instructions would be allowed to execute in order without pre emption. No other instructions would be run, so no unexpected modifications could be made to the shared variable. This solution is not feasible in a multi processor environment. Disabling interrupts on a multi processor can be time consuming, as the message is passed to all the processors.

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

The definition of the TestAndSet() instruction.

```
do
{
while(TestAndSetLock(&lock))

;  // do nothing
// critical section lock = FALSE;
//remainder section
}while(TRUE);
```
Mutual-Exclusion Implementation with TestAndSet()

This message passing delays entry into each critical section and system efficiency decreases. Many modern computer systems provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words **atomically** that is as one uninterruptable unit.
The TestAndSet () instruction can be defined as above. This instruction is executed atomically. Thus, if two TestAndSet () instructions are executed simultaneously each on a different CPU, they will be executed sequentially in some order.

The Swap () instruction operates on the contents of two words – Void
```
swap(boolean *a, boolean *b)
{
boolean temp = *a;
*a = *b;
*b = temp;
}
```
Definition of Swap Instruction

```
do
{
key=TRUE; while(key==TRUE)

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

Mutual Exclusion Implementation with the Swap() function.

This is also executed atomically. If the machine supports Swap() instruction, then mutual exclusion can be provided by using a global Boolean variable lock initialized to false. Each process has a local Boolean variable key.
But these algorithms do not satisfy the bounded – waiting requirement.

The below algorithm satisfies all the critical section problems: Common data structures used in this algorithm are: Boolean

```
waiting[n];
Boolean lock;
```

Both these data structures are initialized to false. For proving that the mutual exclusion requirement is met, we must make sure that process Pi can enter its critical section only if either waiting[i] == false or key == false. The value of key can become false only if the TestAndSet() is executed.

Semaphores

The various hardware based solutions to the critical section problem are complicated for application programmers to use. To overcome this difficulty, we can use a synchronization tool called a semaphore. A semaphore S is an integer variable that is accessed only through standard atomic operations: wait() and signal().

```
Wait:                     Signal:
Wait(S)                   signal(S)
{                         {
While S<=0;               S++; S--; }
}
```

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.

