Deadlock

Every process needs some resources to complete its execution. However, the resource is granted in a sequential order.

1. The process requests for some resource.
2. OS grant the resource if it is available otherwise let the process waits.
3. The process uses it and release on the completion.

Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.

Consider an example when two trains are coming toward each other on same track and there is only one track, none of the trains can move once they are in front of each other. Similar situation occurs in operating systems when there are two or more processes hold some resources and wait for resources held by other(s). For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.

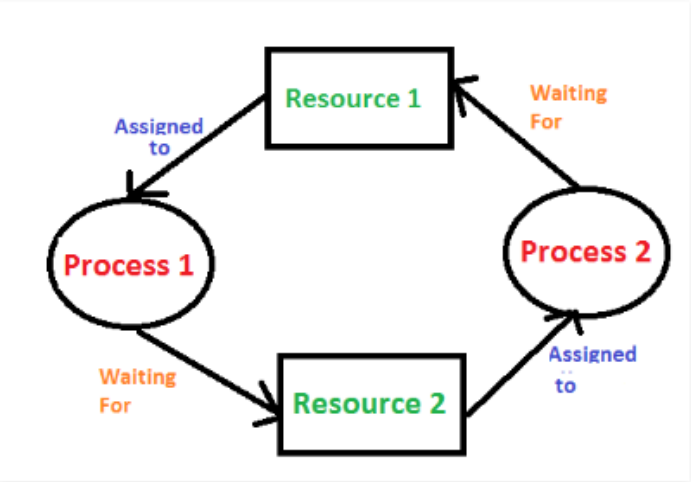


Fig: Deadlock

Necessary conditions for Deadlocks

1. **Mutual Exclusion**

   A resource can only be shared in mutually exclusive manner. It implies, if two process cannot use the same resource at the same time.

2. **Hold and Wait**

   A process waits for some resources while holding another resource at the same time.

3. **No preemption**

   The process which once scheduled will be executed till the completion. No other process can be scheduled by the scheduler meanwhile.

4. **Circular Wait**

   All the processes must be waiting for the resources in a cyclic manner so that the last process is waiting for the resource which is being held by the first process.

Preemptable and Non-preemptable Resources

The **resource** is the object granted to a process.

Preemptable and Non-preemptable Resources

- Resources come in two types
  1. **Preemptable**, meaning that the resource can be taken away from its current owner (and given back later). An example is memory.
  2. **Non-preemptable**, meaning that the resource cannot be taken away. An example is a printer, CD.
- Life history of a resource is a sequence of
  1. Request
  2. Allocate
  3. Use
  4. Release
- Processes make requests, use the resource, and release the resource. The allocate decisions are made by the system and there are various policies used to make these decisions.

Deadlock modeling

These four conditions can be modeled using resource allocation graph is a directed graph that is used for describing and reasoning about deadlock.

The graph's set of vertices V consists of two partitions: P = {P1, P2, P3, …, Pn} is the set of active processes, while R = {R1, R2, R3, …, Rm} is the set of available resource types.

The two kinds of vertices (nodes): **processes, shown as circles, and resources, shown as squares**.

The graph's set of edges E is also partitioned into two types: request edges (Pi→Rj) indicate that Pi is waiting for an instance of Rj, while assignment edges (Rj→Pi) indicate that Rj instance has been allocated to Pi.

An arc from a resource node (square) to a process node (circle) means that the resource has previously been requested by, granted to, and is currently held by that process. In fig, resource *R* is currently assigned to process *A* and resource S is currently assigned to B. An arc from a process node (circle) to a resource node (square) means that the process is currently blocked waiting for that resource.

