Module 3

Process Synchronization

A situation 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**.

Producer-Consumer Problem

Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process.

To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced. In this situation, the consumer must wait until an item is produced.

✦ *unbounded-buffer* places no practical limit on the size of the buffer.

✦ *bounded-buffer* assumes that there is a fixed buffer size.

Bounded-Buffer – Shared-Memory Solution

The consumer and producer processes share the following variables.
Shared data

```
#define BUFFER_SIZE 10
Typedef struct
 {
. . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Solution is correct, but can only use BUFFER_SIZE-1 elements.
The shared buffer is implemented as a circular array with two logical pointers: in and out. The variable in points to the next free position in the buffer; out points to the first full position in the buffer. The buffer is empty when in == out ; the buffer is full when ((in + 1) % BUFFERSIZE) == out.

The code for the producer and consumer processes follows. The producer process has a local variable `nextproduced` in which the new item to be produced is stored:

Bounded-Buffer – Producer Process

```
item nextProduced;
while (1)
{
while (((in + 1) % BUFFER_SIZE) == out)
; /* do nothing */
buffer[in] = nextProduced;
in = (in + 1) % BUFFER_SIZE;
}
```

Bounded-Buffer – Consumer Process

```
item next Consumed;
while (1)
```

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

The critical section problem

Consider a system consisting of n processes {Po,P1, ..., Pn-1). Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. Thus, the execution of critical sections by the processes is **mutually exclusive** in time. 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.

```
do{
        Entry section
                Critical section
        Exit section
                Remainder section
}while(1);
```

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 and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.
3. Bounded Waiting: There exists a bound 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

Peterson’s solution is a software based solution to the critical section problem.

Consider two processes P0 and P1. For convenience, when presenting Pi, we use Pi to denote the other process; that is, j == 1 - i.

The processes share two variables:

```
boolean flag [2] ;
int turn;
```

Initially flag [0] = flag [1] = false, and the value of turn is immaterial (but is either 0 or 1). The structure of process Pi is shown below.

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

To enter the critical section, process Pi first sets flag [il to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section it can do so. If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time. Only one of these assignments

will last; the other will occur, but will be overwritten immediately. The eventual value of turn decides which of the two processes is allowed to enter its critical section first.
We now prove that this solution is correct. We need to show that:
1. Mutual exclusion is preserved,
2. The progress requirement is satisfied,
3. The bounded-waiting requirement is met.

To prove property 1, we note that each Pi enters its critical section only if either flag [jl == false or turn == i. Also note that, if both processes can be executing in their critical sections at the same time, then flag [i] ==flag [jl == true. These two observations imply that P0 and P1 could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1, but cannot be both. Hence, one of the processes say Pj-must have successfully executed the while statement, whereas Pi had to execute at least one additional statement ("turn == j"). However, since, at that time, flag [j] == true, and turn == j, and this condition will persist as long as Pi is in its critical section, the result follows:

To prove properties 2 and 3, we note that a process Pi can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag [j] == true and turn == j; this loop is the only one. If Pi is not ready to enter the critical section, then flag [j] == false and Pi can enter its critical section. If Pi has set flag[j] to true and is also executing in its while statement, then either turn == i or turn == j. If turn == i, then Pi will enter the critical section. If turn == j, then Pi will enter the critical section. However, once Pi exits its critical section, it will reset flag [jl to false, allowing Pi to enter its critical section. If Pi resets flag [j 1 to true, it must also set turn to i.

Thus, since Pi does not change the value of the variable turn while executing the while statement, Pi will enter the critical section (progress) after at most one entry by Pi (bounded waiting).

Synchronization Hardware

As with other aspects of software, hardware features can make the programming task easier and improve system efficiency. In this section, we present some simple hardware instructions that are available on many systems, and show how they can be used effectively in solving the critical-section problem.

The definition of the TestAndSet instruction.