```
    do{
            waiting(mutex);
                    //critical section
            Signal(mutex);
                    //remainder section
            }while(true);
```

Usage

OS's distinguish between counting and binary semaphores. The value of a counting semaphore can range over an unrestricted domain. The value of a binary semaphore can range only between 0 and 1. Binary semaphores are known as mutex locks as they are locks that provide mutual exclusion.

Binary semaphores are used to deal with the critical section problem for multiple processes. 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.

Semaphores can be used to solve various synchronization problems. Simple binary semaphore example with two processes: Two processes are both vying for the single semaphore. Process A performs the acquire first and therefore is provided with the semaphore. The period in which the semaphore is owned by the process is commonly called a *critical section*. The critical section can be performed by only one process, therefore the need for the coordination provided by the semaphore. While Process A has the semaphore, Process B is not permitted to perform its critical section.

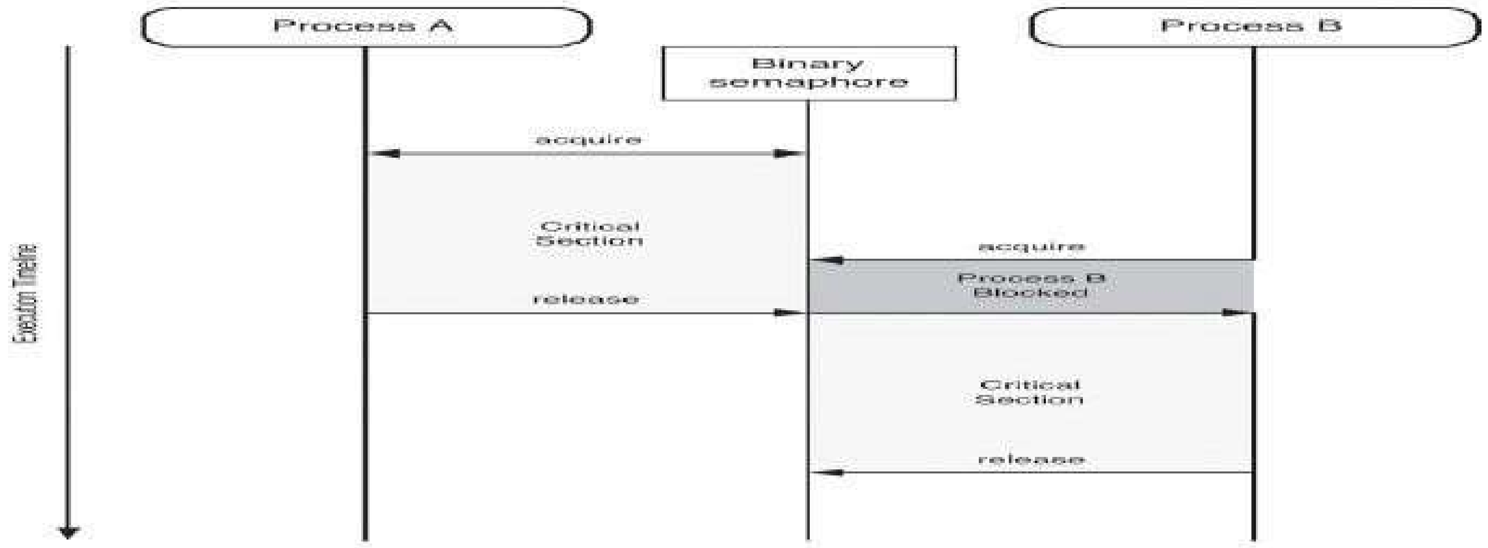


Fig.3.1. Binary Semaphore

Note that while Process A is in its critical section, Process B attempts to acquire the semaphore. As the semaphore has already been acquired by Process A, Process B is placed into a blocked state. When Process A finally releases the semaphore, it is then granted to Process B, which is allowed to enter its critical section. Process B at a later time releases the semaphore, making it available for acquisition.

In the counting semaphore example, each process requires two resources before being able to perform its desired activities. In this example, the value of the counting semaphore is 3, which means that only one process will be permitted to fully operate at a time. Process A acquires its two resources first, which means that Process B blocks until Process A releases at least one of its resources.

Fig.3.2. Counting Semaphore

Implementation

The main disadvantage of the semaphore is that it 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 called a **spinlock** because the process spins while waiting for the lock. To overcome, the need for busy waiting the definition of wait () and signal() semaphore operations can be modified. When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait. Rather than engaging in busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore and the state of the process is switched to the waiting state. Then control is transferred to CPU scheduler which selects another process to execute.

A process that is blocked waiting on a semaphore S, should be restarted when some other process executes a signal() operation. The process is restarted by a wakeup() operation which changes the process from the waiting state to the ready state. Process is then placed in the ready queue.

To implement semaphores under this definition, we define a semaphore as a "C" struct:

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

Each semaphore has an integer value and a list of processes list. When a process must wait on a semaphore, it is added to the list of processes. A signal () operation removes one process from the list of waiting processes and awakens that process. A signal() operation removes one process from the list of waiting processes and awakens that process.

Wait() semaphore operation can be defined
as wait(semaphore *S) { S-

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

Signal() semaphore operation can be defined as

```
signal(semaphore *S) {
```

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

The block operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process
P. These two operations are provided by the operating system as basic system calls. Note that, although under the classical definition of semaphores with busy waiting the semaphore value is never negative, this implementation may have negative semaphore values. If the semaphore value is negative, its magnitude is the number of processes waiting on that semaphore. This fact results from switching the order of the decrement and the test in the implementation of the wait() operation. The list of waiting processes can be easily implemented by a link field in each process control block (PCB). Each semaphore contains an integer value and a pointer to a list of PCBs. One way to add and remove processes from the list in a way that ensures bounded waiting is to use a FIFO queue, where the semaphore contains both head and tail pointers to the queue. In general, however, the list can use *any* queueing strategy. Correct usage of semaphores does not depend on a particular queueing strategy for the semaphore lists. The critical aspect of semaphores is that they be executed atomically- We must guarantee that no two processes can execute wait() and signal() operations on the same semaphore at the same time. This is a critical-section problem; and in a single-processor environment (that is, where only one CPU exists), we can solve it by simply inhibiting interrupts during the time the wait () and signal () operations are executing. This scheme works in a single processor environment because, once interrupts are inhibited, instructions from different processes cannot be interleaved. Only the currently running process executes until interrupts are reenabled and the scheduler can regain control.
In a multiprocessor environment, interrupts must be disabled on every processor; otherwise, instructions from different processes (running on different processors) may be interleaved in some arbitrary way. Disabling interrupts on every processor can be a difficult task and furthermore can seriously diminish performance. Therefore, SMP systems must provide alternative locking techniques— such as spinlocks—to ensure that wait() and signal() are performed atomically. It is important to admit that we have not completely eliminated busy waiting with this definition of the wait() and signal() operations. Rather, we have removed busy waiting from the entry section to the critical sections of application programs.

Deadlocks and Starvation

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a signal() operation. When such a state is reached, these processes are said to be **deadlocked.**

To illustrate this, we consider a system consisting of two processes, P0 and P1, each accessing two semaphores, S and Q, set to the value 1:

```
P0                                P1
wait(S);                         wait(Q);
wait(Q);                         wait(S);
signal(S);                       signal(Q);
signal(Q);                       signal(S);
```

Suppose that *P0* executes wait (S) and then P1 executes wait (Q). When Po executes wait(Q), it must wait until P1 executes signal(Q). Similarly, when P1 executes wait(S), it must wait until Po executes signal(S). Since these signal () operations cannot be executed, Po and P1 are deadlocked.

We say that a set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are *resource acquisition andrelease.* Another problem related to deadlocks is **indefinite blocking,** or **starvation,** a situation in which processes wait indefinitely within the semaphore. Indefinite blocking may occur if we add and remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

Classic problems of synchronization

These synchronization problems are examples of large class of concurrency control problems. In solutions to these problems, we use semaphores for synchronization.

The Bounded Buffer problem

Here the pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n, the semaphore full is initialized to value 0.

The code below can be interpreted as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

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

The structure of the Producer

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

The structure of the Consumer Process

The Readers – Writers Problem

A data base is to be shared among several concurrent processes. Some of these processes may want only to read the database (readers) whereas others may want to update the database (writers). If two readers access the shared data simultaneously, no adverse effects will result. If a writer and some other thread, access the database simultaneously, problems occur.

To prevent these problems, writers have exclusive access to the shared database. This synchronization problem is referred to as readers – writers problem.

```
do {
    wait(wrt);

    //Writing is Performed

    signal(wrt);
     }while(True);
```

The Structure of a Writer Process

The simplest readers writers problem requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared object. No reader should wait for other readers to finish simply because a writer is waiting.

```
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);
```

The Structure of a Reader Process

The second readers writers problem requires that once a writer is ready, that writer performs its write as soon as possible, that is if a writer is waiting to access the object, no new readers may start reading.

In the solution to the first readers – writers problem, the reader processes share the following data structures: Semaphore
mutex, wrt;
int readcount;

Semaphores mutex and wrt are initialized to 1, readcount is initialized to 0. Semaphore wrt is common to both reader and writer processes. The mutex semaphore is used to ensure mutual exclusion when the variable readcount is updated. The readcount variable keeps track of how many processes are currently reading the object. The semaphore wrt functions as a mutual exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical section.

The readers – writers problem and its solutions has been generalized to provide reader – writer locks on some systems. Acquiring the reader – writer lock requires specifying the mode of the lock, either read or write access. When a process only wishes to read shared data, it requests the reader – writer lock in read mode; a process wishing to modify the shared data must request the lock in write mode. Multiple processes are permitted to concurrently acquire a reader – writer lock in read mode, only one process may acquire the lock for writing as exclusive access is required for writers.

Reader – writer locks are useful in the following situations:

- In applications where it is easy to identify which processes only read shared data and which threads only write shareddata.
- In applications that have more readers than writers.

Dining Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chop sticks. When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her. A philosopher may pick up only one chopstick at a time. She cannot pick up a chopstick that is already in the hand of the neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chop sticks. When she is finished eating, she puts down both of her chop sticks and starts thinkingagain.

The dining philosopher's problem is considered a classic synchronization problem as it an example of a large class of concurrency control problems. One simple solution is to represent each chop stick with a semaphore.

Fig.3.3 Dining Philosopher

A philosopher tries to grab a chop stick by executing a wait () operation on that semaphore; she releases her chop sticks by executing the signal () operation on the appropriate semaphores. Thus, the shared data are Semaphore chopstick [5]; where all elements of chopstick are initialized to 1.This solution is rejected as it could create a dead lock. Suppose that all five philosophers become hungry simultaneously and each grabs her left chop stick. All the elements of chop stick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

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

The structure of philosopher i.

Solution to dining philosophers problem:

- Allow at most four philosophers to be sitting simultaneously at the table
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available.
- Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick whereas an even philosopher picks up her right chopstick and then her left chopstick.

Critical Regions

- A **critical region** is a section of code that is always executed under mutual exclusion.

- Critical regions shift the responsibility for enforcing mutual exclusion from theprogrammer (where it resides when semaphores are used) to the compiler.
- They consist of two parts:
  1. Variables that must be accessed under mutual exclusion.
  2. A new language statement that identifies a critical regionin which thevariables are accessed.

Example:

```
var
v : shared T;
...
region v do
begin
...
end;
```

All critical regions that are ‘tagged’ with the same variable have compiler-enforced mutual exclusion so that only one of them can be executed at a time:

```
Process A:
region V1 do
begin
{ Do some stuff. } end;
region V2 do
begin
{ Do more stuff. }
end;

