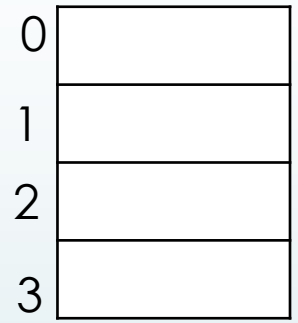
OPERATING SYSTEMS

Module3_Part1

Textbook : Operating Systems Concepts by Silberschatz

Producer consumer problem using shared memory

```
//shared data
Int item,in,out;
in=0;out=0;
Int BUFFER_SIZE=4
```



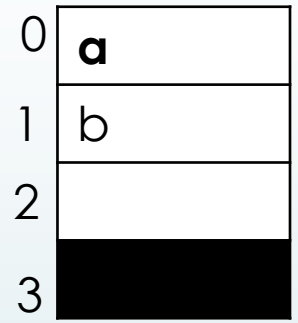

```
//producer process
while (true) {
    /* produce an item in next produced */
    while (((in + 1) % BUFFER_SIZE ) == out)
        ; /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

```
//consumer process

while (true) {
    while (in == out)

        ; /* do nothing */
    next_consumed = buffer[out];

    out = (out + 1) % BUFFER_SIZE;

}
```

Producer consumer problem using shared memory

```
//shared data
Int n=4,item,in,out;
in=0;out=0;
Int BUFFER_SIZE=4
```

in=2



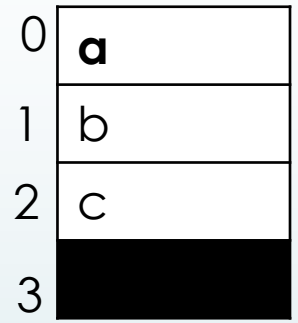

```
//producer process
while (true) {
   /* produce an item in next produced */
   while (((in + 1) % BUFFER_SIZE ) == out)
      ; /* do nothing */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
}
```

```
//consumer process

while (true) {
    while (in == out)

        ; /* do nothing */
    next_consumed = buffer[out];

    out = (out + 1) % BUFFER_SIZE;

}
```

Producer consumer problem using shared memory

```
//shared data
Int n=4,item,in,out;
in=0;out=0;
Int BUFFER_SIZE=4
```

in=3 producer waits



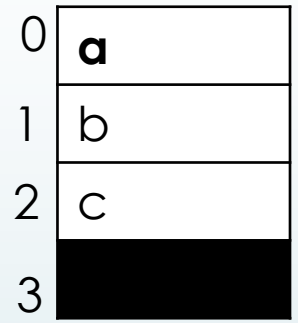

```
//producer process
while (true) {
   /* produce an item in next produced */
   while (((in + 1) % BUFFER_SIZE ) == out)
      ; /* do nothing */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
}
```

```
//consumer process
while (true) {
    while (in == out)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
}
```

Producer consumer problem using shared memory

```
//shared data
Int n=4,item,in,out;
in=0;out=0;
Int BUFFER_SIZE=4
```

in=3 producer waits



Now consumer runs

```
//producer process
while (true) {
   /* produce an item in next produced */
   while (((in + 1) % BUFFER_SIZE ) == out)
      ; /* do nothing */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
}
```

```
//consumer process
while (true) {
   while (in == out)
      ; /* do nothing */
   next_consumed = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
}
```

The solution allowed at most BUFFER_SIZE - 1 items in the buffer at the same time.

we can modify the algorithm to remedy this deficiency.

One possibility is to add an integer variable counter, initialized to 0.

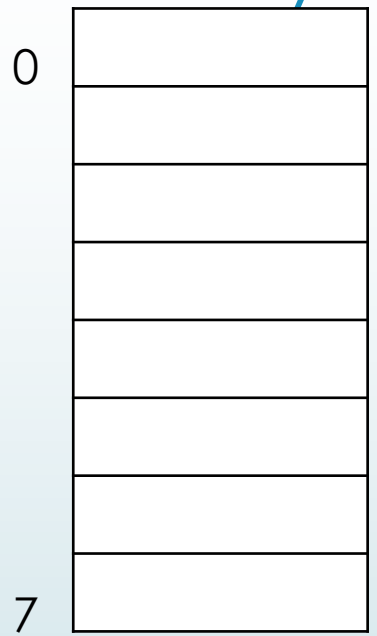
counter is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer.

Producer consumer problem using shared memory

```
//shared data
int count=0, n=8;
```

```
//producer process
while (true) {

    while (counter == BUFFER_SIZE)
        ; /* do nothing */

    buffer[in] = next_produced;

    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

```
//consumer process
while (true) {
    while (counter == 0)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
}
```



Producer consumer problem using shared memory

```
//shared data
int count=0;
```

Current status count=5

0	**x1**
1	X2
2	X3
3	X4
4	x5
5	
6	
7	

```
//producer process
while (true) {

    while (counter == BUFFER_SIZE)
        ; /* do nothing */

    buffer[in] = next_produced;

    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

```
register1 = counter
register1 = register1 + 1
counter = register1
```

```
//consumer process
while (true) {
    while (counter == 0)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
}
```

```
register2 = counter
register2 = register2 - 1
counter = register2
```

Producer consumer problem using shared memory

```
//shared data
int count=0;
```

Producer runs: S0: producer execute
`register1 = counter` {register1 = 5}
S1: producer execute
`register1 = register1 + 1` {register1 = 6}
Process switch occurs

0	**x1**
1	X2
2	X3
3	X4
4	X5
5	x6
6	
7	

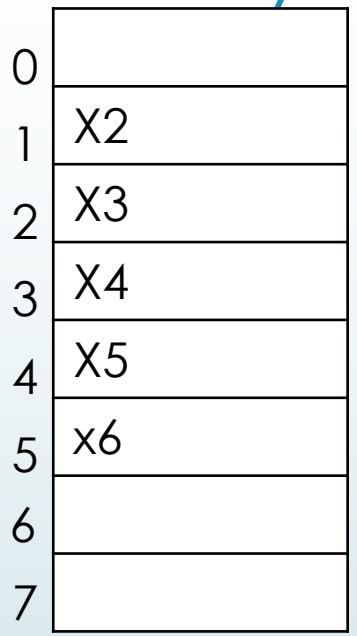
```
//producer process
while (true) {

    while (counter == BUFFER_SIZE)
        ; /* do nothing */

    buffer[in] = next_produced;

    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

```
//consumer process
while (true) {
    while (counter == 0)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
}
```

Producer consumer problem using shared memory

```
//shared data
int count=0;
```

S2: consumer execute
`register2 = counter` {register2 = 5}
S3: consumer execute
`register2 = register2 - 1` {register2 = 4}
Process switch occurs




```
//producer process
while (true) {

    while (counter == BUFFER_SIZE)
        ; /* do nothing */

    buffer[in] = next_produced;

    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

```
//consumer process
while (true) {
        while (counter == 0)
            ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
}
```

Producer consumer problem using shared memory

```
//shared data
int count=0;
```

S4: producer execute
`counter = register1` {counter = 6 }
Process switch occurs

0	
1	X2
2	X3
3	X4
4	X5
5	x6
6	
7	

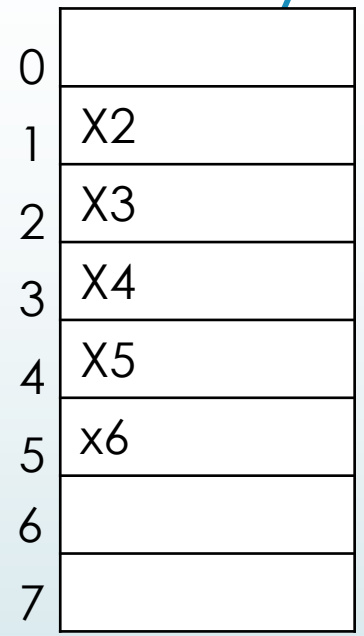
```
//producer process
while (true) {

    while (counter == BUFFER_SIZE)
        ; /* do nothing */

    buffer[in] = next_produced;

    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

```
//consumer process
while (true) {
    while (counter == 0)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
}
```

Producer consumer problem using shared memory

```
//shared data
int count=0;
```

S5: consumer execute **`counter = register2`**
{counter = 4}




```
//producer process
while (true) {

    while (counter == BUFFER_SIZE)
        ; /* do nothing */

    buffer[in] = next_produced;

    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