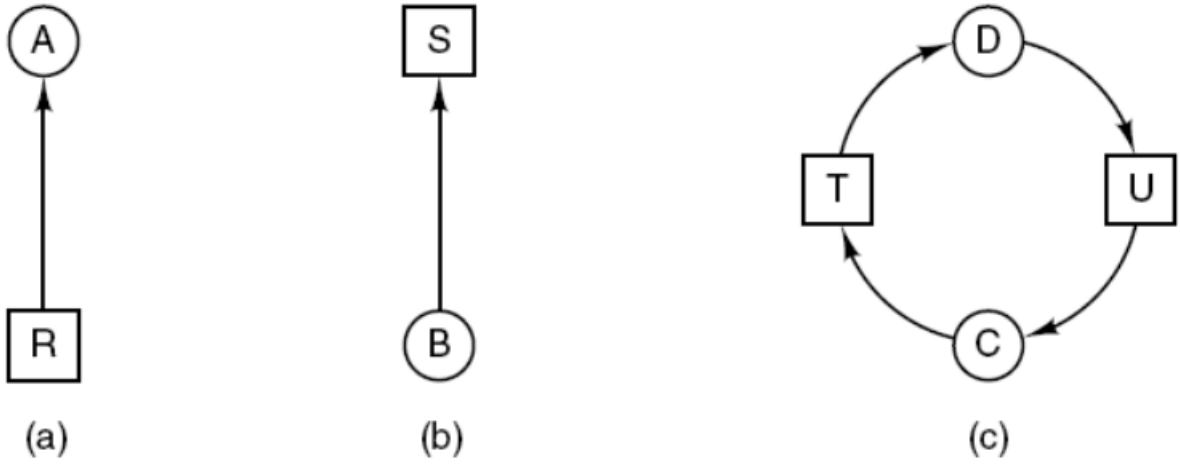


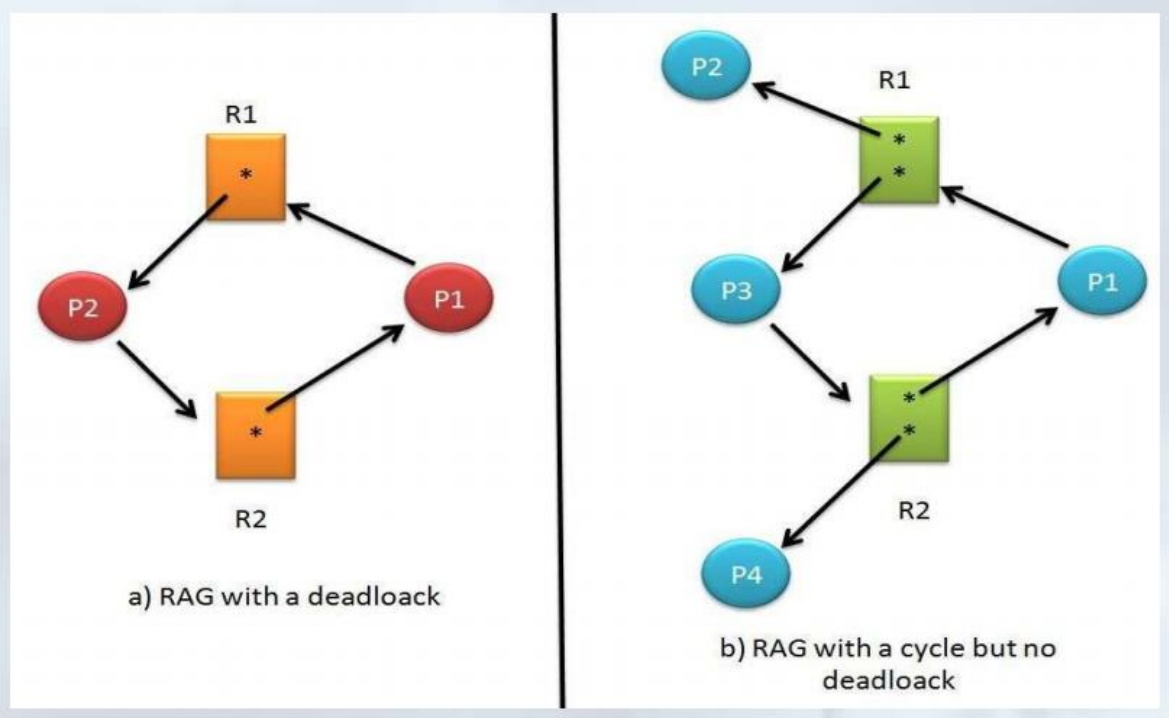
Fig:Resource allocation graph

By expressing some combination of processes, resource types, requests, and allocations as a resource-allocation graph, certain conclusions can be drawn based on the graph's structure:

• If the graph has no cycles, then there is no deadlock.

• If the graph does have a cycle, then there might be deadlock (i.e., the existence of a cycle is necessary but not sufficient):

- If the resource types involved in the cycle have only one instance (only one instance of that resource), then we have deadlock. In other words, the existence of a cycle becomes necessary and sufficient when we only have one instance per resource type in the cycle.
- If the resource types involved in the cycle have more than one instance (multiple instance of same resource), then there may and may not have deadlock.



There are 4 ways to overcome deadlock

- Deadlock avoidance
- Deadlock prevention
- Deadlock detection and recovery
- Deadlock ignorance

Deadlock Avoidance (Deadlock detection)

Banker's Algorithm

Banker's Algorithm is resource allocation and deadlock avoidance algorithm which test all the request made by processes for resources, it runs safety algorithm checks for the safe state, if after granting request system remains in the safe state it allows the request and if there is no

safe state it doesn't allow the request made by the process. Banker's algorithm ensures that the system will never enter in an unsafe state.