Process B:
region V1 do
begin
```

```
{ Do other stuff. }
end;
```

Here process A can be executing inside its V2 region while process Bis executing inside its V1 region, but if they both want to execute inside their respective V1 regions only one will be permitted to proceed.

Each shared variable (V1 and V2 above) has a queue associated with it. Once one process is executing code inside a region tagged with a shared variable, any other processes that attempt to enter a region tagged with the same variable are blocked and put in the queue.

Conditional Critical Regions

Critical regions aren't equivalent to semaphores. They lack condition synchronization. We can use semaphores to put a process to sleep until some condition is met (e.g. see the bounded-buffer Producer-Consumer problem), but we can't do this with critical regions.

Conditional critical regions provide condition synchronization for critical regions:

```
region v when B do
begin
...
end;
```

where B is a boolean expression (usually B will refer to v).

Conditional critical regions work as follows:

1. A process wanting to enter a region for v must obtain the mutex lock. If it cannot, then it is queued.
2. Once the lock is obtained the boolean expression B is tested. If B evaluates to true then the process proceeds, otherwise it releases the lock and is queued. When it next gets the lock it must retest B.

Implementation

Each shared variable now has two queues associated with it. The **main queue** is for processes that want to enter a critical region but find it locked. The **event queue** is for the processes that have blocked because they found the condition to be false. When a process leaves the conditional critical region the processes on the event queue join those in the main queue. Because these processes must retest their condition they are doing something akin to busy-waiting, although the frequency with which they will retest the condition is much less. Note also that the condition is only retested when there is reason to believe that it may have changed (another process has finished accessing the shared variable, potentially altering the condition). Though this is more controlled than busy-waiting, it may still be sufficiently close to it to be unattractive.

Limitations

- Conditional critical regions are still distributed among the program code.
- There is no control over the manipulation of the protected variables — no information hiding or encapsulation. Once a process is executing inside a critical region it can do whatever it likes to the variables it has exclusive access to.
- Conditional critical regions are more difficult to implement efficiently than semaphores.

Monitors

Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect since these errors happen only if some particular execution sequences take place and these sequences do not always occur. Suppose that a process interchanges the order in which the wait () and signal () operations on the semaphore mutex are executed.

```
Signal (mutex);
…….
Critical section
……..
Wait (mutex);
```

Here several processes may be executing in their critical sections simultaneously, violating the mutual exclusion requirement.

* Suppose that a process replaces signal (mutex) with wait (mutex) that is it executes

```
Wait(mutex);
……
Critical section
……. Wait(mutex);
```

Here a deadlock will occur.

* Suppose that a process omits the wait(mutex), or the signal(mutex) or both. Here, either mutual exclusion is violated or a dead lock will occur.

To deal with such errors, a fundamental high level synchronization construct called **monitor** type is used.

Usage

A monitor type presents a set of programmer defined operations that are provided mutual exclusion within the monitor. The monitor type also contains the declaration of variables whose values define the state of an instance of that type, along with the bodies of the procedures or functions that operate on those variables. The representation of a monitor type cannot be used directly by the various processes. Thus, a procedure defined within a monitor can access only those variables declared locally within the monitor and its formal parameters. The local variables of a monitor can be accessed by only the local procedures.

```
Name:monitor
…..local declarations
…..initialize local data
Proc1(….parameters)
……statement list
Proc2(…parameters)
…….statement list
```

```
                    Proc3(…parameters)

                    …….statement list
```

The monitor construct ensures that only one process at a time can be active within the monitor. But this monitor construct is not powerful for modeling some synchronization schemes. For this we need additional synchronization mechanisms. These mechanisms are provided by condition construct. The only operations that can be invoked on a condition variable are wait() and signal(). The operation x.wait() means that the process invoking this operation is suspended until another process invokes x.signal(). The x.signal() operation resumes exactly one suspended process.

When x.signal() operation is invoked by a process P, there is a suspended process Q associated with condition x. If suspended process Q is allowed to resume its execution, the signaling process P must wait. Otherwise, both P and Q would be active simultaneously within the monitor

However, both processes can conceptually continue with their execution. Two possibilities exist:

- Signal and wait – P either waits until Q leaves the monitor or waits for another condition.
- Signal and condition – Q either waits until P leaves the monitor or waits for another condition.

Dining Philosophers Solution using Monitors

This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available. To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we use this data structure enum {thinking, hungry, eating } state[5];

Philosopher i can set the variable state[i] = eating only if her two neighbors are not eating: (state [(i+4) % 5]! = eating) and (state [(i+1)%5]!=eating) Also declare condition self [5]; where philosopher i can delay herself when she is hungry but is unable to obtain the chop sticks she needs. The distribution of the chopsticks is controlled by the monitor dp. Each philosopher before starting to eat, must invoke the operation pickup().

Fig.3.4. Monitors

This may result in the suspension of the philosopher process. After the successful completion of the operation the philosopher may eat. Following this, the philosopher invokes the putdown() operation. Thus philosopher i must invoke the operations pickup() and putdown() in the following sequence:

```
dp.pickup(i);
……. Eat
…….
dp.putdown(i);
```

This solution ensures that no two neighbors are eating simultaneously and that no deadlocks will occur.

Implementing a Monitor using Semaphores

For each monitor, a semaphore mutex initialized to 1 is provided. A process must execute wait(mutex) before entering the monitor and must execute(signal) after leaving the monitor. Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore next initialized to 0, on which the signaling processes may suspend themselves. An integer variable next_count is used to count the number of processes suspended on next. Thus, each external procedure P is replaced by wait(mutex);

```
body of F
if (next_count > 0) signal(next);
else
signal(mutex);
```

Mutual exclusion within a monitor is thus ensured.

We can now describe how condition variables are implemented. For each condition x, we introduce a semaphore x_sem and an integer variable x_count, both initialized to 0. The operation x. wait () can now be implemented as x_count++;

```
if (next_count > 0)
signal(next);
else signal(mutex);
wait(x_sem);
x_count—;
```

The operation x. signal () can be implemented as

```
if (x_count > 0) {
next_count++;
signal(x_sem); wait(next) ;
next_count—;
```

}

Resuming Processes Within a Monitor

If several processes are suspended on condition x, and an x.signal() operation is executed by some process, then for determining which suspended process should be resumed next, we use FCFS ordering so that the process waiting the longest is resumed first. Or conditional wait construct() can be used as x.wait(c); where c is an integer expression that is evaluated when the wait () operation is executed. The value of c, which is called a **priority number,** is then stored with the name of the process that is suspended. When x. signal () is executed, the process with the smallest associated priority number is resumed next.

