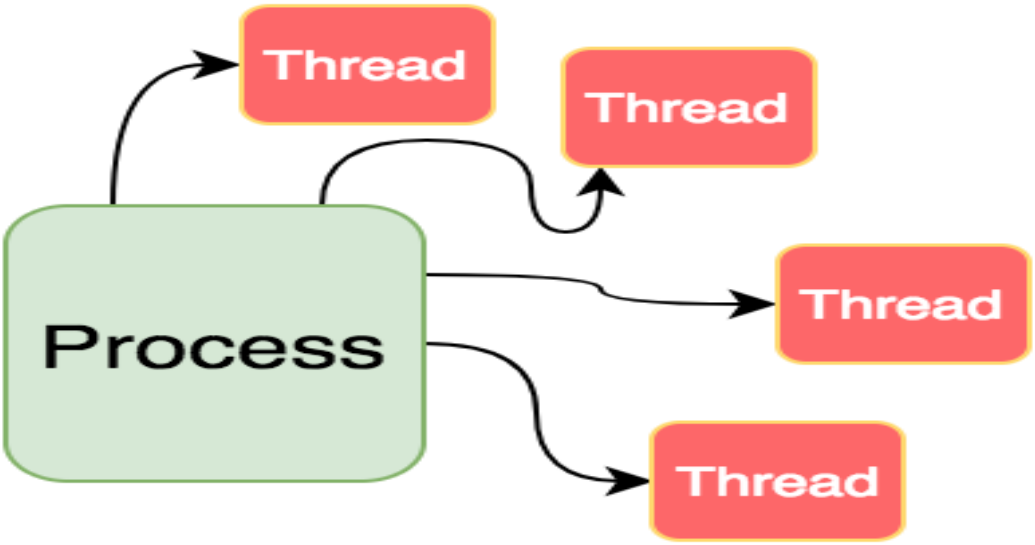
UNIT-2

A process is an active program i.e. a program that is under execution. It is more than the program code as it includes the program counter, process stack, registers, program code etc. Compared to this, the program code is only the text section.

A thread is a lightweight process that can be managed independently by a scheduler. It improves the application performance using parallelism. Thread shares information like data segment, code segment, files etc. with its peer threads while it contains its own registers, stack, counter etc.



The major differences between a process and a thread are given as follows:

Comparison Basis	Process	Thread
Definition	A process is a program under execution i.e an active program.	A thread is a lightweight process that can be managed independently by a scheduler.
Context switching time	Processes require more time for context switching as they are more heavy.	Threads require less time for context switching as they are lighter than processes.
Memory Sharing	Processes are totally independent and don't share memory.	A thread may share some memory with its peer threads.
Communication	Communication between processes requires more time than between threads.	Communication between threads requires less time than between processes .

www.jntufastupdates.com

Blocked	If a process gets blocked, remaining processes can continue execution.	If a user level thread gets blocked, all of its peer threads also get blocked.
Resource Consumption	Processes require more resources than threads.	Threads generally need less resources than processes.
Dependency	Individual processes are independent of each other.	Threads are parts of a process and so are dependent.
Data and Code sharing	Processes have independent data and code segments.	A thread shares the data segment, code segment, files etc. with its peer threads.
Treatment by OS	All the different processes are treated separately by the operating system.	All user level peer threads are treated as a single task by the operating system.
Time for creation	Processes require more time for creation.	Threads require less time for creation.

Inter process communication (IPC) is used for exchanging data between multiple threads in one or more processes or programs. The Processes may be running on single or multiple computers connected by a network. The full form of IPC is Inter-process communication.

It is a set of programming interface which allow a programmer to coordinate activities among various program processes which can run concurrently in an operating system. This allows a specific program to handle many user requests at the same time.

Since every single user request may result in multiple processes running in the operating system, the process may require to communicate with each other. Each IPC protocol approach has its own advantage and limitation, so it is not unusual for a single program to use all of the IPC methods.

Approaches for Inter-Process Communication

Here, are few important methods for interprocess communication:



Pipes

Pipe is widely used for communication between two related processes. This is a half-duplex method, so the first process communicates with the second process. However, in order to achieve a full-duplex, another pipe is needed.

Message Passing:

It is a mechanism for a process to communicate and synchronize. Using message passing, the process communicates with each other without resorting to shared variables.

IPC mechanism provides two operations:

- Send (message)- message size fixed or variable
- Received (message)

Message Queues:

A message queue is a linked list of messages stored within the kernel. It is identified by a message queue identifier. This method offers communication between single or multiple processes with full-duplex capacity.

Direct Communication:

In this type of inter-process communication process, should name each other explicitly. In this method, a link is established between one pair of communicating processes, and between each pair, only one link exists.

Indirect Communication:

Indirect communication establishes like only when processes share a common mailbox each pair of processes sharing several communication links. A link can communicate with many processes. The link may be bi-directional or unidirectional.

Shared Memory:

Shared memory is a memory shared between two or more processes that are established using shared memory between all the processes. This type of memory requires to protected from each other by synchronizing access across all the processes.