```
//consumer process
while (true) {
    while (counter == 0)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
}
```

- **`counter++`** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **`counter--`** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

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

S0: producer execute **`register1 = counter`** {register1 = 5}
S1: producer execute **`register1 = register1 + 1`** {register1 = 6}
S2: consumer execute **`register2 = counter`** {register2 = 5}
S3: consumer execute **`register2 = register2 - 1`** {register2 = 4}
S4: producer execute **`counter = register1`** {counter = 6}
S5: consumer execute **`counter = register2`** {counter = 4}

- Notice that we have arrived at the incorrect state "counter == 4", indicating that four buffers are full, when, in fact, five buffers are full.
- We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently.
- A situation like this, where 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 a race condition.

To guard against the race condition above,

we need to ensure that only one process at a time can be manipulating the variable counter.

To make such a guarantee, we require that the processes be synchronized in some way.

OPERATING SYSTEMS

Module3_Part2

Textbook : Operating Systems Concepts by Silberschatz

Process synchronisation

- 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 a **race condition.**

To avoid race condition, processes are to be synchronized in some way.

To do process synchronization, we should first to know what a **critical section** is.

Critical section

Consider a system consisting of *n* processes {Po, P1 , ... , *Pn* _ I}.

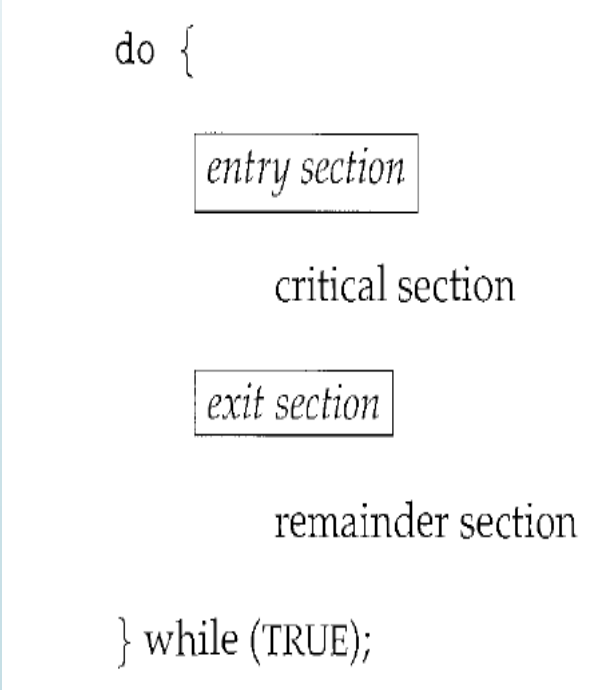
Each process has a segment of code, called a critical section in which the process may be changing common variables, updating a table, writing a file, and so on.(shared)

Part of the program where the shared memory is accessed is called **critical section** or critical region of **that process**

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. If so we can avoid race conditions.

Critical section

- The *critical-section problem* is to design a protocol that the processes can use to cooperate. 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*.
- The general structure of a typical process *Pi* is shown in



Critical section

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

1. **Mutual exclusion.** If process *Pi* 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 some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, this selection cannot be postponed indefinitely.
3. **Bounded waiting.** There exists a bound, or limit, 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

One of the process synchronization mechanism

- Peterson's solution is a classic software-based solution to the critical-section problem
- it provides a good algorithmic description of solving the critical-section problem
- Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.

The processes are numbered *Po* and P1.

Peterson's solution

- The two processes share two variables:
  - `int turn;`
  - `boolean flag[2]`
- The variable **`turn`** indicates whose turn it is to enter the critical section
- The **`flag`** array is used to indicate if a process is ready to enter the critical section.
  - `flag[i] = true` implies that process $\mathbf{P_i}$ is ready!

Algorithm for Process P_i

```
while (true){

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

        /* critical section */

    flag[i] = false;

    /* remainder section */

}
```

Structure for p0

Initially flag[0] and flag[1] assigns false

```
while (true){
    flag[0] = true;
    turn = 1;
    while (flag[1] && turn = = 1)
        ;

/* critical section */

    flag[0] = false;

    /* remainder section */
}
```

structure for p1

```
while (true){
    flag[1] = true;
    turn = 0;
    while (flag[0] && turn = = 0)
        ;

/* critical section */

    flag[1] = false;

    /* remainder section */
}
```

Peterson’s solution

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

1. **Mutual exclusion.**
2. **Progress.**
3. **Bounded waiting.**

Structure for p0

```
while (true){
    flag[0] = true;
    turn = 1;
    while (flag[1] && turn = = 1)
        ;

/* critical section */

    flag[0] = false;

    /* remainder section */
}
```

structure for p1

```
while (true){
    flag[1] = true;
    turn = 0;
    while (flag[0] && turn = = 0)
        ;

/* critical section */

    flag[1] = false;

    /* remainder section */
}
```

P0 and p1 willing
P1 in CS and p0 wants to enter
P0 in CS and p1 wants to enter
P0 wants to enter and P1 has no interest

Structure for p0

```
while (true){
    flag[0] = true;
    turn = 1;
    while (flag[1] && turn = = 1)
        ;

/* critical section */

    flag[0] = false;

    /* remainder section */
}
```

P0 wants to enter and P1 has no interest
If p0 wants to enter ,it is possible, we can say there is progress

structure for p1

```
while (true){
    flag[1] = true;
    turn = 0;
    while (flag[0] && turn = = 0)
        ;

/* critical section */

    flag[1] = false;

    /* remainder section */
}
```

Structure for p0

```
while (true){
    flag[0] = true;
    turn = 1;
    while (flag[1] && turn = = 1)
        ;

/* critical section */

    flag[0] = false;

    /* remainder section */
}
```

structure for p1

```
while (true){
    flag[1] = true;
    turn = 0;
    while (flag[0] && turn = = 0)
        ;

/* critical section */

    flag[1] = false;

    /* remainder section */
}
```

In Peterson algorithm a process will never wait longer than one turn for entrance to the critical section

OPERATING SYSTEMS

Module3_Part3

Textbook : Operating Systems Concepts by Silberschatz

Mutex Locks

- The hard ware based solution Test and Set Lock for critical section problem is complicated and inaccessible to programmers.
- Another software tool used to solve critical section problem is Mutex Locks.
- We use murex locks to protect critical regions and avoid race conditions
- Process must acquire lock before entering the critical region and release the lock when it exits.
- The acquire function() acquire the lock and release function() releases the lock

Mutex locks

- Mutex has a boolean variable available whose value indicates lock is available or not

    if lock is available call to acquire() succeeds; the lock is then considered unavailable

    the process attempt to acquire unavailable lock is blocked until the lock is available

```
acquire() {
    while(!available)
    ;/* busy wait*/
    available:=false;
    }
```

The definition of release() as follows

```
    release() {
        available=true;
        }
```

Mutex locks

- do {

Acquire lock

Critical section

Release lock

noncritical section

} while(true);

Call to either acquire() or release() must be performed atomically

Solution to critical section problem using mutex locks

Mutex locks

The main disadvantage of mutex locks is **busy waiting**

while a process is in its critical section, any other process that tries to enter its critical section loops continuously in the call to acquire()

This type mutex lock is also called spin lock ,since the process spins while waiting for the lock to become available

The spin locks do have advantage that **no context switch** is required when process wait on a lock(context switch may take considerable time)

When locks are expected to be held for short times spin locks are useful

Process synchronization using semaphore

- A **semaphore** is a synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities..
- A semaphore S is an integer variable that, apart from initialization, is accessed only

  through two standard atomic operations: wait () and signal ().

  wait () operation was originally termed P;

  signal() was originally called V.

Process synchronization using semaphore

The Definition of wait () is as follows:

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

The definition of signal() is as follows:

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

semaphore

- All modifications to the integer value of the semaphore in the wait () and signal() operations must be executed indivisibly.
- That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.
- In addition, in the case of wait (S), the testing of the integer value of S (S <= 0), as well as its possible modification (S--), must be executed without interruption

semaphore

Usage

Two types of semaphores

counting semaphore: The value can range over an unrestricted domain.

binary semaphore:The value can range only between 0 and 1.

binary semaphores are known as mutex locks, as they are locks that provide

mutual exclusion.

Binary semaphore

- We can use binary semaphores to deal with the critical-section problem £or multiple processes. Then processes share a semaphore, mutex, initialized to 1. Each process *Pi* is organized as shown below

```
do {
   wait(mutex);

      // critical section

   signal(mutex);

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

Mutual-exclusion implementation with semaphores.

Counting semaphore

- Counting semaphores can be used to control access to a given resource consisting of a finite number o£ 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 (thereby decrementing the count).
- When a process releases a resource, it performs a signal() operation (incrementing the count).
- 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

- We can also use semaphores to solve various synchronization problems.
- For example, consider two concurrently running processes: P1 with a statement S1 and P2 with a statement S2 .

