Unit – III

SYNCHRONIZATION

DEAD LOCKS

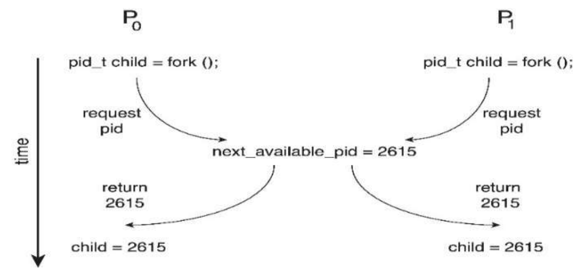
Synchronization

A cooperating process is one that can affect or be affected by other processes executing in the system. Cooperating process can either directly share a logical address space or be allowed to shared data only through files or messages.

Concurrent access to shared data may result in data inconsistency. Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

Race Condition

- Processes P_0 and P_1 are creating child processes using the fork() system call. Here, we allowed both the processes to manipulate the variable next_available_pid 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**.
- Race condition on kernel variable next_available_pid which represents the next available process identifier (pid).
- To guard against the race condition above, we ensure that only one process at a time can be manipulating the variable next_available_pid.



- Unless there is a mechanism to prevent P0 and P1 from accessing the variable next_available_pid the same pid could be assigned to two different processes!
- To make such a guarantee, we require that the processes be synchronized in some way.

1. The Critical Section Problem

- Consider a system consisting of n processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has a segment of code, called a **critical section**
  - Process may be changing common variables, updating table, writing file, etc.
  - When one process in critical section, no other may be in its critical section
- Critical section problem is to design protocol to solve this
- Each process must ask permission to enter critical section in entry section, may follow critical section with exit section, then remainder section
- General Structure of process $\mathbf{P_i}$

```
while (true) {

      entry section

          critical section

      exit section

          remainder section

}
```

Requirements for solution to critical-section problem

- **Mutual Exclusion -** If process Pi is executing in its critical section, then no other processes can be executing in their critical sections
- **Progress -** If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the process that will enter the critical section next cannot be postponed indefinitely
- **Bounded Waiting -** A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
    - Assume that each process executes at a nonzero speed
    - No assumption concerning relative speed of the n processes
- Two general approaches are used to handle critical sections in operating systems: preemptive kernels and non-preemptive kernels.
- A preemptive kernel allows a process to be preempted while it is running in kernel mode. A non-preemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exists kernel mode.

2. Peterson's Solution

- Peterson solution is one of the solutions to critical section problem involving two processes. This solution states that when one process is executing its critical section then the other process executes the rest of the code and vice versa.
- Peterson solution requires two shared data items:
  1) turn: indicates whose turn it is to enter into the critical section. If turn == i , then process i is allowed into their critical section.
  2) flag: indicates when a process wants to enter into critical section. When process i wants to enter their critical section, it sets flag[i] to true.

```
while (true){

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

          /* critical section */

       flag[i] = false;

       /* remainder section */

}
```

Correctness of Peterson's Solution

Provable that the three critical-section problem requirements are met.

- **Mutual Exclusion is preserved:**
  P_i enters CS only if:
  either flag[j] = false or turn = i
- **Progress requirement is satisfied:**
  A process P_i can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag[j] = true or turn = j
- **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.

3. Mutex Locks

- Operating System designers build software tools to solve critical section problem.
- Simplest is mutex lock. Boolean variable indicating if lock is available or not
- Protect a critical section by
    - First **acquire**() a lock
    - Then **release**() the lock
- Calls to **acquire**() and **release**() must be atomic
    - Usually implemented via hardware atomic instructions such as compare-and-swap.
- Disadvantage: It requires **busy waiting**. While a process is in critical section, any other process that tries to enter its critical section must loop continuously in the call to **acquire**().
- This lock therefore called a **spinlock** because the process "spins" while waiting for the lock to become available.

Solution to Critical Section Problem using Mutex Locks

-
  ```
  while(true)
  {
          acquire lock
                  critical section
          release lock
                  remainder section
  }
  ```

4. Semaphores

- A **semaphore *S*** is an integer variable can be accessed only through two standard atomic operations: *wait()* and *signal()*.
- The definition of *wait()* is as follows:

```
wait(S) {
while (S<=0);
S--;
```

- The definition of *signal()* is as follows:

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

- When one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.