```
boolean TestAndSet(boo1ean &target)

 {
        boolean rv = target;
        target = true; return
        rv;
}
```

The critical-section problem could be solved simply in a uniprocessor environment if we could forbid interrupts to occur while a shared variable is being modified. In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable.

Unfortunately, this solution is not feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time-consuming, as the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases. Also, consider the effect on a system's clock, if the clock is kept updated by interrupts.

Many machines therefore 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 uninterruptible unit. We can use these special instructions to solve the critical-section problem in a relatively simple manner. Rather than discussing one specific instruction for one specific machine, let us abstract the main concepts behind these types of instructions.
The TestAndSet instruction can be defined as shown in code. The important characteristic is that 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 arbitrary order.
Mutual-exclusion implementation with **`TestAndSet`**

```
do{

        while(TestAndSet(lock));
```

```
                critical section
        lock=false
                Remainder section
}while(1);

void Swap(boo1ean &a, boolean &b) {
boolean temp = a;
a = b;
b = temp}
```

If the machine supports the TestAndSet instruction, then we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false.

If the machine supports the Swap instruction, then mutual exclusion can be provided as follows. A global Boolean variable lock is declared and is initialized to false. In addition, each process also has a local Boolean variable key.

Semaphores

The solutions to the critical-section problem presented before are not easy to generalize to more complex problems. To overcome this difficulty, we can use a synchronization tool called a semaphore. A **semaphore** S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: **wait** and **signal**. These operations were originally termed P (for wait; from the Dutch proberen, to test) and V (for signal; from verhogen, to increment). The classical definition of wait in pseudocode is

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

The classical definitions of signal in pseudocode is

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

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 the wait (S), the testing of the integer value of S (S 5 O), and its possible modification (S--), must also be executed without interruption.

Usage

We can use semaphores to deal with the n-process critical-section problem. The n processes share a semaphore, mutex (standing for mutual exclusion), initialized to 1. Each process Pi is organized as shown in Figure.We can also use semaphores to solve various synchronization problems.
For example, consider two concurrently running processes: PI with a statement S1 and P2 with a statement S2. Suppose that 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 in process PI, and the statements

```
wait (synch)
; s2;
```

in process P2. Because synch is initialized to 0, P2 will execute S2 only after PI
has invoked signal (synch) , which is after S1.

```
s1;
signal (synch)
; do {
        wait (mutex) ;
                critical section
        signal (mutex) ;
                remainder section
} while (1);
```
Mutual-exclusion implementation with semaphores.

Implementation

The main disadvantage of the mutual-exclusion solutions and of the semaphore definition given here, is that they all require 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. This type of semaphore is also called a spinlock (because the process "spins" while waiting for the lock). Spinlocks are useful in multiprocessor systems. The advantage of a spinlock is that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. Thus, when locks are expected to be held for short times, spinlocks are useful.

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 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.

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. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)
To implement semaphores under this definition, we define a semaphore as a "C" struct:

```
typedef struct {

int value ;
struct process *L;
) semaphore;
```

Each semaphore has an integer value and a list of processes. 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

```
void wait(semaphore S)
{ S.value--;
if (S.value < 0) {
add this process to S .
L; block() ;
}
```
The signal semaphore operation can now be defined
```
as void signal(semaphore S) {
S.value++;
if (S.value <= 0) {
remove a process P from S . L ;
wakeup (PI) ;
}
```

The block operation suspends the process that invokes it. The wakeup(P1) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls.

Binary Semaphores

The semaphore construct described in the previous sections is commonly known as a counting semaphore,

since its integer value can range over an unrestricted domain. A binary semaphore is a semaphore with an integer value that can range only between 0 and 1. A binary semaphore can be simpler to implement than a counting semaphore, depending on the underlying hardware architecture. We will now show how a counting semaphore can be implemented using binary semaphores. Let S be a counting semaphore.
To implement it in terms of binary semaphores we need the following data structures:

```
binary-semaphore S1, S2;


int C;
```

Initially S1 = 1, S2 = 0, and the value of integer C is set to the initial value of the counting semaphore S.

The wait operation on the counting semaphore S can be implemented as follows:

```
wait (S1) ;
C--;
i f (C < 0) {
signal(S1) ;
wait (S2) ;
}
signal(S1);
```

The signal operation on the counting semaphore S can be implemented as follows:

```
w a i t (S1) ;
C++ ;
i f (C <= 0)
signal (S2)
; e l s e
signal (S1)
;
```

Classic Problems of Synchronization

We present a number of different synchronization problems as examples for a large class of concurrency-control problems. These problems are used for testing nearly every newly proposed synchronization scheme.Semaphores are used for synchronization in our solutions.

The Bounded-Buffer Problem

The bounded-buffer problem is commonly used to illustrate the power of synchronization primitives. We present here 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, respectively. The semaphore empty is initialized to the value n; the semaphore f u l l is initialized to the value 0.
The code for the producer process is

```
do{
produce an item in nextp
...
wait (empty) ;
wait (mutex) ;
...
add nextp to buffer
. . .
signal(mutex);
signal (full) ;
) while (1);
```

The code for the consumer process is

```
do{
wait (full) ;
wait (mutex)
;
. . .
remove an item from buffer to nextc
...
signal (mutex) ;
signal (empty) ;
...
consume the item in nextc
...
) while (1);
```

Note the symmetry between the producer and the consumer. 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

A data object (such as a file or record) is to be shared among several concurrent processes. Some of these processes may want only to read the content of the shared object, whereas others may want to update (that is, to read and write) the shared object. We distinguish between these two types of processes by referring to those processes that are interested in only reading as readers, and to the rest as writers. Obviously, if two readers access the shared data object simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the shared object simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared object. This synchronization problem is referred to as the readers-writers problem. Since it was originally stated, it has been used to test nearly every new synchronization primitive. The readers-writers problem has several variations, all involving priorities. The simplest one, referred to as the first readers-writers problem, requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting. The second readers-writers problem requires that, once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve. For this reason, other variants of the problem have been proposed. In this section, we present a solution to the first 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 w r t are initialized to 1; readcount is initialized to 0. The semaphore w r t is common to both the 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 code for a writer process is

```
do{
wait (wrt) ;
. . .
writing is performed
...
signal(wrt);
}while(1);
```

The code for a reader process is

```
do{
wait (mutex) ;
readcount++;
if (readcount ==
1) wait (wrt) ;
signal (mutex) ;
. . .
reading is performed
...
wait (mutex) ;
readcount--;
if (readcount ==
0) signal(wrt1;
signal (mutex) ;
}while(1);
```

Note that, if a writer is in the critical section and n readers are waiting, then one reader is queued on wrt, and n - 1 readers are queued on mutex. Also observe that, when a writer executes signal (wrt), we may resume the execution of either the waiting readers or a single waiting writer.

The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a common 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 chopsticks. 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 (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.

The structure of philosopher i