```
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;
```

A monitor to allocate a single resource

we consider the Resource Allocator monitor shown in which controls the allocation of a single resource among competing processes. Each process, when requesting an allocation of this resource, specifies the maximum time it plans to use the resource. The monitor allocates the resource to the process that has the shortest timeallocation request. A process that needs to access the resource in question must observe the following sequence:

R.acquire(t);

access the resource;

R. release () ;

where R is an instance of type ResourceAllocator. Unfortunately, the monitor concept cannot guarantee that the preceding access sequence will be observed. In particular, the following problems can occur:

- A process might access a resource without first gaining access permission to the resource.
- A process might never release a resource once it has been granted access to the resource.
- Synchronization Example
- A process might attempt to release a resource that it never request.
- A process might request the same resource twice (without first releasing the resource).

The same difficulties are encountered with the use of semaphores, and these difficulties are similar in nature to those that encouraged us to develop the monitor constructs in the first place. Previously, we had to worry about the correct use of semaphores.Now, we have to worry about the correct use of higher-level programmer-defined operations, with which the compiler can no longer assist us.

One possible solution to the current problem is to include the resource access operations within the ResourceAllocator monitor. However, using this solution will mean that scheduling is done according to the built-in monitor-scheduling algorithm rather than the one we have coded.

To ensure that the processes observe the appropriate sequences, we must inspect all the programs that make use of the ResourceAllocator monitor and its managed resource. We must check two conditions to establish the correctness of this system. First, user processes must always make their calls on the monitor in a correct sequence. Second, we must be sure that an uncooperative process does not simply ignore the mutual-exclusion gateway provided by the monitor and try to access the shared resource directly, without using the access protocols. Only if these two conditions can be ensured can we guarantee that no time- dependent errors will occur and that the scheduling algorithm will not be defeated. Although this inspection may be possible for a small, static system, it is not reasonable for a large system or a dynamic system. This access-control problem can be solved only by additional mechanisms.

Deadlock Problem

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

System has 2 tape drives.

P_0 and P_1 each hold one tape drive and each needs another one, semaphores A and B, initialized to 1

P_0	$P1$
wait (A);	*wait(B);*
wait (B);	*wait(A);*

Bridge Crossing Example

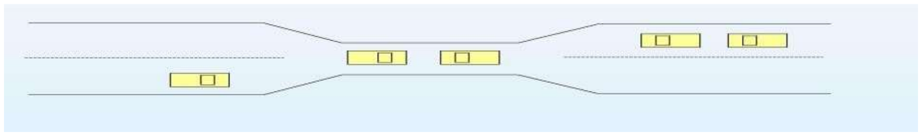


Fig.3.5. Bride Crossing Problem

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

1. Resource types R_1, R_2, . . ., R_m
2. *CPU cycles, memory space, I/O devices*

3. 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 $\{P_0, P_1, \ldots, P_0\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_0 is waiting for a resource that is held by P_0.