4.1 Usage

- The value of a **counting semaphore** can range over an unrestricted domain. The value of a **binary semaphore** can range only between 0 and 1. Same as Mutex Locks.
- Each process that wishes to use a resource performs a *wait()* operation on the semaphore. When a process releases a resource, it performs a *signal()* operation. When the count for the semaphore goes to 0, all resources are being used. After that, process that wish to use a resource will block until the count becomes greater than 0.
- We can also use semaphore to solve various synchronization problems.
- Consider two concurrently running processes: **P_1** with a statement **S_1** and **P_2** with a statement **S_2**. Suppose we require that **S_2** be executed only after **S_1** has completed.
- We can implement this scheme readily by letting **P_1** and **P_2** share a common semaphore **synch**, initialized to 0.
- In process **P_1**, we insert the statements

  S_1;
  signal(**synch**);
- In process **P_2**, we insert the statements

  *wait***(synch)**;
  S_2;
- Because **synch** is initialized to 0, **P_2** will execute **S_2** only after **P_1** has invoked signal(**synch),** which is after statement **S_1** has been executed.

4.2 Implementation

- To overcome the need for busy waiting, we can modify the definition of the *wait()* and *signal()* operations as follows: When a process execute the *wait()* operation and finds that the semaphore value is not positive, it must wait.
- However, rather than engaging is 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.
- To implement semaphores under this definition, we define a semaphore as follows:

```
typedef struct {
        int value;
```

```
            struct process *list;
    } semaphore ;
```

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

```
wait(semaphore *S) {
  S→value--;
  if(S→value<0){
          add this process to S→list;
          block();
          }
}
```

and the *signal()* semaphore operation can be defined as

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

- The *block()* operation suspends the process that invokes it. The *wakeup(**P**)* operation resumes the execution of a blocked process **P**.

4.3 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 is question is the execution of a ***signal()*** operation. When such a state is reached, these processes are said to be **deadlocked.**
- To illustrate this, consider a system consisting of two processes, P_0 and P_1, each accessing two semaphores, S and Q, set to the value 1:

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

- Suppose that P_0 executes ***wait(S)*** and then P_1 executes ***wait(Q).*** When P_0 executes ***wait(Q)***, it must wait until P_1 executes ***signal(Q).***
- Similarly, when P_1 executes ***wait(S)***, it must wait until P_0 executes ***signal(S)***. Since these signal() operations cannot be executed, Po and PI are deadlocked.
- We say that a set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set.
- Another problem related to deadlocks is indefinite blocking, or starvation/a situation in which processes wait indefinitely within the semaphore. Indefinite blocking may occur if we add and remove processes from the list associated with a semaphore in LIFO order.

5. Classic problems of Synchronization

5.1. The Bonded-Buffer Problem

- Producer puts information into the buffer, consumer takes it out. The problem arise when the producer wants to put a new item in the buffer, but it is already full. The solution is for the producer has to wait until the consumer has consumed atleast one buffer.
- Similarly, if the consumer wants to remove an item from the buffer and sees that the buffer is empty, it goes to sleep until the producer puts something in the buffer and wakes it up.
- Synchronization problems:
  1. We must guard against attempting to write data to the buffer when the buffer is full, i.e., the producer must wait for an 'empty space'.
  2. We must prevent the consumer from attempting to read data when the buffer is empty; i.e., the consumer must wait for 'data available'.
- To provide for each of these conditions, we require to employ three semaphores. The producer and consumer processes share the following data structure:

```
int n;
semaphore mutex=1;
semaphore empty=n;
semaphore full=0;
```

- 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 **n**; the semaphore full is initialized to **0**.
- The code for the producer process is shown below:

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

- The code for the consumer process is shown below:

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

5.2 The Readers-Writers Problem

- A database is to be shared among several concurrent processes. Some processes (readers) may want only to read the database, some (writers) may want to update the database.
- If two readers access the shared data simultaneously, no problem arises. However, if a writer and some other (either a reader or a writer) access the database simultaneously problem arises.
- To ensure that these difficulties do not arise, we require that the writers have exclusive

access to the shared database. This synchronization problem is referred to as the ***readers-writers problem***.

- The readers-writers problem has several variations, all involving priorities.
    1. No reader will be kept waiting unless a writer has already obtained permission to use the shared object.
    2. 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.
- In the solution to the first readers-writers problem, the reader processes share the following data structures:

```
        semaphore rw_mutex=1;
        semaphore mutex=1;
        int read_count=0;
```

- The semaphores ***mutex*** and ***rw_mutex*** are initialized to 1; **read_count** is initialized to 0.
- The **mutex** semaphore is used to ensure mutual exclusion when the variable **read_count** is updated. The **read_count** variable keeps track of how many processes are currently reading the object. The semaphore ***rw_mutex*** functions as a mutual-exclusion semaphore for the writers.
- The code for a writer process is shown below:

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

- The code for a reader process is shown below:

```
do {
        wait(mutex);
        read_count++;
        if (read_count ==1)
            wait(rw_mutex);
        signal(mutex);
            //reading is performed
        wait(mutex);
        read_count--;
        if(read_count==0)
            signal(w_mutex);
        signal(mutex);
        }while(TRUE);
```

- If a writer is in the critical section and ***n*** readers are waiting, then one reader is queued on ***rw_mutex*** and n-1 readers are queued on ***mutex.***

5.3 The Dining-Philosophers Problem

- Five philosophers are seated on 5 chairs across a table. Each philosopher has a plate full of noodles. Each philosopher needs a pair of forks to eat it. There are only 5 forks available all together. There is only one fork between any two plates of noodles.
- In order to eat, a philosopher lifts two forks, one to his left and the other to his right. If he is successful in obtaining two forks, he starts eating, after some time he stops eating and keeps both the forks down.

- What if all the 5 philosophers decide to eat at the same time?
    - All the 5 philosophers would attempt to pick up two forks at the same time. So, none of them succeed.
- One simple solution is to represent each fork with a semaphore. A philosopher tries to grab a fork by executing a wait() operation on that semaphore; he releases her forks by executing the signal() operation. Thus, the shared data are semaphore fork[5]; where all the elements of fork are initialized to 1.
- The structure of philosopher i is shown below.

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

- Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it could create a deadlock.
- Suppose all 5 philosophers become hungry simultaneously and each grabs his left fork, he will be delayed forever.
- Several possible remedies to the deadlock problem are listed.

1) Allow at most 4 philosophers to be sitting simultaneously at the table.
2) Allow a philosopher to pick up his fork only if both forks are available
3) An odd philosopher picks up first his left fork and then her right fork, whereas an even philosopher picks up his right fork and then his left fork.

6. Monitors

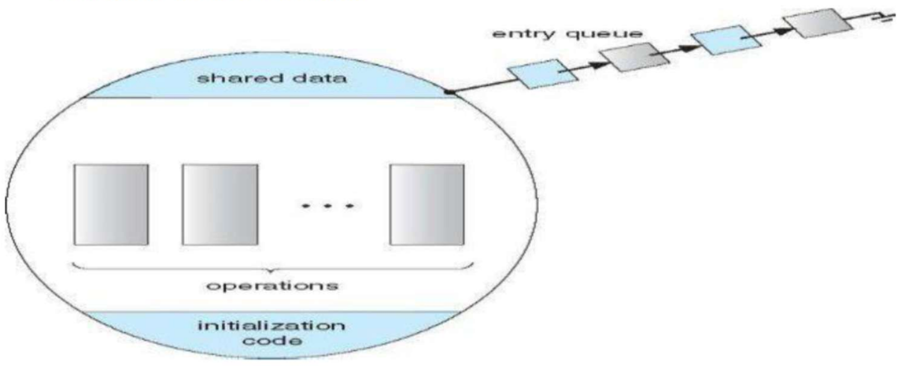
- Semaphore is unstructured construct. Wait and signal operations can be scattered in a program and hence debugging becomes difficult.
- Using semaphores incorrectly can result in timing errors that are difficult to detect, since these errors happen only if particular execution sequences take place and these sequences do not always occur.

6.1 Usage

- A monitor is an object that contains both the data and procedures needed to perform allocation of a shared resource. To accomplish resource allocation using monitors, a process must call a monitor entry routine.
- Many processes may want to enter the monitor at the same time, but only one process at a time is allowed to enter.
- Monitor data is accessible only within the monitor. There is no way for processes outside the monitor to access monitor data. This is a form of information hiding.
- If a process calls a monitor entry routine while no other processes are executing inside the monitor, the process acquires a lock on the monitor and enters it. While a process is in the monitor, other processes may not enter the monitor to acquire the resource.
- If a process calls a monitor entry routine while the other monitor is locked, the monitor makes the calling process wait outside the monitor until the lock on the monitor is released.

- The process that has the resource will call a monitor entry routine to release the resource. This routine could free the resource and wait for another requesting process to arrive. Monitor entry routine calls signal to allow one of the waiting processes to enter the monitor and acquire the resource.
- Monitor gives high priority to waiting processes than to newly arriving ones.