Suppose we require that S2 be executed only after S1 has completed.

We can implement this scheme readily by letting P1 and P2 share a common semaphore synch, initialized to 0, and by

inserting the statements

S1;

signal(synch) ;

in process P1

and the statements

wait(synch);

S2; in process P2.

Because synch is initialized to 0, P2 will execute S2 only after P1

has invoked signal (synch), which is after statement S1 has been executed.

OPERATING SYSTEMS

Module3_Part4

Textbook : Operating Systems Concepts by Silberschatz

Semaphore Implementation

- The main **disadvantage** of the semaphore definition given here 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.

  This continual looping is clearly a problem in a real multiprogramming system where a single CPU is shared among many processes.

  Busy waiting wastes CPU cycles that some other process might be able to use productively.

Semaphore Implementation

- To overcome the need for busy waiting, we can modify the definition of

  the wait() and signal() semaphore operations.
- When a process executes the wait () operation and finds that the semaphore value is not positive, it must wait.
- However, 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 the CPU scheduler, which selects another process to execute.

Semaphore Implementation

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. The 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;
```

Semaphore Implementation

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.

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

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

Semaphore Implementation

- The signal () semaphore operation can now 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.

Semaphore Implementation

- In this implementation, semaphore values may be negative, although semaphore values are never negative under the classical definition of semaphores with busy waiting.
- If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore.
- Each semaphore contains an integer value and a pointer to a list of PCBs. One way to add and remove processes from the list so as to ensure bounded waiting is to use a FIFO queue, where the semaphore contains both head and tail pointers to the queue.

Semaphore Implementation

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

Deadlock and starvation

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

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
.	.
.	.
.	.
signal(S);	signal(Q);
signal(Q);	signal(S);

Suppose that *Po* 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.

Deadlock and starvation

- 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 and release.*

- Another problem is starvation--a situation in which processes wait indefinitely within the semaphore. Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

Priority inversion problem

Suppose there are three processes L(low priority process),M(Medium priority process) and H(High priority process)

- L is running in CS ; H also needs to run in CS ; H waits for L to come out of CS ; M interrupts L and starts running ; M runs till completion and relinquishes control ; L resumes and starts running till the end of CS ; H enters CS and starts running. Note that neither L nor H share CS with M.
- Here, we can see that running of M has delayed the running of both L and H. Precisely speaking, H is of higher priority and doesn't share CS with M; but H had to wait for M. This is where Priority based scheduling didn't work as expected because priorities of M and H got inverted in spite of not sharing any CS. This problem is called Priority Inversion.

Priority inheritance protocol

These systems solve the problem by implementing a *priority inheritance protocol*

According to this protocol, all processes that are accessing resources, needed by a higher-priority process, inherit the higher priority until they are finished with the resources in question. When they are finished, their priorities revert to their original values.

In the example above, a priority-inheritance protocol would allow process *L* to temporarily inherit the priority of process H, thereby preventing process *M* from preempting its execution. When process *L* had finished using resource *R*, it would relinquish its inherited priority from H and assume its original priority. Because resource *R* would now be available, process H-not M-would run next.

OPERATING SYSTEMS

Module3_Part5

Textbook : Operating Systems Concepts by Silberschatz

Producer consumer problem using semaphores

- we present a general structure of this scheme without committing ourselves to any particular implementation;
- We assume that 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 the value 0.

Producer consumer problem using semaphores

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

Producer consumer problem using semaphores

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

We can interpret this code as the producer producing full buffers for the consumer or as the consumer producing empty buffers for the producer.

The Readers-Writers Problem

- Suppose that 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.
- the former is called readers and to the latter called writers.
- if two readers access the shared data simultaneously, no adverse effects will result.
- However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.
- To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database.
- no reader be kept waiting unless a writer has already obtained permission to use the shared object.

Solution to readers writers problem

- In the solution to the first readers-writers problem, the reader processes share the following data structures:

  semaphore mutex, wrt;

  int readcount;

- The semaphores mutex and wrt are initialized to 1; readcount is initializedto 0.
- The 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 sections.

The Readers-Writers Problem solution

```
typedef int semaphore;
semaphore mutex=1;
semaphore wrt=1;
int readcount=0;
```

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

Structure of a writer

Solution--The Readers-Writers Problem

```
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);
```

Structure of a reader

```
//gain access to readcount
//increment readcount
//if this is the first process to read db
//prevent writer process to access db
//allow other process to access readcount
```