Resource-Allocation Graph

A set of vertices V and a set of edges E. V
is partitioned into two types:
$P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.

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

Request edge – directed edge $P_1 \rightarrow R_j$

Assignment edge – directed edge $R_j \rightarrow P_i$

- Process ○

- Resource Type with 4 instances
- Pi Requests Instance Of Rj
- Pi is Holding An Instance Of Rj

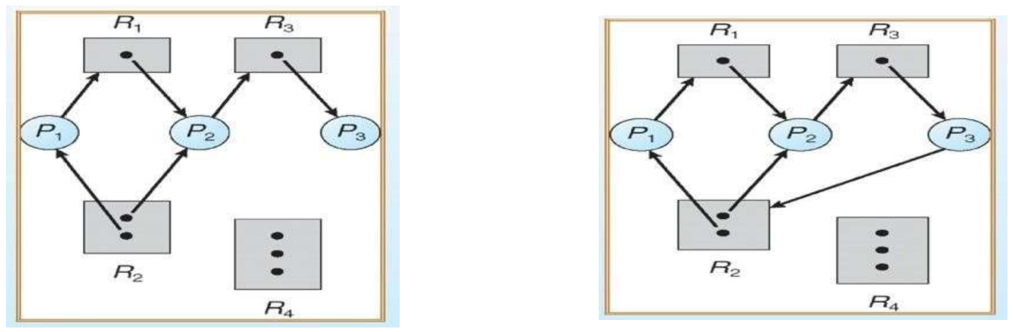


Fig.3.6. Resource Allocation Graph

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

- Deadlock Prevention or Avoidance- Ensure that the system will *never* enter a deadlock state
- Deadlock Detection- 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

- **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.
- **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.

Deadlock Detection

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

Safe State

- When aprocessrequests an availableresource, system must decideif immediate allocation leavesthe systemina 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.

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.

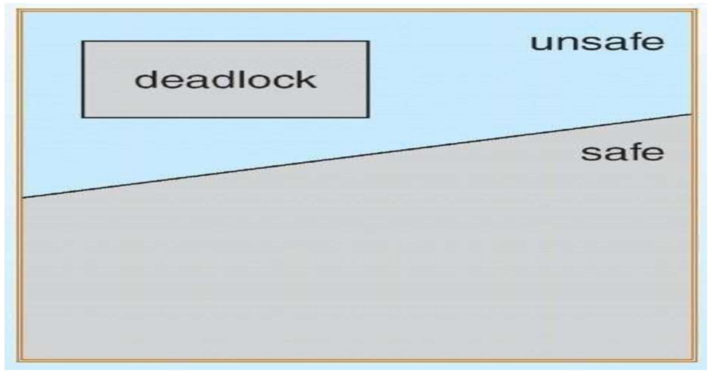


Fig.3.7. Safe and Unsafe State

Resource-Allocation Graph Algorithm

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

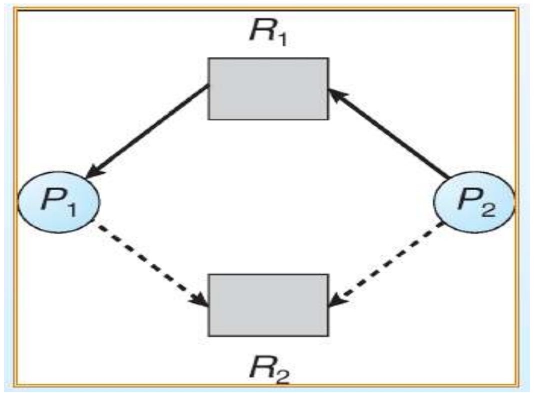


Fig.3.8. Resource Allocation Graph for Deadlock Avoidance

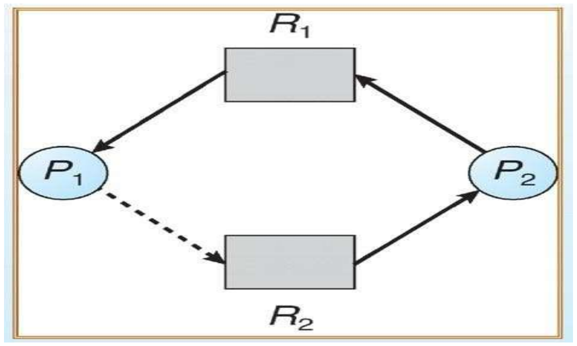


Fig.3.9. Unsafe State In Resource-Allocation Graph

Example formal algorithms

- Banker's Algorithm
- Resource-Request Algorithm
- Safety Algorithm

Banker's Algorithm

- Multiple instances.
- Each process must a priori claim maximum use.
- When a process requests a resource it may have to wait.
- When a process gets all its resources it must return them in a finite amount of time.

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types.

Available: Vector of length m. If available [j] = k, there are k instances of resource type R_j available. *Max: n x m* matrix. If *Max* [i,j] = k, then process P_i may request at most k instances of resource type R_j. *Allocation: n* x *m* matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of $R_{j.}$

Need: n x *m* matrix. If *Need*[i,j] = k, then P_i may need k more instances of R_j to complete its task.

Need [i,j] = *Max*[i,j] – *Allocation* [i,j]

Safety Algorithm

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

Work = *Available*

Finish [*i*] = *false* for *i* - 1,3, …, *n.*

2, Find and *i* such that both:

Finish [*i*] = *false* $Need_i \leq$

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 $R_{j.}$
- If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim.
- If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available.

Pretend to allocate requested resources to P_i by modifying the state as follows:

$Available = Available = Request_i;$

$Allocation_i = Allocation_i + Request_i$; $Need_i =$

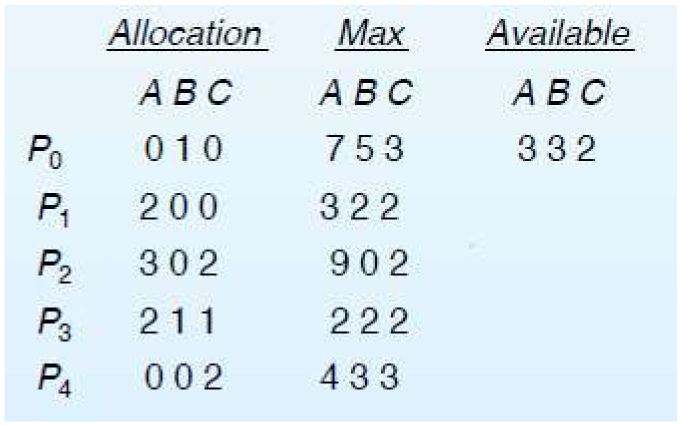
$Need_i – Request_i$

If safe ⇒ the resources are allocated to Pi.

If UNsafe ⇒ Pi must wait, and the old resource-allocation state is restored

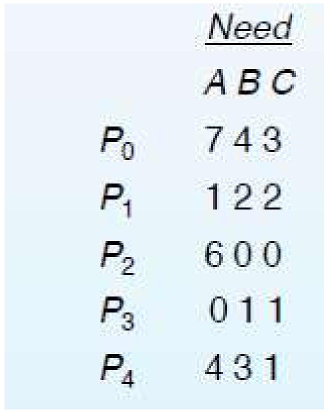
Example of Banker's Algorithm

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