Inputs to Banker's Algorithm:

1. Max need of resources by each process.
2. Currently allocated resources by each process.
3. Max free available resources in the system.

The request will only be granted under the below condition:

1. If the request made by the process is less than equal to max need to that process.
2. If the request made by the process is less than equal to the freely available resource in the system.

Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Question1. What will be the content of the Need matrix?

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

So, the content of Need Matrix is:

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

Deadlock Prevention

We can prevent Deadlock by eliminating any of the above four conditions.

Eliminate Mutual Exclusion

It is not possible to dis-satisfy the mutual exclusion because some resources, such as the tape drive and printer, are inherently non-shareable.

Eliminate Hold and wait

1. Allocate all required resources to the process before the start of its execution, this way hold and wait condition is eliminated but it will lead to low device utilization. for example, if a process requires printer at a later time and we have allocated printer before the start of its execution printer will remain blocked till it has completed its execution.

Eliminate No Preemption

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.

Eliminate Circular Wait

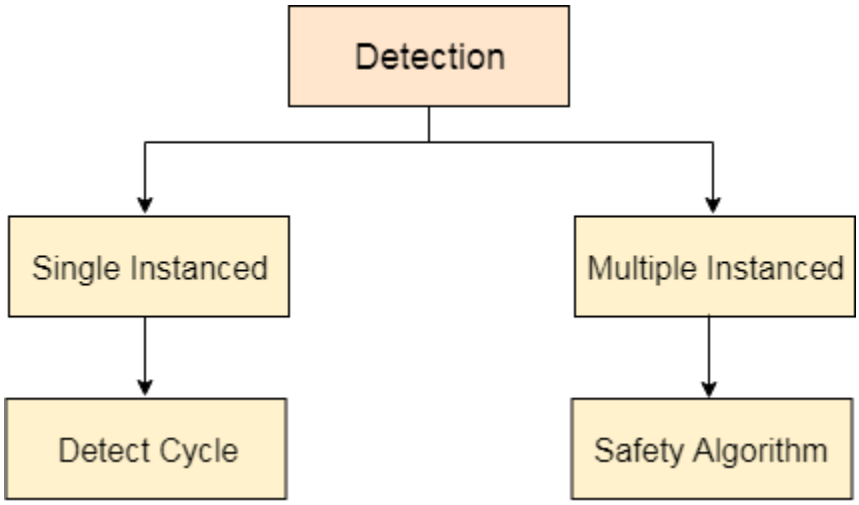
Each resource will be assigned with a numerical number. A process can request the resources increasing/decreasing order of numbering.
For Example, if P1 process is allocated R5 resources, now next time if P1 ask for R4, R3 lesser than R5 such request will not be granted, only request for resources more than R5 will be granted. The problem with this strategy is that it may be impossible to find an ordering that satisfies everyone.

Deadlock Detection and recovery

In this approach, The OS doesn't apply any mechanism to avoid or prevent the deadlocks. Therefore, the system considers that the deadlock will definitely occur. In order to get rid of deadlocks, The OS

periodically checks the system for any deadlock. In case, it finds any of the deadlock then the OS will recover the system using some recovery techniques.

The OS can detect the deadlocks with the help of Resource allocation graph.

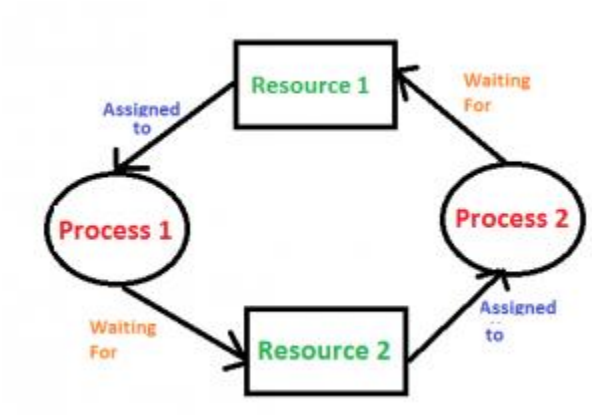


In single instanced resource types, if a cycle is being formed in the system then there will definitely be a deadlock. On the other hand, in multiple instanced resource type graph, detecting a cycle is not just enough. We have to apply the safety algorithm on the system by converting the resource allocation graph into the allocation matrix and request matrix.

Deadlock Detection

1. **If resources have single instance:**

   In this case for Deadlock detection we can run an algorithm to check for cycle in the **Resource Allocation Graph**. Presence of cycle in the graph is the sufficient condition for deadlock.



   In the above diagram, resource 1 and resource 2 have single instances. There is a cycle R1 → P1 → R2 → P2. So, Deadlock is Confirmed.