```
//gain access to readcount
//decrement readcounter
//this is the last process to read db
//leave control of db,allow writer process
//allow other process to access readcount
```

OPERATING SYSTEMS

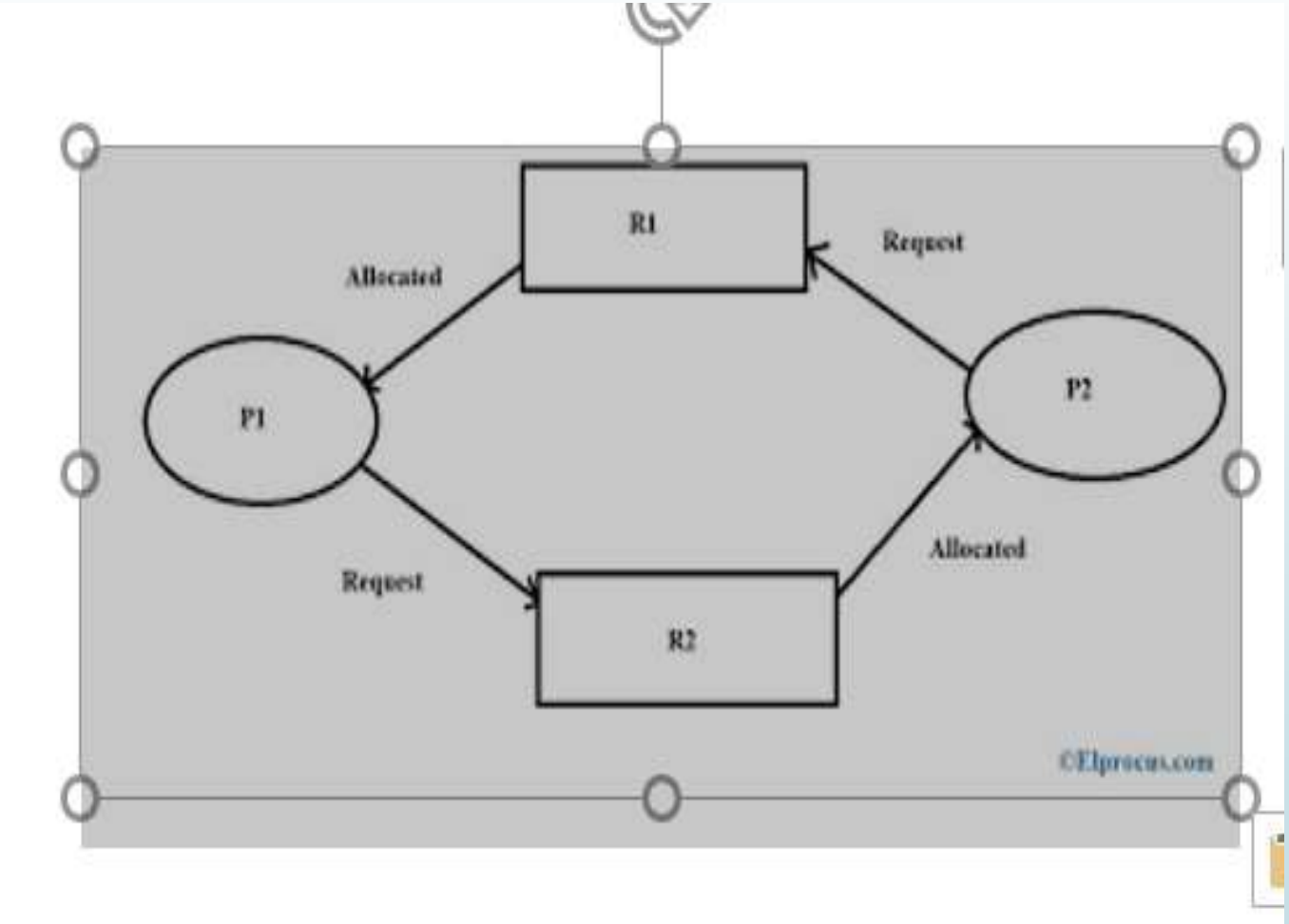
Module3_Part6

Textbook : Operating Systems Concepts by Silberschatz

Deadlocks

- For many applications a process needs exclusive access to not one resource, but several.
- For eg: There are two process each want to record a scanned document on a CD.
- Process A request permission to use the scanner and granted it. Process B programmed differently and requests CD recorder first and it also granted
- Now A asks for CD recorder, but request is denied until B releases it
- Instead of releasing the CD recorder B asks for scanner
- At this point both processes are blocked and remaining so forever.
- This situation is called **Deadlock**

Deadlock



Deadlock

Deadlock definition

- A set of processes is in a deadlocked 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 and release.
- The resources may be either physical resources (for example, printers, tape drives, memory space, and CPU cycles) or logical resources (for example, files, semaphores, and monitors).

Resources

- A system consists of a finite number of resources to be distributed among a number of competing processes.
- The resources are partitioned into several types, each consisting of some number of identical instances.
- Memory space, CPU cycles, files, and I/0 devices (such as printers and DVD drives) are examples of resource types.
- A process must request a resource before using it and must release the resource after using it. A process may request as many resources as it requires to carry out its designated task.
- The number of resources requested may not exceed the total number of resources available in the system.
- In other words, a process cannot request three printers if the system has only two.

Resources

Process utilization of resource in following sequence only

- Request. The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
- Use. The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
- Release. The process releases the resource.
- The request and release of resources are system calls,

Resources

- A system table records whether each resource is free or allocated;
- for each resource that is allocated, the table also records the process to which it is allocated.
- If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

Necessary conditions for a deadlock

features that characterize deadlocks.

- A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1.Mutual exclusion. Each resource is exactly assigned to one process or is available. that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. Hold and wait. A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

ie a process holding a resource can request for some other resource without releasing it.

Necessary conditions for a deadlock

3. **No preemption.** Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

4. **Circular wait**. A set { *P0* , Pl, ... , Pn } of waiting processes must exist such that *Po* is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ... , Pn-1 is waiting for a resource held by *Pn* and Pn is waiting for a resource held by *Po*.
   There should be a circular chain of processes and resources.

all four conditions must hold for a deadlock to occur.

OPERATING SYSTEMS

Module3_Part7

Textbook : Operating Systems Concepts by Silberschatz

Resource-Allocation Graph

- Deadlocks can be described more precisely in terms of a directed graph called a system resource allocation graph.
- This graph consists of a set of vertices *V* and a set of edges *E.* The set of vertices V is partitioned into two different types of nodes:

  P == { P1, P2, ... , Pn}, the set consisting of

  all the active processes in the system,

  R == {R1, R2, ... , *Rm}* the set consisting of all resource types in the system.

Resource-Allocation Graph

- A directed edge from process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$;
  it signifies that process P_i has requested an instance of resource type *Rj* and
  is currently waiting for that resource.

A directed edge from resource type *Rj* to process P_i is denoted by $Rj \rightarrow P_i$; it signifies that an instance of resource type *Rj* has been allocated to process P_i ;

A directed edge $P_i \rightarrow R_j$ is called a request edge;
a directed edge $Rj \rightarrow P_i$; is called an assignment edge

Resource-Allocation Graph

- Pictorially we represent each process P_i as a circle and each resource type *Rj* as a rectangle.
- Since resource type *Rj* may have more than one instance, we represent each such instance as a dot within the rectangle.
- Note that a request edge points to only the rectangle *Rj,* whereas an assignment edge must also designate one of the dots in the rectangle.
- When process P_i requests an instance of resource type *Rj,* a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is *instantaneously* transformed to an assignment edge. When the process no longer needs access to the resource, it releases the resource; as a result, the assignment edge is deleted.

Resource-Allocation Graph

Consider the current situation in the system

The sets *P,* R and *E:*

P == {P1, P2, P3}

R== {R1, R2, R3, R4}

E == {Pl→ Rl, p2→ *R3,* Rl→ *p2,* R2→ *p2,*R2→ Pl, R3→ P3}

Resource instances:

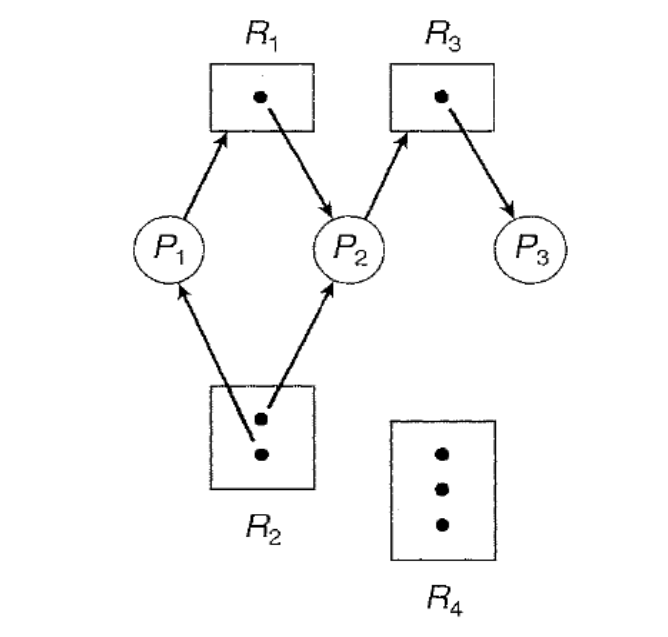
One instance of resource type R1

Two instances of resource type R2

One instance of resource type R3

Three instances of resource type R4

Resource allocation graph



Process states:
Process P1 is holding an instance of resource type R2 and is waiting for an instance of resource type R1 .

Process P2 is holding an instance of R1 and an instance of R2 and is waiting for an instance of R3.
Process *P3* is holding an instance of R3 .

Resource allocation graph

Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked.

If the graph does contain a cycle, then a deadlock may exist.

If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred.

If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked. In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of deadlock.

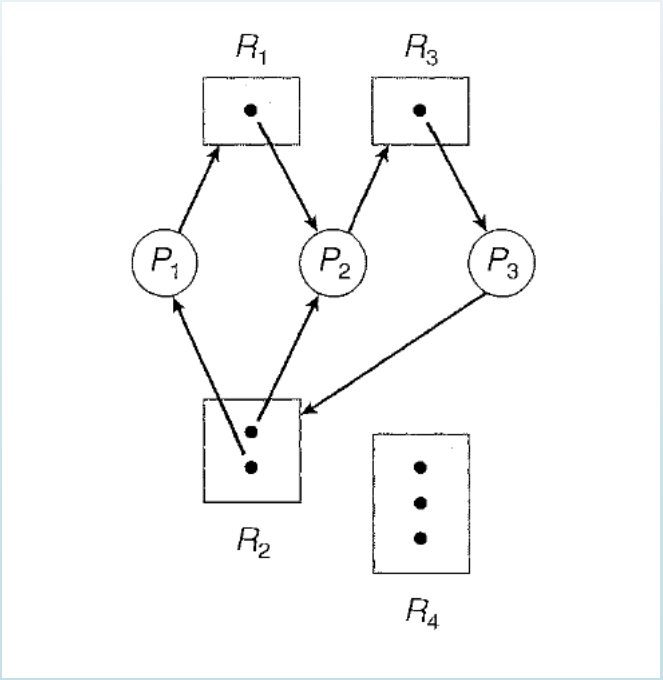
Resource allocation graph

- If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of deadlock.

To illustrate this concept, we return to the resource-allocation graph depicted in Figure. Suppose that process *P3* requests an instance of resource type R2. Since no resource instance is currently available, a request edge *P3* → R2 is added to the previous graph. At this point, two minimal cycles exist in the system:

Resource allocation graph

- P1 → *R* 1 → P2 → R3 → *P3* → R2 → P1
- P2 → R3 → *P3* → R2 → P2



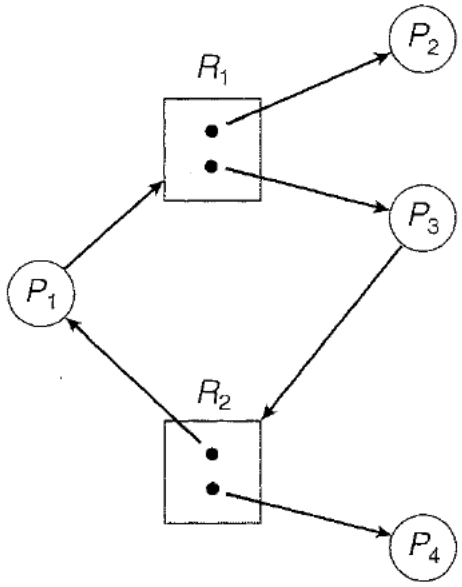