	Allocation	*Max*	*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	

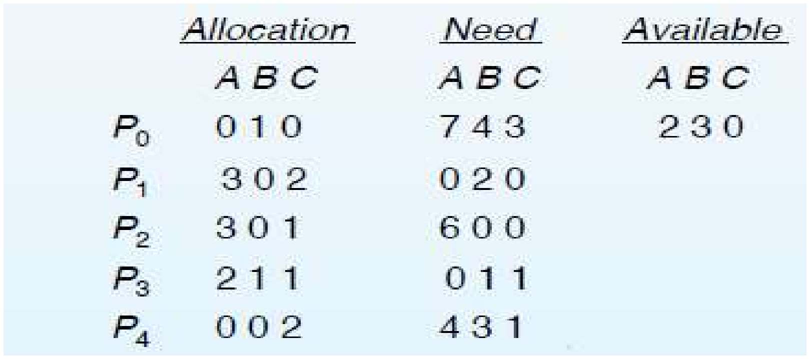
- The content of the matrix. Need is defined to be Max – Allocation.



	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

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

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



	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true.
- Executing safety algorithm shows that sequence <P1, P3, P4, P0, P2> satisfies safety requirement.
- Can request for (3,3,0) by P4 be granted?
- Can request for (0,2,0) by P0 be granted?

Deadlock Detection

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

Single Instance of Each Resource Type

- Maintain *wait-for* graph
- Nodes are processes.
- $P_i \rightarrow P_j$ if P_i is waiting for P_j.
- Periodically invoke an algorithm that searches for a cycle in the graph.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in thegraph.

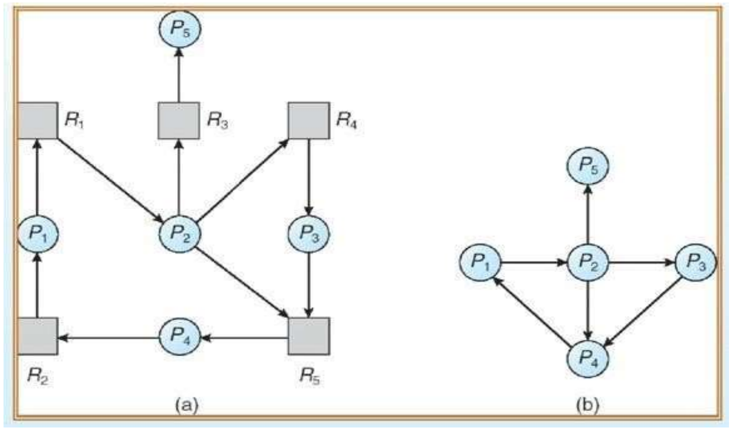


Fig.3.10 Resource-Allocation Graph and Corresponding wait-for graph

Several Instances of a Resource Type

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

Detection Algorithm

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

a, Work = *Available*

b, For $i = 1,2, \ldots, n$, if $Allocation_i \neq 0$, then *Finish*[i] = false; otherwise, *Finish*[i] = *true*.

2, Find an index *i* such that both:

a, Finish[*i*] == *false*

b, $Request_i \leq Work$

If no such *i* exists, go to step 4 3, *Work* =

$Work + Allocation_i$

Finish[*i*] = *true*

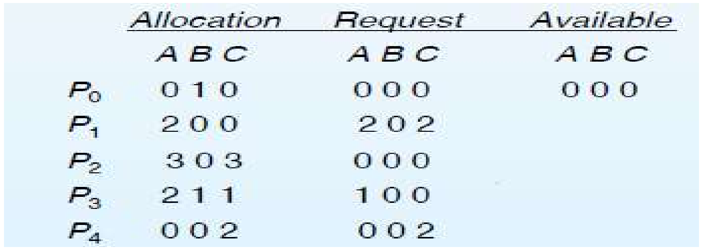
go to step 2.

4, If *Finish*[*i*] == false, for some i, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if *Finish*[*i*] == *false*, then P_i is deadlocked.

Algorithm requires an order of O(m x $n^{2)}$ operations to detect whether the system is in deadlocked state

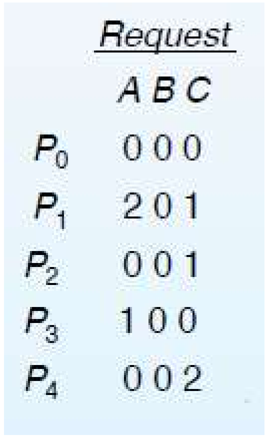
Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), B (2 instances), and C (6 instances).
- SnapShot at Time T0



	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

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



	Request
	A B C
P_0	0 0 0
P_1	2 0 1
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- P_2 requests an additional instance of type *C*.

Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests. Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.

Detection-Algorithm Usage

When, and how often, to invoke depends on:

1. How often a deadlock is likely to occur?
2. 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?
- 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

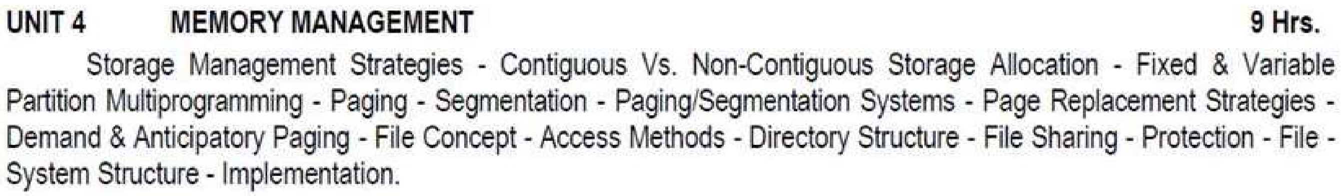
- Selecting a victim – minimize cost.
- Rollback – return to some safe state, restart process for that state.
- Starvation – same process may always be picked as victim, include number of rollback in cost factor.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

Common to : CSE , IT

UNIT IV



UNIT 4 MEMORY MANAGEMENT 9 Hrs.

Storage Management Strategies - Contiguous Vs. Non-Contiguous Storage Allocation - Fixed & Variable Partition Multiprogramming - Paging - Segmentation - Paging/Segmentation Systems - Page Replacement Strategies - Demand & Anticipatory Paging - File Concept - Access Methods - Directory Structure - File Sharing - Protection - File - System Structure - Implementation.