```
do {

wait (chopstick[i]) ;
wait (chopstick[(i+1) % 5] ) ;
...
eat
. . .
signal (chopstick [i] ;
signal(chopstick[(i+1) % 5] )
;
. . .
think
...
) while (1);
.
```

The dining-philosophers problem is considered a classic synchronization problem, neither because of its practical importance nor because computer scientists dislike philosophers, but because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock- and starvation free manner.

One simple solution is to represent each chopstick by a semaphore. A philosopher tries to grab the chopstick by executing a wait operation on that semaphore; she releases her chopsticks by executing the signal operation on the appropriate semaphores. Thus, the shared data are semaphore chopstick [5] ; where all the elements of chopstick are initialized to 1.

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it has the possibility of creating a deadlock. Suppose that all five philosophers become hungry simultaneously, and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

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.Finally, any satisfactory solution to the dining-philosophers problem must guard against the possibility that one of the philosophers will starve to death.

Deadlocks

A set of process is in a deadlock state if each process in the set is waiting for an event that can be caused by only another process in the set. In other words, each member of the set of deadlock processes is waiting for a resource that can be released only by a deadlock process. None of the processes can run, none of them can release any resources, and none of them can be awakened.

The resources may be either physical or logical. Examples of physical resources are Printers, Hard Disc Drives, Memory Space, and CPU Cycles. Examples of logical resources are Files, Semaphores, and Monitors.

The simplest example of deadlock is where process 1 has been allocated non-shareable resources A (say a Hard Disc drive) and process 2 has be allocated non-sharable resource B (say a printer). Now, if it turns out that process 1 needs resource B (printer) to proceed and process 2 needs resource A (Hard Disc drive) to proceed and these are the only two processes in the system, each is blocked the other and all useful work in the system stops. This situation is termed deadlock. The system is in deadlock state because each process holds a resource being requested by the other process neither process is willing to release the resource it holds.

Preemptable and Nonpreemptable Resources

Resources come in two flavors: preemptable and nonpreemptable.

A preemptable resource is one that can be taken away from the process with no ill effects. Memory is an example of a preemptable resource.

A nonpreemptable resource is one that cannot be taken away from process (without causing ill effect). For example, CD resources are not preemptable at an arbitrary moment.

Reallocating resources can resolve deadlocks that involve preemptable resources. Deadlocks that involve nonpreemptable resources are difficult to deal with.

Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. **Request:** If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.
2. **Use:** The process can operate on the resource.
3. **Release:** The process releases the resource.

Necessary Conditions for Deadlock

Coffman (1971) identified four conditions that must hold simultaneously for there to be a deadlock.

1. **Mutual Exclusion Condition**: The resources involved are non-shareable.

   Explanation: At least one resource must be held in a non-shareable mode, that is, only one process at a time claims exclusive control of the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. **Hold and Wait Condition:** Requesting process hold already the resources while waiting for requested resources.

   Explanation: There must exist a process that is holding a resource already allocated to it while waiting for additional resource that are currently being held by other processes.

4. **No-Preemptive Condition:** Resources already allocated to a process cannot be preempted.

   Explanation: Resources cannot be removed from the processes are used to completion or released voluntarily by the process holding it.

4. **Circular Wait Condition**: The processes in the system form a circular list or chain where each process in the list is waiting for a resource held by the next process in the list. There exists a set {P0, P1, …, P0} of waiting processes such that P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, …, Pn–1 is waiting for a resource that is held by Pn, and P0 is waiting for a resource that is held by P0.

Note: It is not possible to have a deadlock involving only one single process. The deadlock involves a circular "hold-and-wait" condition between two or more processes, so "one" process cannot hold a resource, yet be waiting for another resource that it is holding. In addition, deadlock is not possible between two threads in a process, because it is the process that holds resources, not the thread that is, each thread has access to the resources held by the process.

Resource-Allocation Graph

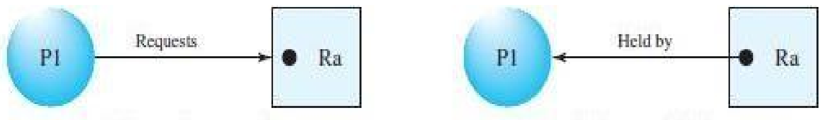
Deadlocks can be described in terms of a directed graph called a system resource-allocation graph

This graph consists of a set of vertices V and a set of edges E. The set of vertices V is partitioned int two different types of nodes P = {PI, P2, ..., Pn}, the set consisting of all the active processes in the system, and R = {R1, R2, ..., Rm}, the set consisting of all resource types in the system.

A directed edge from process Pi to resource type Rj is denoted by Pi Rj; it signifies that process Pi requested an instance of resource type Rj and is currently waiting for that resource. A directed edge Pi Rj is called a request edge.

A directed edge from resource type Rj to process Pi is denoted by Rj Pi; it signifies that an instance of resource type Rj has been allocated to process Pi. A directed edge Rj Pi is called an assignment edge.

Pictorially, we represent each process Pi as a circle and each resource type Rj as a square. Since resource type Rj may have more than one instance, we represent each such instance as a dot within the square. A request edge points to only the square Rj, whereas an assignment edge must designate one of the dots in the square.



(a) Resource is requested (b) Resource is held

The resource-allocation graph shown below depicts the following situation.

The sets P, R, and E:

P={P1,P2,P3}

R={R1,R2,R3,R4}

E={P1→R1, P2→R3, R1→P2, R2→P2, R2→P1, 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

Process states:

Process PI is holding an instance of resource type R2, and is waiting for an instance of resource type R1.

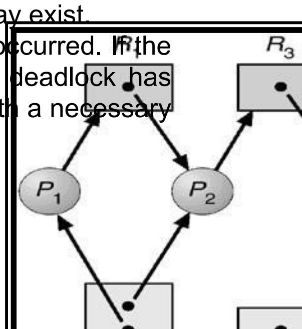
Process P2 is holding an instance of R1 and R2, and is waiting for an instance of resource type R3.

Process P3 is holding an instance of R3.

Example of a 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.



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, let us 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 graph. At this point, two minimal cycles exist in the system:

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

Resource Allocation Graph with a deadlock

Now consider the resource-allocation graph in the following Figure. In this example, we also have a cycle 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.