Schematic view of a Monitor



- The syntax of monitor is as follows:

```
Monitor monitor_name
{
/*Shared variable declarations*/
      Procedure body P1 (………) {
      . . . . . . . .
      }
      Procedure body P2 (………) {
      . . . . . . . .
      }
      .
      .
      Procedure body Pn (………) {
      . . . . . . . .
      }
      {
      Initialization Code
      }
}
```

- A programmer who needs to write a tailor-made synchronization scheme can define one or more variable of type **condition:**

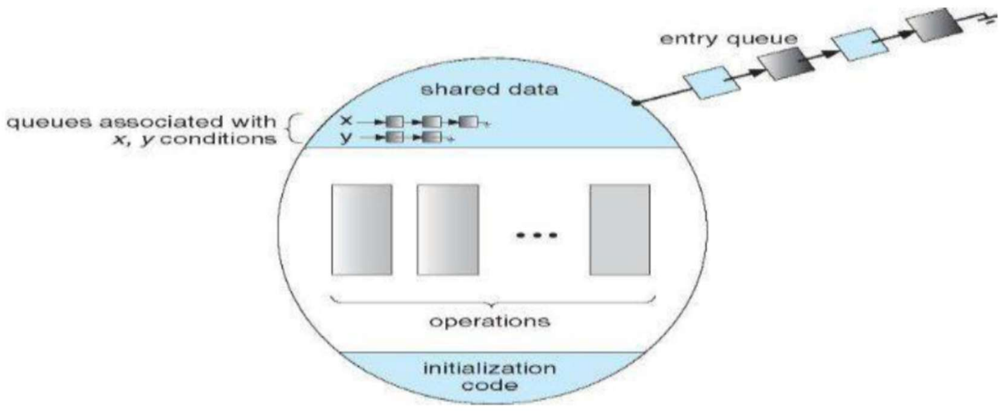
  condition x , y;
- The only operations that can be invoked on a condition variable are **wait()** and **signal().**
- The operation

  x.wait();

  means that the process invoking this operation is suspended until another process invokes

  x.signal();
- The **x.signal()** operation resumes exactly one suspended process. If no process is suspended, then the **signal()** operation has no effect, i.e., the state of **x** is the same as if the operation had never been executed.

Monitor with Condition Variables



- Suppose that when the **x.signal()** operation is invoked by a process **P,** there exists a suspended process **Q** associated with the condition **x.** Clearly, if the suspended process **Q** is allowed to resume its execution, the signaling process **P** must wait.
- Otherwise, both **P** and **Q** would be active simultaneously within the monitor.
- Every condition variable has an associated queue. A process calling wait on a particular condition variable is placed into the queue associated with the condition variable. A process calling signal on a particular condition variable causes a process waiting on that condition variable to be removed from the queue associated with it.
- Two possibilities exist:
  1) Signal and wait: **P** either waits until **Q** leaves the monitor or waits for another condition.
  2) Signal and continue: **Q** either waits until **P** leaves the monitor or waits for another condition.

6.2 Dining-Philosophers Solution using Monitors

- This solution imposes the restriction that a philosopher may pick up his forks only if both of them are available.
- To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:

  enum {THINKING, HUNGRY, EATING } state[5];
- Philosopher ***i*** can set the variable state[i] = Eating only if his two neighbors are not eating: (state[(i+4)%5]!=EATING) and (state[(i+1)%5]!=EATING).
- We also need to declare

  condition self[5];

  This allows philosophers ***i*** to delay himself when he is hungry but is unable to obtain the forks he needs
- A monitor solution to the dining-philosopher problem is as follows:

```
monitor DiningPhilosophers
{
   enum {THINKING, HUNGRY, EATING} state[5];
   condition self[5];
   void pickup(int i) {
           state[i] = HUNGRY;
           test(i);
```

```
            if (state[i]!=EATING)
                    self[i].wait();
            }
    void putdown(int i) {
            state[i]=THINKING;
            test((i+4)%5);
            test((i+1)%5);
            }
    void test(int i){
            if (state[i+4]%5]!=EATING) &&
              (state[i]==HUNGRY) &&
              (state[(i+1)%5]!=EATING)) {
                    state[i]=EATING;
                    self[i].signal();
                    }
            }
    initialization_code() {
            for(int i=0; i<5; i++)
                    state[i]=THINKING;
    }
 }
```

- The distribution of the forks is controlled by the monitor DiningPhilosophers. Each philosopher, before starting to eat, must invoke the operation **pickup()**. This act may result in the suspension of the philosopher process.
- After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the **putdown()** operation.