2. **If there are multiple instances of resources:**

   Detection of the cycle is necessary but not sufficient condition for deadlock detection. In this case, the system may or may not be in deadlock varies according to different situations.

   In a Resource Allocation Graph where all the resources are NOT single instance,

   - If a cycle is being formed, then system may be in a deadlock state.

- Banker's Algorithm is applied to confirm whether system is in a deadlock state or not.
- If no cycle is being formed, then system is not in a deadlock state. Presence of a cycle is a necessary but not a sufficient condition for the occurrence of deadlock.

Deadlock recovery

When a Deadlock Detection Algorithm determines that a deadlock has occurred in the system, the system must recover from that deadlock.

There are two approaches of breaking a Deadlock:

1. Process Termination:

To eliminate the deadlock, we can simply kill one or more processes.

For this, we use two methods:

(a) Abort all the Deadlocked Processes:

Aborting all the processes will certainly break the deadlock, but with a great expense. The deadlocked processes may have computed for a long time and the result of those partial computations must be discarded and there is a probability to recalculate them later.

(b) Abort one process at a time until deadlock is eliminated:

Abort one deadlocked process at a time, until deadlock cycle is eliminated from the system. Due to this method, there may be considerable overhead, because after aborting each process, we have to run deadlock detection algorithm to check whether any processes are still deadlocked.

2. Resource Preemption:

To eliminate deadlocks using resource preemption, we preempt some resources from processes and give those resources to other processes. This method will raise three issues –

(a) Selecting a victim:

We must determine which resources and which processes are to be preempted and also the order to minimize the cost.

(b) Rollback:

We must determine what should be done with the process from which resources are preempted. One simple idea is total rollback. That means abort the process and restart it.

(c) Starvation:

In a system, it may happen that same process is always picked as a victim. As a result, that process will never complete its designated task. This situation is called **Starvation** and must be avoided. One solution is that a process must be picked as a victim only a finite number of times.

Deadlock Ignorance (Ostrich Algorithm)

- Easiest way to deal with the problem
- Pretend that there is no problem
- This strategy says that stick your head into sand and pretend that there is no problem at all.

- This strategy suggests to ignore the deadlock because deadlock occur very rare, but the system crashes due to the hardware failure, compiler error, and operating system bugs frequently, then not to pay large penalty in the system performance or convince to eliminate deadlocks.

Issues related to deadlock

There is real danger of deadlock whenever multiple processes are running at the same instance of time.

The approach used is called two phase locking

In first phase, the process tries to lock all the records that it needs, one at a time. And if it succeeds, then it begins the second phase, performing its updates and releasing the locks. No any real work is done in the first phase.

If during the first phase, some records that is already locked, the process just releases all its locks and starts the first phase all over.

It is very difficult to implement in real situations.

Non-resource deadlocks:

Deadlock occurring without acquiring any resources by any processes. It is the situation when each process is waiting other process to do something.

For example: say you have two nodes in a network that communicate with each other and the following scenario occurs:

- Node1 sends a message to node 2 and waits for a response.

- Node2 receives the message and sends back the response to node1 and waits.
- But the response is lost on the network due to a temporary disruption.

  Both nodes are waiting for each other => Deadlock

Starvation

Starvation or indefinite blocking is phenomenon associated with the Priority scheduling algorithms, in which a process ready to run for CPU can wait indefinitely because of low priority. In heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU.

Some of the common causes of starvation are as follows:

1. If a process is never provided the resources it requires for execution because of faulty resource allocation decisions, then starvation can occur.
2. A lower priority process may wait forever if higher priority processes constantly monopolize the processor.
3. Starvation may occur if there are not enough resources to provide to every process as required.
4. If random selection of processes is used then a process may wait for a long time because of non-selection.

Some solutions that can be implemented in a system to handle starvation are as follows:

1. An independent manager can be used for allocation of resources. This resource manager distributes resources fairly and tries to avoid starvation.
2. Random selection of processes for resource allocation or processor allocation should be avoided as they encourage starvation.
3. The priority scheme of resource allocation should include concepts such as aging, where the priority of a process is increased the longer it waits. This avoids starvation.

There has been rumors that in 1967 Priority Scheduling was used in IBM 7094 at MIT, and they found a low-priority process that had not been submitted till 1973.

Assignment2: What is aging? Differentiate between deadlock and starvation.

What are Threads?