Resource-allocation graph with a deadlock

Resource allocation graph

- Processes *P1*, *P2*, and *P3* are deadlocked. Process *P2* is waiting for the resource
- *R3*, which is held by process *P3*. Process *P3* is waiting for either process *P1* or
- process *P2* to release resource R2. In addition, process *P1* is waiting for process
- *P2* to release resource R1.

Resource allocation graph

- Now consider the resource-allocation graph below. in this example,
- we also have a cycle:
- P1→ R1→P3→ R2→ P1



Resource allocation graph with cycle and no deadlock

However, there is no deadlock. Observe that process *P4* may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle. In summary, if a resource-allocation graph does not have a cycle, then the system is *not* in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state. This observation is important when we deal with the deadlock problem.

OPERATING SYSTEMS

Module3_Part8

Textbook : Operating Systems Concepts by Silberschatz

Methods of handling the deadlock

we can deal with the deadlock problem in the following ways:

1. We can use a protocol to prevent a deadlock by structurally negating one of the four conditions necessary to occur a deadlock.
2. Deadlock avoidance by careful resource allocation, ensuring that the system will *never* enter a deadlocked state.
2. We can allow the system to enter a deadlocked state, detect it, and recover. Let deadlock occur, detect them, and take action.
3. We can ignore the problem altogether and pretend that deadlocks never occur in the system

Deadlock prevention

For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can *prevent* the occurrence of a deadlock.

1. Attacking the mutual exclusion condition

If no resource were ever assigned exclusively to a single process, we would never have a deadlock.

Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock.

Deadlock prevention

- Hard to avoid mutual exclusion for non sharable resources

  Printer & other I/O devices

  Files

- However, many resources are sharable, so deadlock can be avoided for those resources

  Read-only files

- For printer, avoid mutual exclusion through spooling — then process won't have to wait on physical printer
- In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable.

Deadlock prevention

2. Attacking hold and wait condition

- To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.
- One protocol that can be used requires each process to request and be allocated all its resources before it begins execution.
- An alternative protocol allows a process request some resources and use them. Before it can request any additional resources, however, it must release all the resources that it is currently allocated

Deadlock prevention

Both these protocols have **two main disadvantages.**

First, **resource utilization may be low**, since resources may be allocated but unused for a long period.

Second, **starvation is possible**. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

Deadlock prevention

3. Attacking no preemption condition

- The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated. To ensure that *this condition does not hold*, we can use the following protocol.
- Allow preemption
- If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources the process is currently holding are preempted.(take away).

  The preempted resources are added to the list of resources for which the process is waiting.

  The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Deadlock prevention

- Alternatively, if a process requests some resources, we first check whether they are available. If they are, we allocate them.
- If they are not, we check whether they are allocated to some other process that is waiting for additional resources.
- If so, we preempt the desired resources from the waiting process and allocate them to the requesting process.

This protocol is often applied to resources whose state can be easily saved and restored later, such as CPU registers and memory space.

It cannot generally be applied to such resources as printers and tape drives.

Deadlock prevention

4.Attacking circular wait condition

The fourth and final condition for deadlocks is the circular-wait condition.

Ensure that this condition never holds

> impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.

To illustrate, we let *R* = { R1, R2, ... , Rm} be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering. Formally, we define a one-to-one Function F: *R* → *N,* where *N* is the set of natural numbers

- For example, if the set of resource types *R* includes tape drives, disk drives, and printers, then the function *F* might be defined as follows:

  F (tape drive) = 1

  F (disk drive) = 5

  F (printer) = 12

Deadlock prevention

- We can now consider the following protocol to prevent deadlocks: Each process can request resources only in an increasing order of enumeration.
- That is, a process can initially request any number of instances of a resource type
- -say, R_i. After that, the process can request instances of resource type *Rj* if

  and only if F(Rj) > F(Ri).
- For example, using the function defined previously, a process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer.
- Alternatively, we can require that a process requesting an instance of resource type *Rj* must have released any resources R_i such that F(Ri) >= F(Rj).

OPERATING SYSTEMS

Module3_Part9

Textbook : Operating Systems Concepts by Silberschatz

Deadlock recovery

- When a detection algorithm determines that a deadlock exists, some way is needed to recover from a deadlock

- The ways to recover from a deadlock

1. Simply to abort one or more processes to break the circular wait(Process termination).

2. To preempt some resources from one or more of the deadlocked processes (Resource Preemption).

Process termination

There are two methods for process termination

Abort all deadlocked processes.

This method clearly will break the deadlock cycle, but at great expense.The deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.

Abort one process at a time until the deadlock cycle is eliminated.

This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Process termination

Aborting a process may not be easy.

If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state.

if the process was in the midst of printing data on a printer, the system must reset the printer to a correct state before printing the next job.

If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated.

Process termination

- we should abort those processes whose termination will incur the minimum cost. Unfortunately, the term *minimum cost* is not a precise one.

- Many factors may affect which process is chosen, including:

  1.What the priority of the process is

  2. How long the process has computed and how much longer the process will compute before completing its designated task