6.3 Implementing a Monitor Using Semaphores

- For each monitor, a semaphore **mutex(**initialized to 1**)** is provided. A process must execute **wait(mutex)** before entering the monitor and must execute **signal(mutex)** after leaving the monitor.
- Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore, **next,** is introduced, initialized to 0.
- The signaling processes can use **next** to suspend themselves. An integer variable **next_count** is also provided to count the number of processes suspended on **next**. Thus, each external function F is replaced by

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

Mutual exclusion within a monitor is ensured

- For each condition **x,** we introduce a semaphore **x_sem** and an integer variable **x_count,** both initialized to 0. The operation **x.wait()** can now be implemented as

```
                    x_count++;
                    if(next_count>0)
                            signal(next);
                    else
```

```
        signal(mutex);
    wait(x_sem);
    x_count--;
```

- The operation **x.signal()** can be implemented as

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

6.4 Resuming Processes Within a Monitor

- If several processes are suspended on condition variable **x** and **x.signal()** is executed, then how to determine which of the suspended processes should be resumed next ?
- One simplest solution is to use a First-Come, First-served (FCFS) ordering, so that the process that has been waiting the longest time is resumed first. Such a scheme is not adequate.
- Use the **conditional-wait** construct of the form

  x.wait(c)

  where **c** is an integer (called the priority number)

  The process with lowest number(highest priority) is resumed next.
- To illustrate this new mechanism, consider the ResouceAllocator monitor shown below Where R is an instance type of type ResourceAllocator

```
monitor ResourceAllocator
{
   boolean busy;
   condition x;
   void acquire(int time) {
            if (busy)
               x.wait(time);
            busy = true;
   }
   void release() {
            busy = false;
            x.signal();
   }
   initialization code() {
    busy = false;
   }
}
```

Single Resource Allocation

- Allocate a single resource among competing processes using priority numbers that specifies the maximum time a process plans to use the resource
- The process with the shortest time is allocated the resource first
- Let R is an instance of type ResourceAllocator

```
R.acquire(t);
   ...
  access the resurce;
   ...
R.release();
```

Dead Locks

1. System Model

- A system consists of a finite number of resources to be distributed among a number of competing threads.
- A thread must request a resource before using it and must release the resource after using it. A thread may request as many resources as it requires to carry out its designated task.
- Under the normal mode of operation, a thread may utilize a resource in only the following sequence:
  1. **Request**. The thread requests the resource. If the request cannot be granted immediately, then the requesting thread must wait until it can acquire the resource.
  2. **Use.** The thread can operate on the resource.
  3. **Release.** The thread releases the resource.
- A set of threads is in a deadlocked state when every thread in the set is waiting for an event that can be caused only by another thread in the set. The events with which we are mainly concerned here are resource acquisition and release.

2. Deadlock characterization:

In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting.

2.1. Necessary Conditions:

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

Mutual Exclusion: At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource.

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.

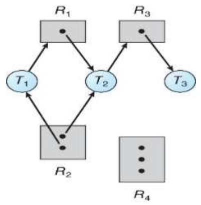
No preemption: Resources cannot be preempted; that is , a resource can be released only voluntarily by the process holding it, after that process has completed its task.

Circular wait: A set $\{P_0 ,P_1 ,\ldots P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource held by P_1 , P_1 is waiting for a resource held by P_2 …. P_{n-1} is waiting for a resource held by P_n.

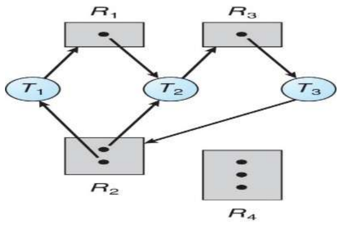
2.2. Resource-Allocation Graph

- 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 into two different types of nodes: $T = \{T_1, T_2, ..., T_n \}$, the set consisting of all the active threads in the system, and $R = \{R_1, R_2, ..., R_m \}$, the set consisting of all resource types in the system.
- A directed edge $T_i \rightarrow R_j$ is called a ***request edge***; it signifies that thread T_i has requested an instance of resource type R_j and is currently waiting for that resource.
- A directed edge $R_j \rightarrow T_i$ is called an ***assignment edge***; it signifies that an instance of resource type R_j has been allocated to thread T_i
- Pictorially, we represent each thread T_i as a circle and each resource type R_j as a rectangle. Each instance as a dot within the rectangle.
- The resource-allocation graph shown in Figure below depicts the following situation.
- The sets T, R, and E:
  i. $T = \{T_1, T_2, T_3 \}$
  ii. $R = \{R_1, R_2, R_3, R_4 \}$

iii. E = {$T_1 \rightarrow R_1$, $T_2 \rightarrow R_3$, $R_1 \rightarrow T_2$, $R_2 \rightarrow T_2$, $R_2 \rightarrow T_1$, $R_3 \rightarrow T_3$}



- Resource Instances:
  - i. One instance of R1
  - ii. Two instances of R2
  - iii. One instance of R3
  - iv. Three instance of R4
- Process/Thread States:
  - i. T1 holds one instance of R2 and is waiting for an instance of R1
  - ii. T2 holds one instance of R1, one instance of R2, and is waiting for an instance of R3
  - iii. T3 is holds one instance of R3
- Given the definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no thread 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.
- 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 thread T_3 requests an instance of resource type R_2. Since no resource instance is currently available, we add a request edge $T_3 \rightarrow R_2$ to the graph below.