Thread is an execution unit which consists of**process id (Pid), its own program counter, a stack, and a set of registers.** Threads are also known as Lightweight processes.Threads have same properties as of the process so they are called as light weight processes. For example, in a browser, multiple tabs can be different threads. MS Word uses multiple threads: back ground thread may check spelling and grammar, while the foreground thread process inputs, while yetone thread to loads images from the hard drives, and a fourth does specific periodic automatic backup of the file being edited, etc.Threads are executed one after another but gives the illusion as if they are executing in parallel. Each thread has different states. Each thread has

1. Process ID
2. A program Counter
3. A register set
4. A stack space

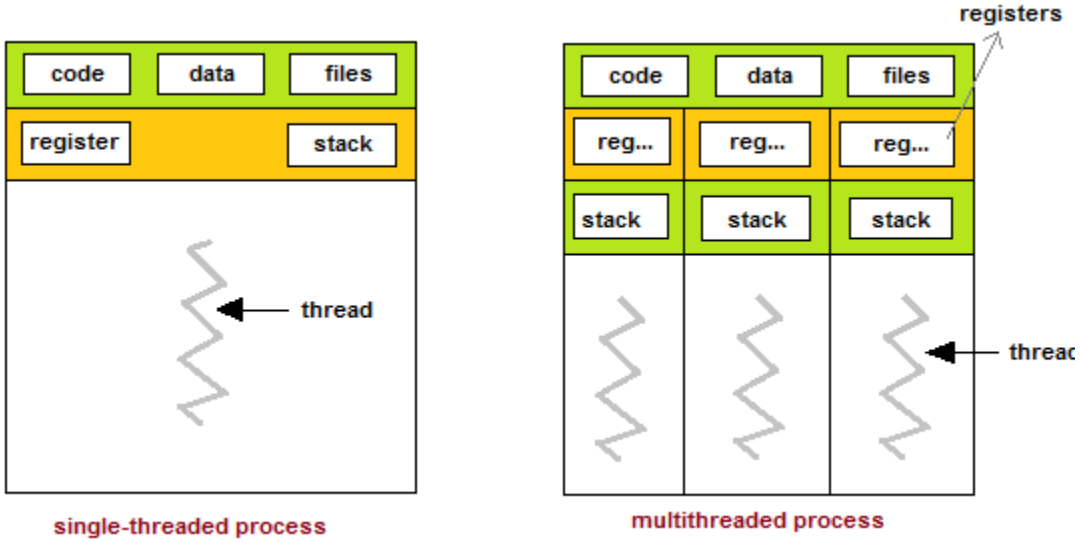


Fig: Threads

Advantages/Usages of Thread over Process

1. Responsiveness: If the process is divided into multiple threads, if one thread completes its execution, then its output can be immediately returned.

2. Faster context switch: Context switch time between threads is lower compared to process context switch. Process context switching requires more overhead from the CPU.

3. Effective utilization of multiprocessor system: If we have multiple threads in a single process, then we can schedule multiple threads on multiple processors. This will make process execution faster.

4. Resource sharing: Resources like code, data, and files can be shared among all threads within a process.

Note: stack and registers can't be shared among the threads. Each thread has its own stack and registers.

5. Communication: Communication between multiple threads is easier, as the threads shares common address space. while in process we have to follow some specific communication technique for communication between two processes.

6. Enhanced throughput of the system: If a process is divided into multiple threads, and each thread function is considered as one job, then the number of jobs completed per unit of time is increased, thusincreasing the throughput of the system.

Types of Threads

There are two types of threads

- ## User level (space) threads

  It is implemented in the user level library;and the kernel is not aware of the existence of these threads. They are not created using the system calls, the user-level threads are implemented by users. Thread switching does not need to call OS and to cause interrupt to Kernel. **Kernel doesn't know about the user level thread and manages them as if they were single-threaded processes.**User-level threads are small and much faster than kernel level threads. They are represented by **a program counter(PC), stack, registers and a small process control block**. Also, there is no kernel involvement in synchronization for user-level threads.

- ## Kernel level (space) threads

  Kernel-level threads are handled by the operating system directly and the thread management is done by the kernel. The context information for the process as well as the process threads is all managed by the kernel. Because of this, kernel-level threads are slower than user-level threads.

Multi-Threading Models

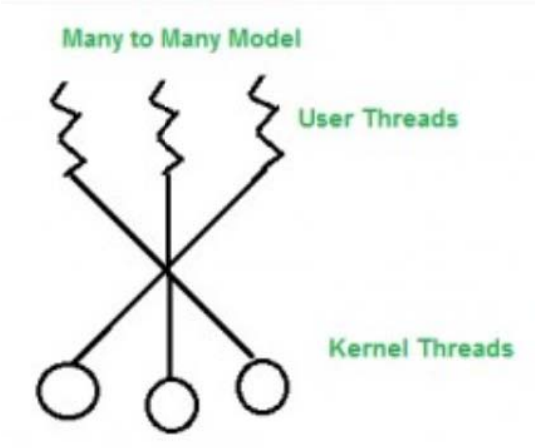
Many operating systems support kernel thread and user thread in a combined way. Example of such system is Solaris. Multi-threading model are of three types.

