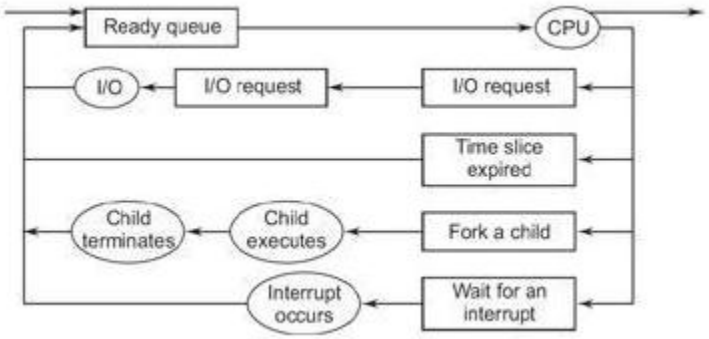
Scheduling and Threads

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The objective of time sharing is to switch the CPU among processes so frequently that it appears to the user that all programs are running concurrently. In uni-processor only one process is running.

Scheduling Queues

As process enter the system, they are put into a job queue. The figure below is showing processes in main memory are ready and waiting to execute. These are kept in ready queue. List of processes waiting for I/O device is called a device queue.



In the figure above rectangular box represents a queue such as ready and device queues. Circles represent resources that serve the queues. Arrows represent flow of processes.

Events

- Process issue an Input/Output request and are then placed in an Input/Output queue (I/O queue).
- Process executes only in the specified time interval, they are then shifted to the ready queue.
- Process creates a new sub-process and waits for its termination.
- Process removed forcibly from the CPU, as a result of an interrupt and then put back in the ready queue.

Schedulers
A process migrates between the various scheduling queues throughout its lifetime, such as longterm scheduling, medium-term scheduling and short-term scheduling.The details of the scheduling concepts will be discussed in the next article.

Co-Operating Processes

A process is independent when it can't be affect or be affected by the execution of another process. When the process shares data with other processes, it is called co-operating processes.A cooperating process can affect or be affected by the execution of another process.

Reasons for Co-operation

- Information sharing.For example: concurrent access to a shared file.
- Computation speedup. For example: parallel execution of sub-tasks.
- Modularity
- Convenience.For example: a user doing multi-tasking.
  But concurrent execution that requires cooperation among processes requires mechanism to allow process to communicate with each other and to synchronize their actions.
  One of the best example of such process is **Producer-Consumer Problem**.

Producer-Consumer Problem

Producer process produces information and consumer process consumes information, by using buffer. That is buffer can be filled by the producer and emptied by the consumer. Consumer must wait until an item is to be produced.Synchronization is necessary so that consumer cannot consume an item which is not produced.Similarly producer should also wait if the buffer is full,

Conditions

- Producer should wait if the buffer is full and
- Consumer should wait if the buffer is empty
  For example, print program produces information for printing and printer driver can consume it by printing.

Coding part

- Variables ‘in’ and ‘out’ initialized to zero (0)
- ‘In’ points to the next free position in the buffer.
- ‘Out’ points to the first full position in the buffer.
- Buffer is empty when in = out
- Buffer is full when in + 1 mod n = out
- noop is a no operation instruction i.e., do nothing
- while <condition> do noop test repetitively until it reaches to false.
- nextp points the location in the buffer where to place the produced item.
- nextc points the location in the buffer which item is to consume.

Producer Process

Code:

```
repeat
        ...
        produce an item in next p
        ...
        while in + 1 mod n = out do noop;
        buffer[in]: = next p;
        in: = in + 1 mod n;
until false;
```

Consumer process

Code:

```
repeat
        while in = out do noop;
        next c: = buffer[out];
        out: = out + 1 mod n;
        ...
        Consume the item in next c
        ...
Until false;
```

Inter Process Communication (IPC)

Inter process communication facility in operating system provide the means for co-operating processes to communicate and to synchronize their actions. That is to communicate with each other via IPC. Inter process communication is best provided by a message system. Many tasks can be accomplished in isolated processes, but many other tasks require Inter Process Communication.

Basic Structure

The function of a message system is to allow processes to communicate with each other. Inter process communication provides two operations: send and receive message. Messages can be of either fixed or variable size.

Communication link properties to implement

- A link is established between a pair of processes only if they have a shared mail box.
- Link may be associated with more than two processes.
- Between each pair of communicating processes, there may be a number of different links, each link corresponding to one mail box.
- A link may be either unidirectional or bidirectional.
- To know the capacity of a link.
- To know the size of messages.

Several methods for logically implementing a link

- Direct or Indirect communication.
- Symmetric or asymmetric communication.
- Automatic or explicit buffering.
- Send by copy or send by reference.
- Fixed-sized or variable-sized messages.

Naming

In direct communication, name the recipient or sender of the communication, by using send and receive primitives.
Send and receive primitives are:

- send (P, message): Send a message to process ‘P’.
- receive (Q, message): Receive a message from process ‘Q’.