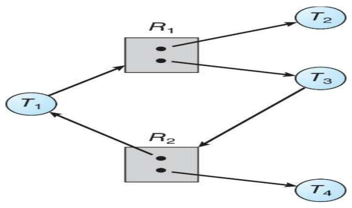


- At this point, two minimal cycles exist in the system:

$T_1 \rightarrow R_1 \rightarrow T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_1$
$T_2 \rightarrow R_3 \rightarrow T_3 \rightarrow R_2 \rightarrow T_2$

- Threads T_1, T_2, and T_3 are deadlocked. Thread T_2 is waiting for the resource R_3, which is held by thread T_3. Thread T_3 is waiting for either thread T_1 or thread T_2 to release resource R_2. In addition, thread T_1 is waiting for thread T_2 to release resource R_1.
- Now consider the resource-allocation graph in Figure below. In this example, we also have a cycle:

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



- However, there is no deadlock. Observe that thread T_4 may release its instance of resource type R_2. That resource can then be allocated to T_3, breaking the cycle.

3. Methods for handling Deadlocks

We can deal with the deadlock problem in one of three ways:

1) We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state.
2) We can allow the system to enter a deadlock state, detect it, and recover.
3) We can ignore the problem altogether and pretend that deadlocks never occur in the system.

- To ensure that deadlocks never occur, the system can use either a deadlock prevention or a deadlock-avoidance scheme.
- **Deadlock prevention** provides a set of methods to ensure that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made.
- **Deadlock avoidance** requires that the operating system be given additional information in advance concerning which resources a process will request and use during the lifetime. With additional knowledge, the operating system can decide for each request whether or not the process should wait.
- If a system does not employ either a deadlock-prevention or a deadlock-avoidance algorithm, then a deadlock situation may arise. In this environment the system can provide an algorithm that examines the state of the system to determine when a deadlock has occurred and an algorithm to recover from the deadlock.

4. Deadlock prevention: By ensuring that at least one of the necessary conditions cannot hold, we can prevent the occurrence of a deadlock.

4.1. Mutual Exclusion

- The mutual exclusion must hold. That is, at least one resource must be non-sharable.
- Sharable resources do not require mutually exclusive access and thus cannot be involved in a deadlock. Read-only files are a good example of a shared resource.
- In general, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable.

4.2. Hold and Wait

- We must guarantee that, whenever a process requests a resource, it does not hold any other resources. Require process to request and be allocated all its resources before it begins execution or allows a process to request resources only when it has none.
- Disadvantages
  - resource utilization may be low, since resources may be allocated but unused for a long period.
  - Starvation is possible. A process that needs several popular resources may have to wait indefinitely.

4.3. No preemption

- We must guarantee that, no preemption of resources that have already been allocated.
- If a process is holding some resources and requests another resource that cannot be immediately allocated to it, then all resources the process is currently holding are preempted.
- 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.

4.4. Circular Wait

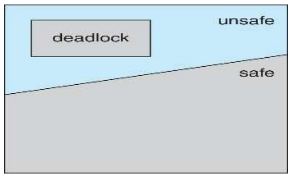
- One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.
- Let R={R_1, R_2,…. R_m,} be a set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering.
- 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 R_i
  - After that, the process can request instances of resource type R_j.. Alternatively, we can require that a process requesting an instance of resource type R_j must have released any resources R_i.

5. Deadlock avoidance

- Deadlock-prevention algorithms prevent deadlocks by limiting how requests can be made. The limits ensure that at least one of the necessary conditions for deadlock cannot occur.
- Possible side effects of preventing deadlocks by this method are low device utilization and reduced system throughput.
- Deadlock avoidance requires additional/prior information about how resources are to be requested.
- Various algorithms that use this approach differ in the amount and type of information required.
- The simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need.
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
- The resource-allocation ***state*** is defined by the number of available and allocated resources and the maximum demands of the process.

5.1. Safe State

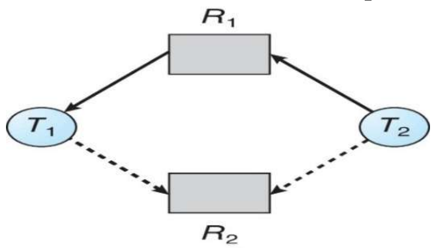
- A system is ***safe*** if the system can allocate resources to each process in some order and still avoid a deadlock.
- When a thread requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in ***safe state*** if there exists a sequence <T_1, T_2, …, T_n> of ALL the threads in the systems such that for each T_i, the resources that T_i can still request can be satisfied by currently available resources + resources held by all the T_j, with $j < i$.
- That is:
  - If T_i resource needs are not immediately available, then T_i can wait until all T_j have finished
  - When T_j is finished, T_i can obtain needed resources, execute, return allocated resources, and terminate
  - When T_i terminates, T_{i+1} can obtain its needed resources, and so on
- If a system is in safe state, then no deadlocks
- If a system is in unsafe state, then there is a possibility of deadlock



- 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.