- Many to many
- Many to one
- One to one

1. Many to many

The many to many model maps many of the user threads to an equal number or lesser kernel threads. The number of kernel threads depends on the application or machine. There can be as many user threads as required and their corresponding kernel threads can run in parallel on a multiprocessor.

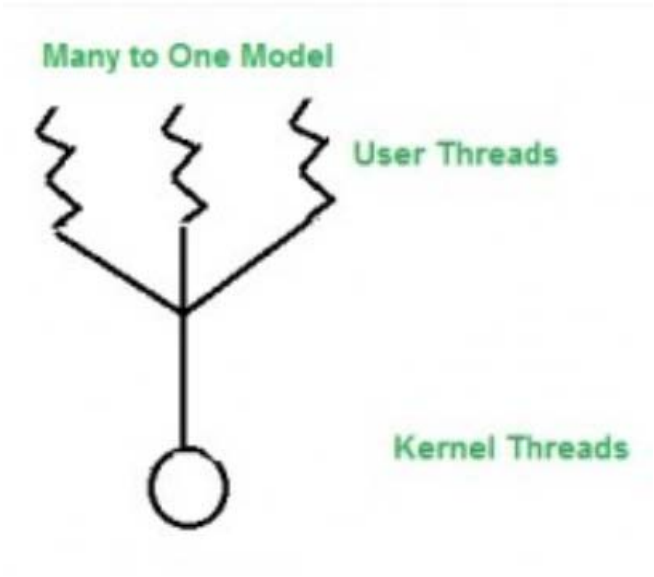
A diagram that demonstrates the “many to many” model is given as follows:



Many to many

2. Many to one

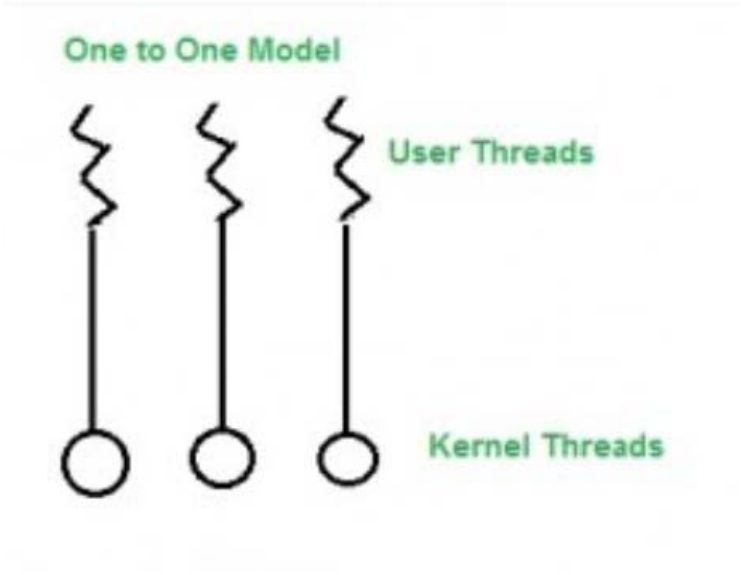
In this model, we have multiple user threads mapped to one kernel thread. In this model when a user thread makes a blocking system call entire process blocks. As we have only one kernel thread and only one user thread can access kernel at a time, so multiple threads are not able access multiprocessor at the same time.



Many to one

3. One to One

In this model, one to one relationship between kernel and user thread. In this model multiple thread can run on multiple processors. Problem with this model is that creating a user thread requires the corresponding kernel thread.



One-to-one model

Assignment3: Differentiate between thread and process.

The producer-consumer problem
(Bounded buffer problem)

The Producer-Consumer problem is a classic problem this is used for multi-process synchronization i.e. synchronization between more than one processes.

In the producer-consumer problem, there is one Producer that is producing something and there is one Consumer that is consuming the products produced by the Producer. The producers and consumers share the same memory buffer that is of fixed-size.

The job of the Producer is to generate the data, put it into the buffer, and again start generating data. While the job of the Consumer is to consume the data from the buffer.

What's the problem here?

The following are the solution to the problems that might occur in the Producer-Consumer:

- The producer should produce data only when the buffer is not full. If the buffer is full, then the producer shouldn't be allowed to put any data into the buffer.

- The consumer should consume data only when the buffer is not empty. If the buffer is empty, then the consumer shouldn't be allowed to take any data from the buffer.
- The producer and consumer should not access the buffer at the same time.

Solution:

The solution for the producer is to either go to sleep or discard data if the buffer is full. When the consumer removes an item from the buffer, it must notify the producer who start to fill the buffer again.