  3.How many and what types of resources the process has used (for example, whether the resources are simple to preempt)

  4.How many more resources the process needs in order to complete

  5. How many processes will need to be terminated

  6.Whether the process is interactive or batch

Resource preemption

To eliminate deadlocks using resource preemption,

we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

Here three issues need to be addressed:

- **Selecting a victim.** Which resources and which processes are to be preempted?

  we must determine the order of preemption to minimize cost.

  Cost factors may include such parameters as

  the number of resources a deadlocked process is holding

  the amount of time the process has thus far consumed during its execution.

- **Rollback.** If we preempt a resource from a process, what should be done with that process?

  Clearly, it cannot continue with its normal execution; it is missing some needed resource.

  We must roll back the process to some safe state and restart it from that state.

Resource preemption

Since, in general, it is difficult to determine what a safe state is, the simplest solution is

- a total rollback: abort the process and then restart it.

Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system *to keep more information about the state of all running processes*

- **Starvation.** How do we ensure that starvation will not occur? That is,how can we guarantee that resources will not always be preempted from the same process?

we must ensure that a process can be picked as a victim" only a (small) finite number of times. The most common solution is to include *the number of rollbacks* in the cost factor.

OPERATING SYSTEMS

Module3_Part10

Textbook : Operating Systems Concepts by Silberschatz

Deadlock avoidance

Avoiding deadlocks requires information about how resources are to be requested.

With the knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock.

Each request requires that in making this decision, the system consider

- the resources currently available,
- the resources currently allocated to each process, and
- the future requests and releases of each process.

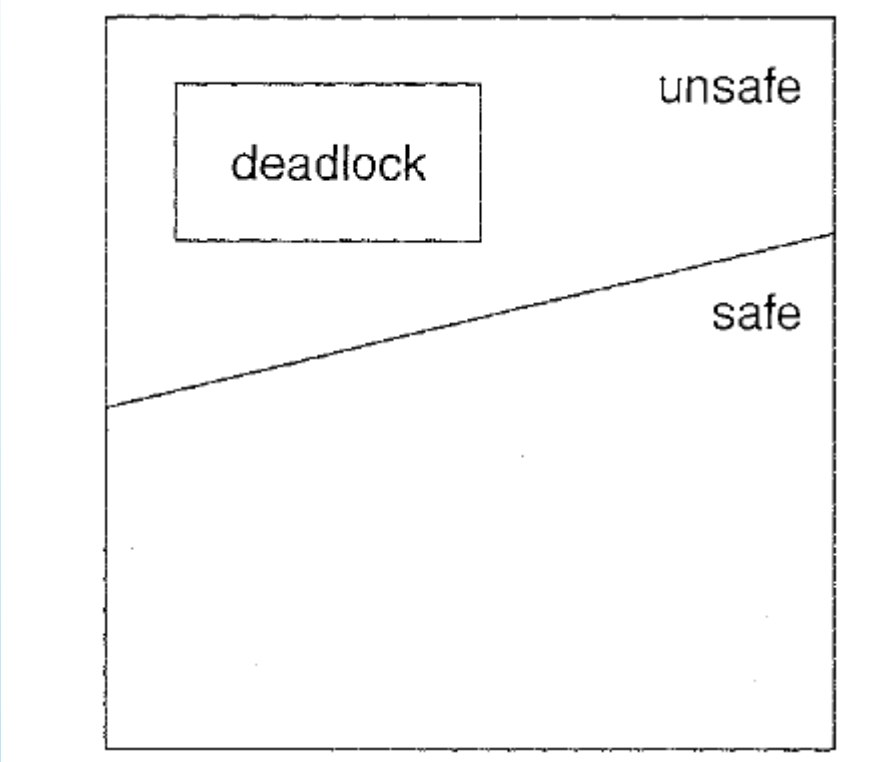
Deadlock avoidance

- The simplest and most useful model requires that **each process declare the *maximum number* of resources of each type that it may need**.
- Using this information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state.
- This deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular wait condition can never exist.
- The resource-allocation *state* is defined by the number of available and allocated resources and the maximum demands of the processes.

Safe State

- A state is *safe* if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence.
- A sequence of processes <P1, P2, ... , *Pn*> is a safe sequence for the current allocation state if, for each *Pi*, the resource requests that *Pi* can still make can be satisfied by the currently available resources plus the resources held by all *Pj*, with $j < i$.
- if the resources that *Pi* needs are not immediately available, then *Pi* can wait until all *Pj* have finished. When they have finished, *Pi* can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When *Pi* terminates, *Pi+l* can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be *unsafe*.

- A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state.
- Not all unsafe states are deadlocks,
- However.an unsafe state *may* lead to a deadlock.
- As long as the state is safe, the operating system can avoid unsafe (and deadlocked) states.
- In an unsafe state, the operating system cannot prevent processes from requesting resources in such a way that a deadlock occurs. The behavior of the processes controls unsafe states.



Example

Consider a system with 12 magnetic tape drives and three processes: *Po*, P1, and P2.

- Process *Po* requires 10 tape drives,
- process P1 may need as many as 4 tape drives,
- process P2 may need up to 9 tape drives.

Suppose that, at time *to*,

process *Po* is holding 5 tape drives,

process P1 is holding 2 tape drives,

process P2 is holding 2 tape drives.

(Thus, there are three free tape drives.)

Example(safe state)

- At time *t0*, the system is in a safe state. The sequence <P1, *P0*, P2> satisfies the safety condition.
- Process P1 can immediately be allocated all its tape drives and then return them (the system will then have five available tape drives);
- then process *Po* can get all its tape drives and return them (the system will then have ten available tape drives);
- and finally process *P2* can get all its tape drives and return them (the system will then have all twelve tape drives available).

	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

Available=3

Example(unsafe state)

- A system can go from a safe state to an unsafe state.
- Suppose that, at time t1, process P2 requests and is allocated one more tape drive. The system is no longer in a safe state.
- At this point, only process P1 can be allocated all its tape drives. When it returns them, the system will have only four available tape drives.
- Since process *Po* is allocated five tape drives but has a maximum of ten,it may request five more tape drives. If it does so, it will have to wait, because they are unavailable.
- Similarly, process P2 may request six additional tape drives and have to wait, resulting in a deadlock.
- Our mistake was in granting the request from process P2 for one more tape drive. If we had made P2 wait until either of the other processes had finished and released its resources, then we could have avoided the deadlock.

	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

Deadlock avoidance

- Given the concept of a safe state, we can define avoidance algorithms that ensure that the system will never deadlock.
- The idea is simply to ensure that the system will always remain in a safe state. Initially, the system is in a safe state.
- Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait.
- The request is granted only if the allocation leaves the system in a safe state.

OPERATING SYSTEMS

Module3_Part11

Textbook : Operating Systems Concepts by Silberschatz

Resource-Allocation-Graph Algorithm

- If we have a resource-allocation system with only one instance of each resource type, we can use a resource-allocation graph for deadlock avoidance.
- In addition to the request and assignment edges already described, we introduce a new type of edge, called a claim edge.
- A claim edge *Pi -> Rj* indicates that process *Pi* may request resource *Rj* at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line.

When process *Pi* requests resource *Rj,* the claim edge *Pi* → *Rj* is converted to a request edge. Similarly, when a resource *Rj* is released by *Pi,* the assignment edge *Rj -> Pi* is reconverted to a claim edge *Pi* → *Rj.*