5.2. Resource-Allocation-Graph Algorithm

- If we have a resource-allocation system with only one instance of each resource type, we can use a variant of the resource-allocation graph. In addition to the request and assignment edges, we introduce a new type of edge, called a **claim edge.**
- Claim edge: $T_i \rightarrow R_j$ indicated that process T_j may request resource R_j; represented by a dashed line.
- Claim edge converts to request edge when a thread requests a resource. Request edge converted to an assignment edge when the resource is allocated to the thread
- When a resource is released by a thread, assignment edge reconverts to a claim edge
  - Resources must be claimed a priori in the system
- Now suppose that process T_i requests resource R_j. The request can be granted only if converting the request edge $T_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow T_j$ does not result in the formation of a cycle in the resource-allocation graph.
- We can 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 T_i will have to wait for its requests to be satisfied.



Resource allocation graph for deadlock avoidance.

5.3. Banker's Algorithm

- Multiple instances of resources. Each thread must a priori claim maximum use.
- When a thread requests a resource, it may have to wait. When a thread gets all its resources it must return them in a finite amount of time
- Let ***n*** = number of processes, and ***m*** = no of resource types.

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 T_i may request at most k instances of resource type R_j

Allocation: A ***n x m*** matrix defines the number of resources of each type currently allocated to each process. If Allocation[i,j] = k then T_i is currently allocated k instances of R_j

Need: A ***n x m*** matrix indicates the remaining resource need of each process. If Need[i,j] = k, then T_i may need k more instances of Rj to complete its task Need [i,j] = Max[i,j] – Allocation [i,j]

Safety Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length m and n, respectively.
   Initialize:
   Work = Available
   Finish [i] = false for i = 0, 1, …, n- 1
2. Find an i such that both:
   (a) ***Finish [i] == false***
   (b) ***$Need_i$ ≤ 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.

Resource-Request Algorithm

$Request_i$ = request vector for process T_i. If **$Request_i$**[j] = k then process T_i wants k instances of resource type R_j

1. If ***$Request_i$ ≤ $Need_i$*** go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If ***$Request_i$ ≤ Available***, go to step 3. Otherwise ***T_i*** must wait, since resources are not available
3. Pretend to allocate requested resources to ***T_i*** by modifying the state as follows:
   Available = Available – $Request_i$;
   $Allocation_i$ = $Allocation_i$ + $Request_i$;
   $Need_i$ = $Need_i$ – $Request_i$;
   If safe ⇒ the resources are allocated to ***T_i***
   If unsafe ⇒ ***T_i*** must wait, and the old resource-allocation state is restored

Example:

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P0	1	1	2	4	3	3	2	1	0
P1	2	1	2	3	2	2			
P2	4	0	1	9	0	2			
P3	0	2	0	7	5	3			
P4	1	1	2	1	1	2			

- calculate the content of the need matrix?
- Check if the system is in a safe state?
- Determine the total sum of each type of resource?

1. The Content of the need matrix can be calculated by using the formula given below:

Need = Max – Allocation

Process	Need		
	A	B	C
P0	3	2	1
P1	1	1	0
P2	5	0	0
P3	7	3	3
P4	0	0	0

2. Let us now check for the safe state.

Safe sequence:

1. For process P0, Need = (3, 2, 1) and Available = (2, 1, 0)
Need <=Available = False
So, the system will move to the next process.
2. For Process P1, Need = (1, 1, 0) Available = (2, 1, 0)
Need <= Available = True
Request of P1 is granted.
Available = Available +Allocation
= (2, 1, 0) + (2, 1, 2) = (4, 2, 2) (New Available)
3. For Process P2, Need = (5, 0, 1) Available = (4, 2, 2)
Need <=Available = False
So, the system will move to the next process.
4. For Process P3, Need = (7, 3, 3) Available = (4, 2, 2)
Need <=Available = False
So, the system will move to the next process.
5. For Process P4, Need = (0, 0, 0) Available = (4, 2, 2)
Need <= Available = True
Request of P4 is granted.
Available = Available + Allocation
= (4, 2, 2) + (1, 1, 2) = (5, 3, 4) now, (New Available)
6. Now again check for Process P2, Need = (5, 0, 1) Available = (5, 3, 4)
Need <= Available = True
Request of P2 is granted.
Available = Available + Allocation
= (5, 3, 4) + (4, 0, 1) = (9, 3, 5) now, (New Available)
7. Now again check for Process P3, Need = (7, 3, 3) Available = (9, 3, 5)
Need <=Available = True
The request for P3 is granted.
Available = Available +Allocation
= (9, 3, 5) + (0, 2, 0) = (9, 5, 5)
8. Now again check for Process P0, = Need (3, 2, 1) = Available (9, 5, 5)
Need <= Available = True
So, the request will be granted to P0.
Safe sequence: < P1, P4, P2, P3, P0>
The system allocates all the needed resources to each process. So, we can say that the system is in a safe state.