Resource Allocation Graph with a Cycle but No Deadlock

METHODS OF HANDLING DEADLOCK

In general, there are four strategies of dealing with deadlock problem:

1. **Deadlock Prevention**: Prevent deadlock by resource scheduling so as to negate at least one of the four conditions.
2. **Deadlock Avoidance**: Avoid deadlock by careful resource scheduling.
3. **Deadlock Detection and Recovery**: Detect deadlock and when it occurs, take steps to recover.
4. **The Ostrich Approach:** Just ignore the deadlock problem altogether.

DEADLOCK PREVENTION

A deadlock may be prevented by denying any one of the conditions.

<u>Elimination of "Mutual Exclusion" Condition:</u> The mutual exclusion condition must hold for non-sharable resources. That is, several processes cannot simultaneously share a single resource. This condition is difficult to eliminate because some resources, such as the Hard disc drive and printer, are inherently non-shareable. Note that shareable resources like read-only-file do not require mutually exclusive access and thus cannot be involved in deadlock.

<u>Elimination of "Hold and Wait" Condition:</u> There are two possibilities for elimination of the second condition. The first alternative is that a process request be granted all of the resources it needs at once, prior to execution. The second alternative is to disallow a process from requesting resources whenever it has previously allocated resources. This strategy requires that all of the resources a process will need must be requested at once. The system must grant resources on "all or none" basis. If the complete set of resources needed by a process is not currently available, then the process must wait until the complete set is available. While the process waits, however, it may not hold any resources. Thus the "wait for" condition is denied and deadlocks cannot occur. This strategy can lead to serious waste of resources.

<u>Elimination of "No-preemption" Condition:</u> The non-preemption condition can be alleviated by forcing a process waiting for a resource that cannot immediately be allocated to relinquish all of its currently held resources, so that other processes may use them to finish. This strategy requires that when a process that is holding some resources is denied a request for additional resources. The process must release its held resources and, if necessary, request them again together with additional resources. Implementation of this strategy denies the "no-preemptive" condition effectively.

<u>Elimination of "Circular Wait" Condition:</u> The last condition, the circular wait, can be denied by imposing a total ordering on all of the resource types and then forcing, all processes to request the resources in order (increasing or decreasing). This strategy impose a total ordering of all resources types, and to require that each process requests resources in a numerical order (increasing or decreasing) of enumeration. With this rule, the resource allocation graph can never have a cycle.

For example, provide a global numbering of all the resources, as shown

1	≡	Card reader
2	≡	Printer
3	≡	Optical driver
4	≡	HDD
5	≡	Card punch

Now the rule is this: processes can request resources whenever they want to, but all requests must be made in numerical order. A process may request first printer and then a HDD(order: 2, 4), but it may not request first a optical driver and then a printer (order: 3, 2). The problem with this strategy is that it may be impossible to find an ordering that satisfies everyone.

DEADLOCK AVOIDANCE

This approach to the deadlock problem anticipates deadlock before it actually occurs. This approach employs an algorithm to access the possibility that deadlock could occur and acting accordingly. If the necessary conditions for a deadlock are in place, it is still possible to avoid deadlock by being careful when resources are allocated. It employs the most famous deadlock avoidance algorithm that is the Banker's algorithm.

A 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 and Unsafe States

A system is said to be in a **Safe State**, if there is a safe execution sequence. An execution sequence is an ordering for process execution such that each process runs until it terminates or blocked and all request for resources are immediately granted if the resource is available.

A system is said to be in an **Unsafe State**, if there is no safe execution sequence. An unsafe state may not be deadlocked, but there is at least one sequence of requests from processes that would make the system deadlocked.

(Relation between Safe, Unsafe and Deadlocked States)

Resource-Allocation Graph Algorithm

The deadlock avoidance algorithm uses a variant of the resource-allocation graph to avoid deadlocked state. It introduces 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 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.

Suppose that process Pi requests resource Rj. The request can be granted only if converting the request edge Pi   Rj to an assignment edge Rj   Pi that does not result in the formation of a cycle in the resource-allocation graph. An algorithm for detecting a cycle in this graph is called 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. Therefore, process *Pi* will have to wait for its requests to be satisfied.

Banker's algorithm

The **Banker's algorithm** is a resource allocation & deadlock avoidance algorithm developed by Edsger Dijkstra that test for safety by simulating the allocation of pre-determined maximum possible amounts of all resources. Then it makes a "safe-state" check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue.

The Banker's algorithm is run by the operating system whenever a process requests resources. The algorithm prevents deadlock by denying or postponing the request if it determines that accepting the request could put the system in an unsafe state (one where deadlock could occur).

For the Banker's algorithm to work, it needs to know three things:

How much of each resource could possibly request by each process.
How much of each resource is currently holding by each process.
How much of each resource the system currently has available.

Resources may be allocated to a process only if it satisfies the following conditions:

1. request ≤ max, else set error as process has crossed maximum claim made by it.
2. request ≤ available, else process waits until resources are available.

Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system. Let n be the number of processes in the system and m be the number of resource types. We need the following data structures:
Available: A vector of length m indicates the number of available resources of each type. If Available[j] = k, there are k instances of resource type Rj available.
Max: An n x m matrix defines the maximum demand of each process. If *Max[i,j]* = 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]* = 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]* = k, then process Pi may need k more instances of resource type *Ri* to complete its task. Note that *Need[i,j] = Max[i,j] - Allocafion[i,j].*
These data structures vary over time in both size and value. 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.

Safety Algorithm

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