FIFO:

Communication between two unrelated processes. It is a full-duplex method, which means that the first process can communicate with the second process, and the opposite can also happen.

Why IPC?

Here, are the reasons for using the interprocess communication protocol for information sharing:

- It helps to speedup modularity
- Computational
- Privilege separation
- Convenience
- Helps operating system to communicate with each other and synchronize their actions.

A livelock is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing. Livelock is a special case of resource starvation; the general definition only states that a specific process is not progressing.

As a real-world example, livelock occurs when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they always both move the same way at the same time.

Livelock is a risk with some algorithms that detect and recover from deadlock. If more than one process takes action, the deadlock detection algorithm can repeatedly trigger. This can be avoided by ensuring that only one process (chosen randomly or by priority) takes action.

Livelock occurs when two or more processes continually repeat the same interaction in response to changes in the other processes without doing any useful work. These processes are not in the waiting state, and they are running concurrently. This is different from a deadlock because in a deadlock all processes are in the waiting state.

Example:
Imagine a pair of processes using two resources, as shown:

```
void process_A(void)
{
   enter_reg(& resource_1);
   enter_reg(& resource_2);
   use_both_resources();
   leave_reg(& resource_2);
   leave_reg(& resource_1);
}
void process_B(void)
{
   enter_reg(& resource_1);
   enter_reg(& resource_2);
   use_both_resources();
   leave_reg(& resource_2);
   leave_reg(& resource_1);
}
```

Each of the two processes needs the two resources and they use the polling primitive enter_reg to try to acquire the locks necessary for them. In case the attempt fails, the process just tries again.

If process A runs first and acquires resource 1 and then process B runs and acquires resource 2, no matter which one runs next, it will make no further progress, but neither of the two processes blocks. What actually happens is that it uses up its CPU quantum over and over again without any progress being made but also without any sort of blocking. Thus this situation is not that of a deadlock(as no process is being blocked) but we have something functionally equivalent to deadlock: LIVELOCK.

What leads to Livelocks?
Occurrence of livelocks can occur in the most surprising of ways. The total number of

allowed processes in some systems, is determined by the number of entries in the process table. Thus process table slots can be referred to as Finite Resources. If a fork fails because of the table being full, waiting a random time and trying again would be a reasonable approach for the program doing the fork.

Consider a UNIX system having 100 process slots. Ten programs are running, each of which having to create 12 (sub)processes. After each process has created 9 processes, the 10 original processes and the 90 new processes have exhausted the table. Each of the 10 original processes now sits in an endless loop forking and failing – which is aptly the situation of a deadlock. The probability of this happening is very little but it could happen.

Difference between Deadlock, Starvation, and Livelock:
A livelock is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing. Livelock is a special case of resource starvation; the general definition only states that a specific process is not progressing.

Livelock:
filter_none
brightness_4

```
var l1 = .... // lock object like semaphore or mutex
etc
var l2 = .... // lock object like semaphore or mutex
etc

      // Thread1
      Thread.Start( ()=> {

   while (true) {

      if (!l1.Lock(1000)) {
         continue;
      }

      if (!l2.Lock(1000)) {
         continue;
      }

      /// do some work
   });

      // Thread2
```

```
Thread.Start( ()=> {

while (true) {

   if (!l2.Lock(1000)) {
      continue;
   }

   if (!l1.Lock(1000)) {
      continue;
   }

   // do some work
});
```

Deadlock:

filter_none

brightness_4

```
var p = new object();
lock(p)
{
   lock(p)
   {
      // deadlock. Since p is previously locked
      // we will never reach here...
   }
```

A *deadlock* is a state in which each member of a group of actions, is waiting for some other member to release a lock. A *livelock* on the other hand is almost similar to a deadlock, except that the states of the processes involved in a livelock constantly keep on changing with regard to one another, none progressing. Thus Livelock is a special case of resource starvation, as stated from the general definition, the process is not progressing.

Starvation:

Starvation is a problem which is closely related to both, Livelock and Deadlock. In a dynamic system, requests for resources keep on happening. Thereby, some policy is needed to make a decision about who gets the resource when. This process, being reasonable, may lead to a some processes never getting serviced even though they are not deadlocked.

filter_none

brightness_4

```
Queue q = .....

      while (q.Count & gt; 0)
{
   var c = q.Dequeue();
   .....
```

```
        // Some method in different thread accidentally
        // puts c back in queue twice within same time
frame
        q.Enqueue(c);
    q.Enqueue(c);

    // leading to growth of queue twice then it
    // can consume, thus starving of computing
}
```

Starvation happens when “greedy” threads make shared resources unavailable for long periods. For instance, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

Deadlock Characteristics,

deadlock has following characteristics.

1. Mutual Exclusion
2. Hold and Wait
3. No preemption
4. Circular wait

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.
2. The process will make a new request for resources after releasing the current set of resources. This solution may lead to starvation.

Eliminate No Preemption
Preempt resources from the process when resources required by other high priority processes.

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.