Resource-Allocation-Graph Algorithm

- We note that the resources must be claimed a priori in the system.

  That is, before process *Pi* starts executing, all its claim edges must already appear in the resource-allocation graph.

- Now suppose that process *Pi* requests resource *Rj.* The request can be granted only if converting the request edge *Pi* → *Rj* to an assignment edge *R1* → *Pi* does not result in the formation of a cycle in the resource-allocation graph.

- We check for safety by using a cycle-detection algorithm.

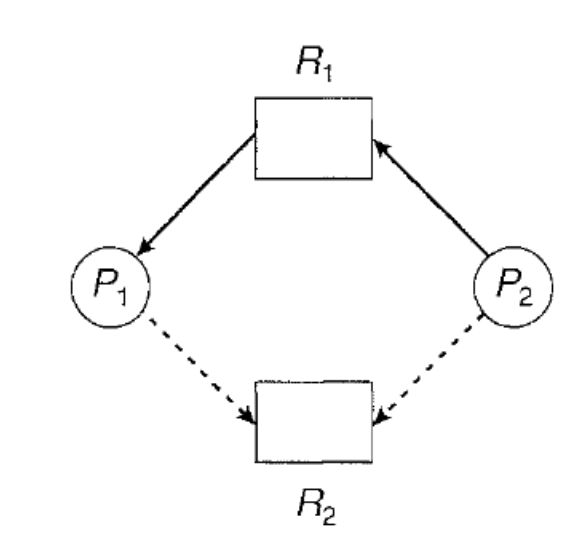
- If no cycle exists,

  then the allocation of the resource will leave the system in a safe state.

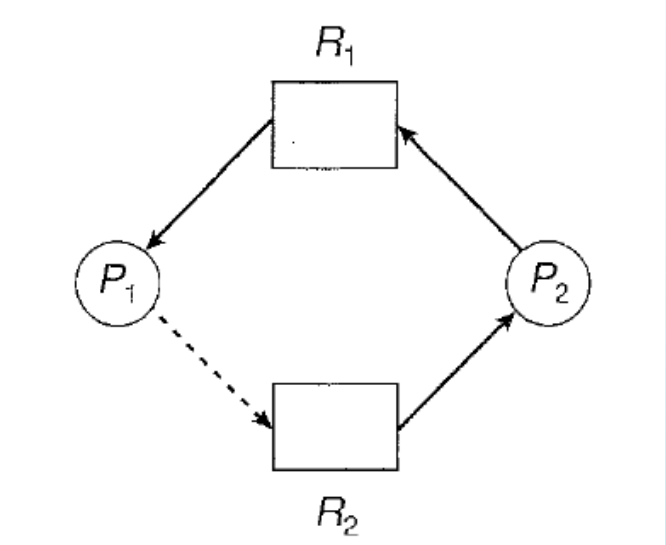
  If a cycle is found,

  then the allocation will put the system in an unsafe state. In that case, process *Pi* will have to wait for its requests to be satisfied.

Resource-Allocation-Graph Algorithm



a



b

Consider the resource-allocation graph ‘a’ . Suppose that *P2* requests R2 . Although R2 is currently free, we cannot allocate it to *P2*, since this action will create a cycle in the graph b. A cycle, as mentioned, indicates that the system is in an unsafe state. If P1 requests R2, and P2 requests R1, then a deadlock will occur.

Banker’s Algorithm

- The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type.

We can use an algorithm commonly known as the *banker's algorithm*.

- When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need.
- This number may not exceed the total number of resources in the system.
- When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state.
- If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Banker's Algorithm

- Several **data structures** must be maintained to implement the banker's algorithm.
- These data structures encode the state of the resource-allocation system.
- We need the following data structures, where *n* is the number of processes in the system and *m* is the number of resource types:
- Available. A vector of length *m* indicates the number of available resources of each type. If *Available[j]* equals *k*, then *k* instances of resource type *Rj* are available.
- Max. An *n* x *m* matrix defines the maximum demand of each process. If *Max[i] [j]* equals *k*, then process *Pi* may request at most *k* instances of resource type Rj.
- Allocation. An *n* x *m* matrix defines the number of resources of each type currently allocated to each process. If *Allocation[i][j]* equals k, then process *Pi* is currently allocated k instances of resource type *Rj.*
- Need. An *n* x *m* matrix indicates the remaining resource need of each process. If *Need[i][j]* equals *k*, then process *Pi* may need *k* more instances of resource type *Rj* to complete its task. Note that *Need[i][j]* equals *Max[i][j]- Allocation [i][j].*

Banker's algorithm

- To simplify the presentation of the banker's algorithm, we next establish some notation.
- Let X and Y be vectors of length *n.* We say that X<= *Y* if and only if *X[i]* <= Y[i] for all *i* = 1, 2, ... , *n.*
- For example, if X = (1,7,3,2) and *Y* =(0,3,2,1), then *Y* <=X.
- We can treat each row in the matrices *Allocation* and *Need* as vectors and refer to them as $Allocation_i$ and $Need_i$.
- The vector $Allocation_i$ specifies the resources currently allocated to process *Pi;*
- the vector $Need_i$ specifies the additional resources that process *Pi* may still request to complete its task.

Banker's algorithm

- Safety algorithm: whether or not a system is in a safe state.

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

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

2.Find an index *i* such that both

Finish[i] ==*false*

$Need_i$ <= *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.

Banker's Algorithm

Resource-Request Algorithm

- Next, we describe the algorithm for determining whether requests can be safely granted.
- Let Request*i* be the request vector for process *Pi*. If *Request[i] [j]* == *k*, then process *Pi* wants *k* instances of resource type *Rj*.

When a request for resources is made by process *Pi*,

the following actions are taken:

1. If Request*i* <= Need*i*, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If Request*i* <= *Available*, go to step 3. Otherwise, *Pi* must wait, since the resources are not available

Banker's Algorithm

3.Have the system pretend to have allocated the requested resources to process *Pi* by modifying the state as follows:

$$Available = Available - Request_i$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- If the resulting resource-allocation state is safe, the transaction is completed, and process *Pi* is allocated its resources.
- However, if the new state is unsafe, then *Pi* must wait for *Request;*, and the old resource-allocation state is restored.

An Illustrative Example

- To illustrate the use of the banker's algorithm, consider a system with 5 processes *Po* through *P4* and three resource types *A, B,* and C.
- Resource type *A* has 10 instances, resource type *B* has 5 instances, and resource type C has 7 instances.
- Suppose that, at time *T0* , the following snapshot of the system has been taken:
- The content of the matrix *Need* is defined to be *Max* - *Allocation* and is as follows:

	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	

	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

We claim that the system is currently in a safe state.
Indeed, the sequence < *P1*, P3, P4, P0, P2> satisfies the safety criteria.

Suppose now that process
P1 requests one additional instance of resource type *A* and two instances of resource type C, so *Request1* = (1,0,2). To decide whether this request can be immediately granted, we first check that *Request1* <= *Available-that* is, that (1,0,2) <= (3,3,2), which is true. We then pretend that this request has been fulfilled, and we arrive at the following new state:

	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	

We must determine whether this new system state is safe. To do so, we execute our safety algorithm and find that the sequence <P1, P3, P4, Po, P2> satisfies the safety requirement. Hence, we can immediately grant the request of process P1.

- You should be able to see, however, that when the system is in this state, a request for (3,3,0) by P4 cannot be granted, since the resources are not available.
- Furthermore, a request for (0,2,0) by *Po* cannot be granted, even though the resources are available, since the resulting state is unsafe.

m=3, n=5 — Step 1 of Safety Algo

Work = Available

Work = | 3 | 3 | 2 |

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

For i = 0 — Step 2

$Need_0$ = 7, 4, 3 (7,4,3 vs 3,3,2)

Finish [0] is false and $Need_0$ > Work