```
1. Let Work and Finish be vectors of length m and n, respectively. Initialize
   Work := Available and Finisk[i] :=false for i = 1,2, ..., n.
2. Find an i such that both a. Finisk[i] =false
                            b. $Need_i \leq Work$.
   If no such i exists, go to step 4.
3. Work := Work + $Allocation_i$
   Finisk[i] := true
   go to step 2.
4. If Finish[i] = true for all i, then the system is in a safe state.
```

This algorithm may require an order of *m* x n^2 operations to decide whether a state is safe.

Resource-Request Algorithm

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 $Request_i \leq Need_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $Request_i \leq Available$, go to step 3. Otherwise, *Pi* must wait, since the resources are not available.
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 $Request_i$ and the old resource-allocation state is restored.

Consider a system with five processes P0 through P4 and three resource types A,B,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:

Allocation	Max	Available
------------	--------	-----------
A B C	A B C	A B C

P0	0 1 0	7 5 3	3 3 2
PI	2 0 0	3 2 2	
P2	3 0 2	9 0 2	
P3	2 1 1	2 2 2	
P4	0 0 2	4 3 3	

	A B C
P0	7 4 3
P1	1 2 2
P2	6 0 0
P3	0 1 1
P4	4 3 1

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

is Need

We claim that the system is currently in a safe state. Indeed, the sequence <PI, P3, P4, P2, P0> 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, (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
P0	0 1 0	7 4 3	2 3 0
PI	3 0 2	0 2 0	
P2	3 0 2	6 0 0	
P3	2 1 1	0 1 1	
P4	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 <PI, P3, P4, P0, P2> satisfies our safety requirement. Hence, we can immediately grant the request of process PI.

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. A request for (0,2,0) by Po cannot be granted, even though the resources are available, since the resulting state is unsafe.

DEADLOCK DETECTION

If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system must provide:

- An algorithm that examines the state of the system to determine whether a deadlock has occurred.
- An algorithm to recover from the deadlock.

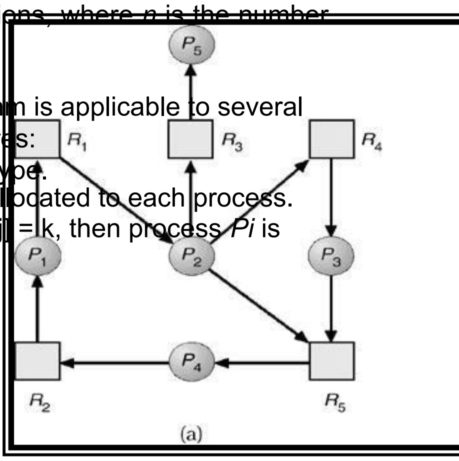
According to number of instances in each resource type, the Deadlock Detection algorithm can be classified into two categories as follows:

1. Single Instance of Each Resource Type: If all resources have only a single instance, then it can define a deadlock detection algorithm that uses a variant of the resource-allocation graph (is called a *wait-for* graph). A wait–for graph can be draw by removing the nodes of type resource and collapsing the appropriate edges from the resource-allocation graph.

An edge from Pi to 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. For Example:

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

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 to 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 the graph.



2. Several Instances of a Resource Type: The following deadlock-detection algorithm is applicable to several instance of a resource type. The algorithm employs several time-varying data structures:
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 *Pi* is requesting *k* more instances of resource type Rj.

The detection algorithm is described as follows:

```
1. Let Work and Finish be vectors of length m and n,
   respectively. Initialize, Work := Available. For i = 1, 2, ..., n,
   if $Allocation_i$ != 0, then Finish[i] :=false; otherwise, Finish[i] := true.
2. Find an index i such that both
      a. Finish[i] =false
```

state.

```
      b. $Request_i \leq Work$.
   If no such i exists, go to step 4. Work := Work +
   $Allocation_i$ Finish[i] := true
   go to step 2.
3. If Finish[i] = false, for some
   i, $1 \leq i \leq n$, then the system
   is in a deadlock state. if
   Finish[i] =false, then
   process Pi is deadlocked.
```

This algorithm requires an order of $m \times n^2$ operations to detect whether the system is in a deadlocked

RECOVERY FROM DEADLOCK

When a detection algorithm determines that a deadlock exists, then the system or operator is responsible for handling deadlock problem. There are two options for breaking a deadlock.

1. Process Termination
2. Resource preemption

Process Termination

There are two method to eliminate deadlocks by terminating a process as follows:

1. **Abort all deadlocked processes:** This method will break the deadlock cycle clearly by terminating all process. This method is cost effective. And it removes the partial computations completed by the processes.
2. **Abort one process at a time until the deadlock cycle is eliminated:** This method terminates one process at a time, and invokes a deadlock-detection algorithm to determine whether any processes are still deadlocked.

Resource Preemption

In resource preemption, the operator or system preempts some resources from processes and give these resources to other processes until the deadlock cycle is broken.

If preemption is required to deal with deadlocks, then three issues need to be addressed:

1. **Selecting a victim:** The system or operator selects which resources and which processes are to be preempted based on cost factor.
2. **Rollback:** The system or operator must roll back the process to some safe state and restart it from that state.
3. **Starvation:** The system or operator should ensure that resources will not always be preempted from the same process?