3. The total amount of resources will be calculated by the following formula:

The total amount of resources= sum of columns of allocation + Available

= [8 5 7] + [2 1 0] = [10 6 7]

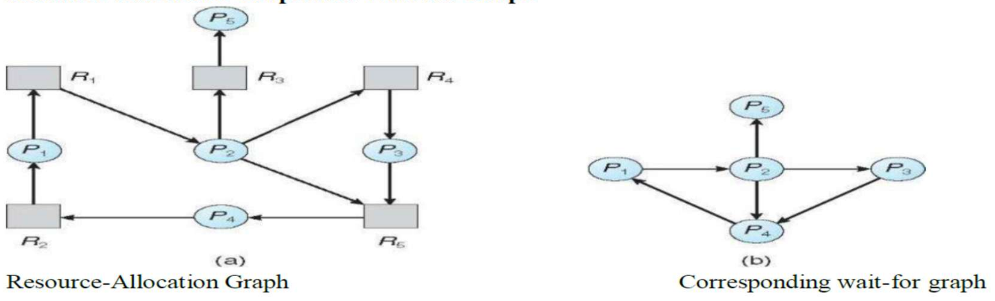
6. Deadlock detection

- If a system does not employ either a deadlock-prevention or a dead-lock avoidance algorithm, then a deadlock situation may occur.
- In this environment, the system may provide:
    - An algorithm that examines the state of the system to determine whether a deadlock has occurred.
    - An algorithm to recover from the deadlock

6.1. Single Instance of Each Resource Type

- If all resources have only a single instance, then we can define a deadlock-detection algorithm that uses a variant of the resource-allocation graph, called ***wait-for*** graph.
- We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.

Resource-Allocation Graph and Wait-for Graph



- An edge $P_i \rightarrow P_j$ in a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i needs. An edge $P_i \rightarrow P_j$ exists in a wait-for graph if and only if the corresponding resource-allocation graph contains two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ for some resources R_q
- 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.

6.2. Several Instances of a Resource Type

- The deadlock-detection algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm

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 reach process. If ***Request[i][j]*** equals ***k,*** then process P_i is requesting ***k*** more instances of resource type R_j.

Detection Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length ***m*** and ***n***, respectively. Initialize ***Work = Available***. For i = 0, 1, ..., n–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. $\boldsymbol{Request_i \leq Work}$

   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, $0 \le i < n$, then the system is in a deadlocked state. Moreover, if **Finish[i] == false**, then thread T_i is deadlocked.

- This algorithm requires an order of $\boldsymbol{m} \times \boldsymbol{n}^2$ operations to detect whether the system is in a deadlocked state.

6.3. Detection-Algorithm Usage

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

7. Recovery from Deadlock

- When a detection algorithm determines that a deadlock exists, several natives are available.
  1) Inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.
  2) Let the system recover from the deadlock automatically.
- There are two options for breaking a deadlock.
  1) Simply abort one or more processes to break the circular wait.
  2) Preempt some resources from one or more of the deadlocked processes

7.1. Process Termination:

- To eliminate deadlocks by aborting a process, we use one of two methods. In both methods, the system reclaim all resources allocated to the terminated processes.
  - Abort all deadlocked processes.
  - Abort one process at a time until the deadlock cycle is eliminated.
- Aborting a process is not be easy task. If partial termination method is used, then we must determine which deadlocked process should be terminated.
- In which order should we choose to abort?
  1) Priority of the thread
  2) How long has the thread computed, and how much longer to completion
  3) Resources that the thread has used
  4) Resources that the thread needs to complete
  5) How many threads will need to be terminated
  6) Is the thread interactive or batch?

7.2. 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
- If preemption is required to deal with deadlocks, then three issues need to be addressed:
  - **Selecting a victim** – We must determine the order of preemption to minimize cost
  - **Rollback** – We must rollback the process to some safe state and restart it from that state
  - **Starvation** – 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.