So P_0 must wait — But Need ≤ Work

For i = 1 — Step 2

$Need_1$ = 1, 2, 2 (1,2,2 vs 3,3,2)

Finish [1] is false and $Need_1$ < Work

So P_1 must be kept in safe sequence

Step 3

3, 3, 2 — 2, 0, 0

Work = Work + $Allocation_1$

	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 (6, 0, 0 vs 5,3, 2)

Finish [2] is false and $Need_2$ > Work

So P_2 must wait

For i=3 — Step 2

$Need_3$ = 0, 1, 1 (0, 1, 1 vs 5, 3, 2)

Finish [3] = false and $Need_3$ < Work

So P_3 must be kept in safe sequence

Step 3

5, 3, 2 — 2, 1, 1

Work = Work + $Allocation_3$

	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 (4, 3, 1 vs 7, 4, 3)

Finish [4] = false and $Need_4$ < Work

So P_4 must be kept in safe sequence

Step 3

7, 4, 3 — 0, 0, 2

Work = Work + $Allocation_4$

	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 (7, 4, 3 vs 7, 4, 5)

Finish [0] is false and Need < Work

So P_0 must be kept in safe sequence

Step 3

7, 4, 5 — 0, 1 , 0

Work = Work + $Allocation_0$

	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 (6, 0, 0 vs 7, 5, 5)

Finish [2] is false and $Need_2$ < Work

So P_2 must be kept in safe sequence

Step 3

7, 5, 5 — 3, 0, 2

Work = Work + $Allocation_2$

	A	B	C
Work =	10	5	7

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

Finish [i] = true for 0 ≤ i ≤ n — Step 4

Hence the system is in Safe state

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

A B C

$Request_1$ = 1, 0, 2

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

Step 1

1, 0, 2 < 1, 2, 2

$Request_1 < Need_1$ ✔

Step 2

1, 0, 2 < 3, 3, 2

$Request_1 < Available$ ✔

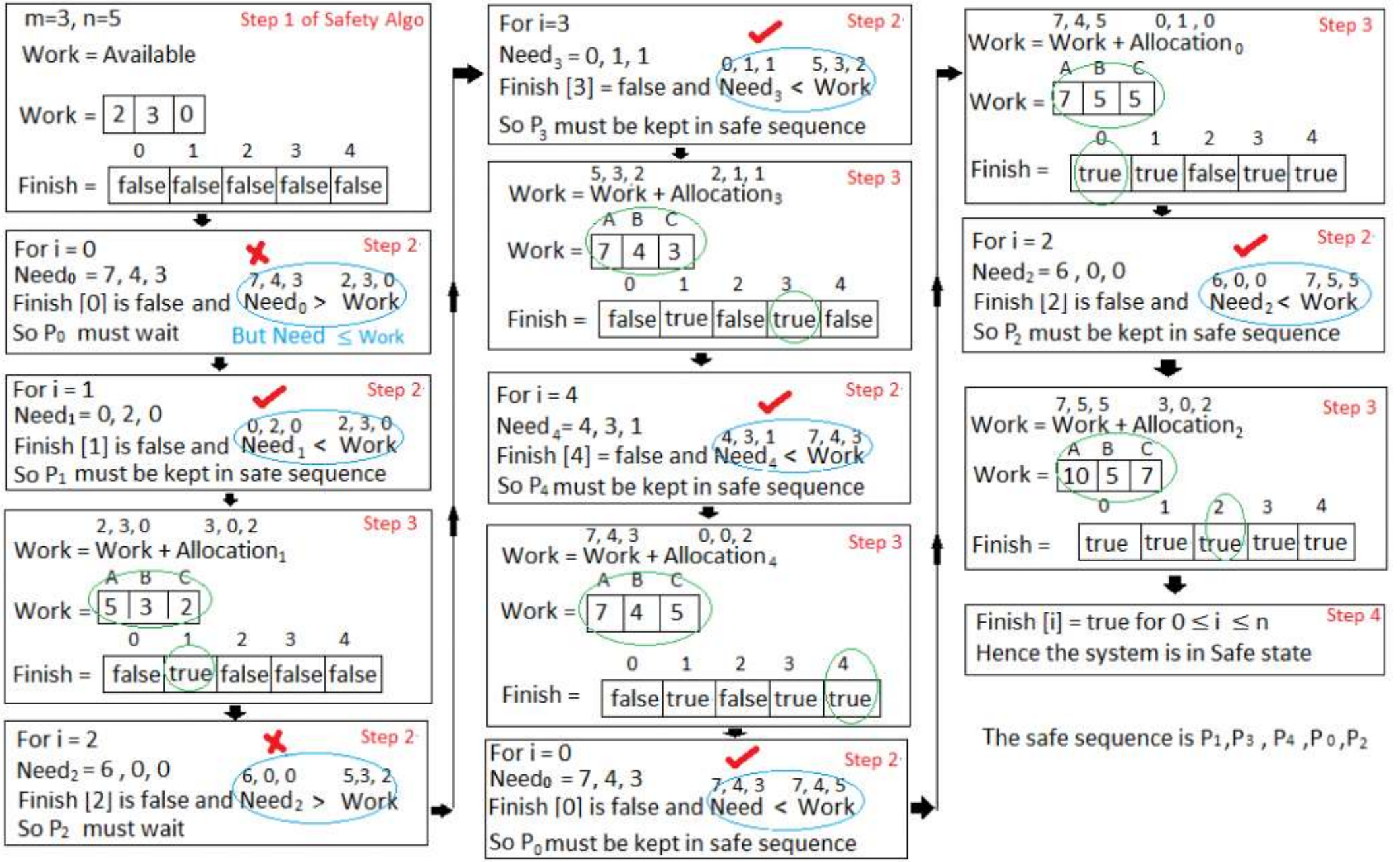
Step 3

$Available = Available - Request_1$

$Allocation_1 = Allocation_1 + Request_1$

$Need_1 = Need_1 - Request_1$

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
Need0 = 7, 4, 3
Finish [0] is false and Need0 > Work
7, 4, 3 2, 3, 0
So P0 must wait
But Need ≤ Work
Step 2
For i = 1
Need1 = 0, 2, 0
Finish [1] is false and Need1 < Work
0, 2, 0 2, 3, 0
So P1 must be kept in sate sequence
Step 2
2, 3, 0 3, 0, 2
Work = Work + Allocation1
A B C
Work = 5 3 2
0 1 2 3 4
Finish = false true false false false
Step 3
For i = 2
Need2 = 6 , 0, 0
Finish [2] is false and Need2 > Work
6, 0, 0 5,3, 2
So P2 must wait
Step 2
For i=3
Need3 = 0, 1, 1
Finish [3] = false and Need3 < Work
0, 1, 1 5, 3, 2
So P3 must be kept in safe sequence
Step 2
5, 3, 2 2, 1, 1
Work = Work + Allocation3
A B C
Work = 7 4 3
0 1 2 3 4
Finish = false true false true false
Step 3
For i = 4
Need4 = 4, 3, 1
Finish [4] = false and Need4 < Work
So P4 must be kept in safe sequence
4, 3, 1 7, 4, 3
Step 2
7, 4, 3 0, 0, 2
Work = Work + Allocation4
A B C
Work = 7 4 5
0 1 2 3 4
Finish = false true false true true
Step 3
For i = 0
Need0 = 7, 4, 3
Finish [0] is false and Need < Work
So P0 must be kept in safe sequence
7, 4, 3 7, 4, 5
Step 2
7, 4, 5 0, 1 , 0
Work = Work + Allocation0
A B C
Work = 7 5 5
0 1 2 3 4
Finish = true true false true true
Step 3
For i = 2
Need2 = 6 , 0, 0
Finish [2] is false and Need2 < Work
So P2 must be kept in safe sequence
6, 0, 0 7, 5, 5
Step 2
7, 5, 5 3, 0, 2
Work = Work + Allocation2
A B C
Work = 10 5 7
0 1 2 3 4
Finish = true true true true true
Step 3
Finish [i] = true for 0 ≤ i ≤ n
Hence the system is in Safe state
Step 4
The safe sequence is P1 ,P3 , P4 ,P0 ,P2

OPERATING SYSTEMS

Module3_Part12

Textbook : Operating Systems Concepts by Silberschatz

Deadlock Detection

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur.

We have to find out whether deadlock is there or not using deadlock detection algorithm

Deadlock detection for systems with

- Single Instance of Each Resource Type
- Several instances of each resource type

Deadlock detection for systems with Single Instance of Each Resource Type

- we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a *wait-for* graph.
- We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.
- More precisely, an edge from *Pi* → *Pj* in a wait-for graph implies that process *Pi* is waiting for process *Pj* to release a resource that *Pi* needs.
- An edge *Pi* → *Pj* exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges *Pi* → *Rq* and *Rq* → *Pj*for some resource *Rq*.

Deadlock detection for systems with Single Instance of Each Resource Type

For example, we present a resource-allocation graph and the corresponding wait-for graph.

- As before, a deadlock exists in the system if and only if the wait-for graph contains a cycle.
- To detect deadlocks, the system needs to *maintain* the wait-for graph and periodically *invoke an algorithm* that searches for a cycle in the graph.

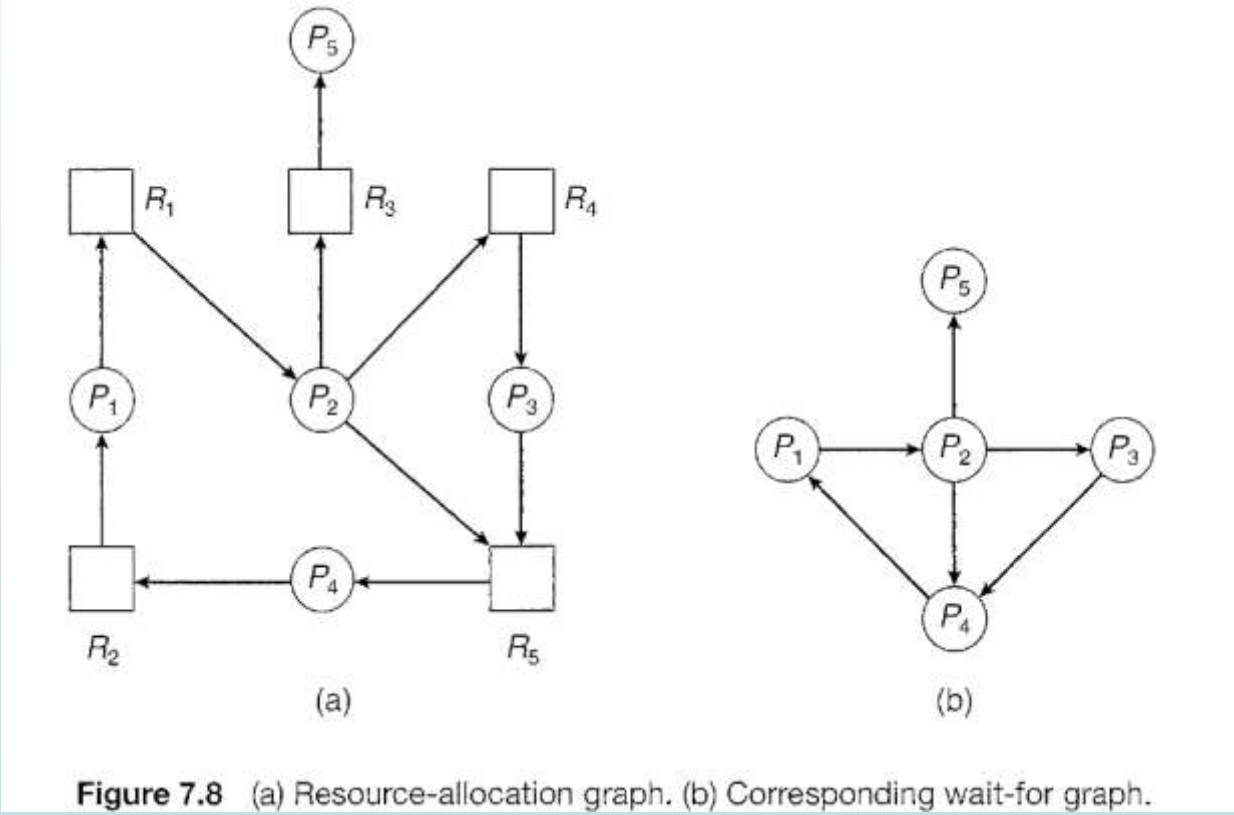


Figure 7.8 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

Deadlock detection for systems with Several instances of each resource type

- The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm
- **Available**. A vector of length *n* indicates the number of available resources of each type.
- **Allocation**. An n x *n* 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]* equals *k*, then process *Pi* is requesting *k* more instances of resource type *Rj.*
- ***Allocation*** and ***Request*** as vectors; we refer to them as $\boldsymbol{Allocation_i}$ and $\boldsymbol{Request_i}$. The detection algorithm described here simply investigates every possible allocation sequence for the processes that remain to be completed.

Deadlock detection for systems with Several instances of each resource type

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize $Work = Available$. For $i = 0, 1, ..., n\text{-}1$, 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, $0 \leq i < n$, then the system is in a deadlocked state. Moreover, if $Finish[i] == false$, then process P_i is deadlocked.

Example

- To illustrate this algorithm, we consider a system with five processes *Po* through *P4* and three resource types *A, B,* and C.
- Resource type *A* has seven instances, resource type *B* has two instances, and resource type C has six instances.
- Suppose that, at time *T0*, we have the following resource-allocation state:

	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	

We claim that the system is not in a deadlocked state. Indeed, if we execute our algorithm, we will find that the sequence <*Po, P2, P3, Pl, P4*> results in *Finish[i] == true* for all *i.*

Example

- Suppose now that process *P2* makes one additional request for an instance
- of type C. The *Request* matrix is modified as follows:

	Allocation	*Available*	*Request*
	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 1
P_3	2 1 1		1 0 0
P_4	0 0 2		0 0 2

We claim that the system is now deadlocked. Although we can reclaim the resources held by process *Po*, the number of available resources is not sufficient to fulfill the requests of the other processes. Thus, a deadlock exists, consisting of processes P1, *P2*, *P3*, and *P4*.