In the same way, consumer can go to sleep if it finds the buffer empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

The solution can be implemented using the IPC (inter-process communication).

pseudocode

```
int itemcount
procedure producer()
{
        while(true)
                {
                        item=produce item();
                                if(itemcount=buffer_size)    //buffer full
                                        sleep;
                                put_item_into_buffer(item);
                                itemcount++;
                                if(itemcount=1)
                                        wakeup(consumer);
                }
}

procedure consumer()
{
        while(true)
                {
                                if(itemcount=0)       //buffer empty
                                        sleep;
                                item=remove_item_from_buffer(item);
                                itemcount--;
                                if(itemcount=buffer_size -1)
                                        wakeup(producer);
                }
}
```

The producer produces the item and inserts it into the buffer. The value of the global variable count got increased at each insertion. If the buffer

is filled completely and no slot is available then the producer will sleep, otherwise it keeps inserting.

On the consumer's end, the value of count got decreased by 1 at each consumption. If the buffer is empty at any point of time then the consumer will sleep otherwise, it keeps consuming the items and decreasing the value of count by 1.

The consumer will be waked up by the producer if there is at least 1 item available in the buffer which is to be consumed. The producer will be waked up by the consumer if there is at least one slot available in the buffer so that the producer can write that.

Well, the problem arises in the case when the consumer got preempted just before it was about to sleep. Now the consumer is neither sleeping nor consuming. Since the producer is not aware of the fact that consumer is not actually sleeping therefore it keep waking the consumer while the consumer is not responding since it is not sleeping.

This leads to the wastage of system calls. When the consumer get scheduled again, it will sleep because it was about to sleep when it was preempted.

The producer keeps writing in the buffer and it got filled after some time. The producer will also sleep at that time keeping in the mind that

the consumer will wake him up when there is a slot available in the buffer.

The consumer is also sleeping and not aware with the fact that the producer will wake him up.

This is a kind of deadlock where neither producer nor consumer is active and waiting for each other to wake them up. This is a serious problem which needs to be addressed.

Semaphore

Semaphore is an integer variable that is used to solve the critical section problem in mutual exclusive manner by using two atomic operations, **wait** (down, P) and **signal** (up, V) that are used for process synchronization.

Types of Semaphores

There are two main types of semaphores i.e. counting semaphores and binary semaphores. Details about these are given as follows:

1. **Counting Semaphores**

   These are integer value semaphores and have an unrestricted value domain. These semaphores are used to coordinate the resource

access, where the semaphore count is the number of available resources. If the resources are added, semaphore count automatically incremented and if the resources are removed, the count is decremented.

2. **Binary Semaphores**

   The binary semaphores are like counting semaphores but their value is restricted to 0 and 1. The wait operation only works when the semaphore is 1 and the signal operation succeeds when semaphore is 0. It is sometimes easier to implement binary semaphores than counting semaphores.

The definitions of wait and signal are as follows:

1. **Wait**

   The wait operation decrements the value of its argument S, if it is positive. If S is zero, then no operation is performed.

```
wait(S)
{
   If (S==1);
      S--; // S=0
}
  else{
        block();
```

```
add this process to susp1end list.
sleep();}
```

2. **Signal**

   The signal operation increments the value of its argument S.

```
signal(S)
{
if (suspend list is empty)
   S++;
else
select a process from suspend list
wake up();
}
```

Producer consumer problem with semaphore:

The above three problems can be solved with the help of semaphores.

In the producer-consumer problem, we use three semaphore variables:

1. **Semaphore S:** This semaphore variable is used to achieve mutual exclusion between processes. By using this variable, either Producer or Consumer will be allowed to use or access the shared buffer at a particular time. This variable is set to 1 initially.
2. **Semaphore E:** This semaphore variable is used to define the empty space in the buffer. Initially, it is set to the whole space of the buffer i.e. "n" because the buffer is initially empty.

3. **Semaphore F:** This semaphore variable is used to define the space that is filled by the producer. Initially, it is set to "0" because there is no space filled by the producer initially.