Deadlock Avoidance
Deadlock avoidance can be done with Banker's Algorithm.

Banker's Algorithm
Bankers's Algorithm is resource allocation and deadlock avoidance algorithm which test all the request made by processes for resources, it 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.

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.

Example:

Total resources in system:

A B C D

6 5 7 6

Available system resources are:

A B C D

3 1 1 2

Processes (currently allocated resources):

A B C D

P1 1 2 2 1

P2 1 0 3 3

P3 1 2 1 0

Processes (maximum resources):

A B C D

P1 3 3 2 2

P2 1 2 3 4

P3 1 3 5 0

Need = maximum resources - currently allocated resources.

Processes (need resources):

A B C D

P1 2 1 0 1

P2 0 2 0 1

P3 0 1 4 0

Note: Deadlock prevention is more strict that Deadlock Avoidance.
Issues and challenges in concurrent programming paradigm

Violating mutual exclusion

Some operations in a concurrent program may fail to produce the desired effect if they are performed by two or more processes
simultaneously. The code that implements such operations constitutes a *critical region* or *critical section*. If one process is in a critical region, all other processes must be excluded until the first process has finished. When constructing any concurrent program, it is essential for software developers to recognize where such *mutual exclusion* is needed and to control it accordingly. Most discussions of the need for mutual exclusion use the example of two processes attempting to execute a statement of the form:

x := x + 1

Assuming that *x* has the value *12* initially, the implementation of the statement may result in each process taking a local copy of this value, adding one to it and both returning *13* to *x* (unlucky!). Mutual exclusion for individual memory references is usually implemented in hardware. Thus, if two processes attempt to write the values 3 and 4, respectively, to the same memory location, one access will always exclude the other in time leaving a value of 3 or 4 and not any other bit pattern.

Deadlock

A process is said to be in a state of *deadlock* if it is waiting for an event that will not occur. Deadlock usually involves several processes and may lead to the termination of the program. A deadlock can occur when processes communicate (e.g., two processes attempt to send

messages to each other simultaneously and synchronously) but is a problem more frequently associated with resource management. In this context there are four necessary conditions for a deadlock to exist [Coffman71]:
1. Processes must claim exclusive access to resources.
2. Processes must hold some resources while waiting for others (i.e., acquire resources in a piecemeal fashion).
3. Resources may not be removed from waiting processes (no preemption).
4. A circular chain of processes exists in which each process holds one or more resources required by the next process in the chain. Techniques for avoiding or recovering from deadlock rely on negating at least one of these conditions. One of the best documented (though largely impractical) techniques for avoiding deadlock is Dijkstra's Banker's Algorithm [Dijkstra68]. Dijkstra also posed what has become a classic illustrative example in this field, that of the
Dining Philosophers [Dijkstra71].

Indefinite postponement (or starvation or lockout)
A process is said to be *indefinitely postponed* if it is delayed awaiting an event that may not occur. This situation can arise when resource requests are administered using an algorithm that makes no allowance for the waiting time of the processes involved. Systematic techniques for avoiding the problem place competing processes in a priority order such that the longer a process waits the higher its priority becomes. Dealing with processes strictly in their delay order is a simpler solution that is applicable in many circumstances. See [Bustard88] for a discussion of these techniques.

Unfairness
It is generally (but not universally) believed that where competition exists among processes of equal status in a concurrent program, some attempt should be made to ensure that the processes concerned make even progress; that is, to ensure that there is no obvious *unfairness* when meeting the needs of those processes. Fairness in a concurrent system can be considered at both the design and system implementation levels. For the designer, it is simply a guideline to observe when developing a program; any neglect of fairness may lead to indefinite postponement, leaving the program incorrect.

For a system implementer it is again a guideline. Most concurrent programming languages do not address fairness. Instead, the issue is left in the hands of the compiler writers and the developers of the
run-time support software.

Generally, when the same choice of action is offered repeatedly in a concurrent program it must not be possible for any particular action to be ignored indefinitely. This is a weak condition for fairness. A stronger condition is that when an open choice of action is offered, any selection should be equally likely.

Busy waiting

Regardless of the environment in which a concurrent program is executed, it is rarely acceptable for any of its processes to execute a loop awaiting a change of program state. This is known as *busy waiting*. The state variables involved constitute a *spin lock*. It is not in itself an error but it wastes processor power, which in turn may lead to the violation of a performance requirement. Ideally, the execution of the process concerned should be suspended and continued only when the condition for it to make progress is satisfied.

Transient errors

In the presence of nondeterminism, faults in a concurrent program may appear as *transient errors*; that is, the error may or may not occur depending on the execution path taken in a particular activation of the SEI-CM-24 Concepts of Concurrent Programming 7 program. The cause of a transient error tends to be difficult to identify because the events that precede it are often not known precisely and the source of the error cannot, in general, be found by experimentation. Thus, one of the skills in designing any concurrent program is an ability to express it in a form that guarantees correct program behavior despite any uncertainty over the order in which some individual operations are performed. That is, there should be no part of the program whose correct behavior is *timedependent*.