By using the above three semaphore variables and by using the *wait()* and *signal()* function, we can solve our problem(the *wait()* function decreases the semaphore variable by 1 and the *signal()* function increases the semaphore variable by 1). So, let's see how:

The following is the pseudo-code for the producer:

```
#define N 100 //maximum slots in buffer
#define count=0 //items in the buffer
void producer()
{
int item;
   while(T) {
                item=produce_item();//generates item
                wait(E); // decrement empty slot
                wait(S); //set S to 0 i.e. enter into CS
                 append(); //add item to buffer
                signal(S); //set S to 1 i.e. informing CS is free
                signal(F); //increment full slot
   }
}
```

The above code can be summarized as:

- *while()* is used to produce data, again and again, if it wishes to produce, again and again.
- *produce()* function is called to produce data by the producer.
- *wait(E)* will reduce the value of the semaphore variable "E" by one i.e. when the producer produces something then there is a decrease in the value of the empty space in the buffer. If the buffer is full i.e. the value of the semaphore variable "E" is "0", then the program will stop its execution and no production will be done.
- *wait(S)* is used to set the semaphore variable "S" to "0" so that no other process can enter into the critical section.
- *append()* function is used to append the newly produced data in the buffer.
- *signal(s)* is used to set the semaphore variable "S" to "1" so that other processes can come into the critical section now because the production is done and the append operation is also done.
- *signal(F)* is used to increase the semaphore variable "F" by one because after adding the data into the buffer, one space is filled in the buffer and the variable "F" must be updated.

This is how we solve the produce part of the producer-consumer problem. Now, let's see the consumer solution. The following is the code for the consumer:

```
void consumer() {
        int item;
                while(T)
        {
                wait(F); // decrement full counter
                wait(S); //set S to 0 i.e. enter to CS
                take(item); // consume item
                signal(S); // set S to 1 i.e. informing CS is free
                signal(E); // increment empty counter
                use(); // make use of removed item if needed by any
process
        }
}
```

The above code can be summarized as:

- *while()* is used to consume data, again and again, if it wishes to consume, again and again.
- *wait(F)* is used to decrease the semaphore variable "F" by one because if some data is consumed by the consumer then the variable "F" must be decreased by one.

- *wait(S)* is used to set the semaphore variable "S" to "0" so that no other process can enter into the critical section.
- *take()* function is used to take the data from the buffer by the consumer.
- *signal(S)* is used to set the semaphore variable "S" to "1" so that other processes can come into the critical section now because the consumption is done and the take operation is also done.
- *signal(E)* is used to increase the semaphore variable "E" by one because after taking the data from the buffer, one space is freed from the buffer and the variable "E" must be increased.
- use () is a function that is used to use the data taken from the buffer by the process to do some operation.

Monitor

The monitor is one of the ways to achieve Process synchronization. The monitor is supported by programming languages to achieve mutual exclusion between processes. For example, Java Synchronized methods. Java provides wait() and notify() constructs.

1. It is the collection of **condition variables and procedures** combined together in a special kind of module or a package.

2. The processes running outside the monitor can’t access the internal variable of the monitor but can call procedures of the monitor.
3. Only one process at a time can execute code inside monitors.

Condition Variables:

Two different operations are performed on the condition variables of the monitor.

Wait.

signal.

let say we have 2 condition variables

condition x, y; // Declaring variable

Wait operation

x.wait() : Process performing wait operation on any condition variable are suspended. The suspended processes are placed in block queue of that condition variable.

Signal operation

x.signal(): When a process performs signal operation on condition variable, one of the blocked processes is given chance.

Solution to Producer consumer problem using monitor

monitor producer_consumer

condition full, empty;

```
int count;
procedure enter:
        begin;
        if count=N then wait(full);
        else enter_item;
        count++;
        if count=1 then signal(empty);
        end;
procedure remove:
        begin
        if count=0 then wait(empty); // do nothing
        else remove_item;
        count--;
        if count=N-1 then signal(full)
        end;
        count=0;
end
end monitor
procedure producer
 begin;
while(true)
        do
                begin
```

```
            produce_item
            producer_consumer.enter
        end
end
procedure consumer
begin
while(true)
        do
            begin
            producer_consumer.remove
            consume item;
        end
end
```

Limitations

- Monitors have to be implemented as part of the programming language. The compiler must generate code for them. This gives the compiler the additional burden of having to know what operating system facilities are available to control access to critical sections in concurrent processes. Some languages that do support monitors are Java,C#,Visual Basic,Ada and concurrent Euclid.

- Both semaphore and monitor are designed for solving the mutual exclusion problem in the one or more cpus' that all have access to common memory i.e. not suitable for system having private memory.

Producer consumer problem using message passing

```
#define N 100 //no of slots in the buffer
void producer (void)
{
int item;
message m; // message buffer
        while (true)
        {
                item = produce.item
                receive (consumer, &m) //wait for an empty slot to arrive
                build_message (&m, item) //construct a message to send
                send(consumer,&m) //send item to consumer
        }
}


void consumer (void)
{
int item, i;
message m; // message buffer
for (i=0; i<N; i++)
        send (producer, &m); //send N empty
```

```
        while (true)
        {
                receive (producer, &m); //get message containing item
                item= extract_item(&m); //extract item from message
                send (producer,&m); //send back empty reply
                consume_item(item); // do something with the item
        }
}
```