E.g., Processes P1, P2, P3 share a mailbox A. P1 sends a message to A, while P2 and P3 execute a receive from A.

You may come up with a very common question

Which process will receive the message sent by P1?

Solution to this question is:

- Allow a link to be associated with at most two processes.
- One process at a time to execute a receive operation.
- System may identify the concerned receiver to the sender.

If each mailbox has a unique owner, there can be no confusion about who should receive a message sent to this mailbox.

Operating system provides a mechanism that allows a process

- To create a new mailbox
- To send and receive messages through mailbox
- To destroy a mailbox.

Buffering

A link can determine the number of messages that can reside in it temporarily.

Messages queue can be implemented in three different ways:

- **Zero Capacity**
  The queue has maximum length zero. In this method sender must wait until the recipient receives the message. Here two processes must be synchronized is called rendezvous.
- **Bounded Capacity**
  The bounded capacity queue has finite length ‘n’, thus at most ‘n’ messages can reside in it.
- **Unbounded Capacity**
  The unbounded capacity queue has potentially infinite length; thus, any number of messages can wait in it. The process of sending a message is delayed until it receives a reply (i.e., acknowledgement). This scheme was implemented in thoth operating system. In this system, messages are of fixed size (8 words each).

Exception Conditions

Message system is very much useful in a distributed environment, where processes may reside at different sites i.e., machines. In such cases there is a probability that an error will occur during communication and processing. Whenever a failure is to occur in either a centralized or distributed system, some error recovery, that is exception-condition handling must take place.

Process Terminates

In this process, either a sender or a receiver may terminate before a message is processed.

Lost Messages

Messages may be lost due to hardware or communication line failure. Methods for dealing this situation are:

- Operating system is responsible for detecting this event and for retransmitting.
- Sending process is responsible for detecting this event and for retransmitting.
  Use timeouts method for determining the message is lost.

Scrambled Messages
Message may be delivered to its destination, but scrambled on the way due to noise in the communication channel. For detecting such errors use error checking codes like checksums, parity, CRC, etc.

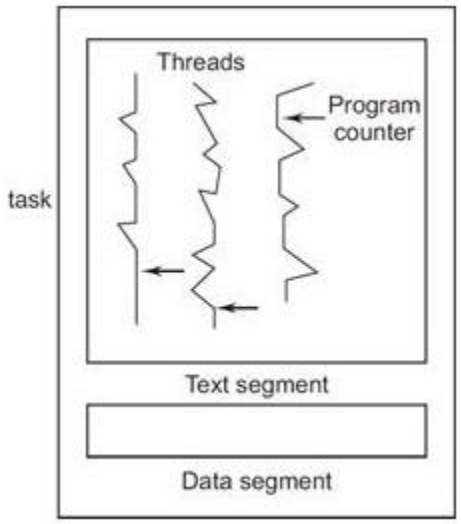
Threads

Thread is also called as Lightweight process (LWP). Thread is a basic unit of CPU utilization and consists of :

- a program counter that keeps track which instruction to execute next,
- a register set which holds its current working variables
- and a stack space which contains the execution history.
  Threads allow multiple execution to take place in the same process environment ta a large degree independent of one another.
  A thread share with its peer threads the code section,data section and resources like open files.
  A traditional or heavy weight process is equal to a task with one thread. A task does not do anything if no threads are in it, and a thread must be in exactly one task.Threads operate, in many respects same as processes. Thread states are ready, waiting, running and terminated. Like processes, threads share CPU and only

one thread at a time is active. A thread within a process executes sequentially, and each thread has its own stack and program counter.
Having multiple threads running in parallel in one process is analogous as to having multiple processes running in parallel in one computer.

Producer and consumer can be threads in a task. In multiprocessor system, producer and consumer threads can execute in parallel. Threads can be supported by the operating system kernel. This is shown in the figure below:



E.g., consider two processes, one with one thread (process ‘a’) and the other with 50 threads (process ‘b’). In general process ‘a’ and process ‘b’ receives the same number of time slices, so the thread in process ‘a’ runs 50 times as fast as a thread in process ‘b’. If each thread is scheduled independently, then process ‘b’ can receive 50 times the CPU time that process ‘a’ receives.

Why Threads?

Lets take an example. In modern GUI user can select an action to be performed at almost anytime i.e. an event can occur at any time. The program must be able to respond to a variety of different events at unknown times and in an unknown order of request.
Such user driven events are usually too small to justify the creation of new process.Instead the action is implemented as a thread. Thread can be executed independently without the overhead of a process.The primary requirement for thread is an area to store the program counter and registers associated with that thread.

Multithreading

It describes a situation of allowing multiple threads executing in the sane process.Modern Operating Systems supprt multiple threads of execution within a single process.

MS DOS supports single thread model.
Unix supports multiple User but supports only one thread per process.
Windows 2000,Solaris,Linux support multple threads.

Advantages Of Threads

- Multiple processes can perform the same task, thus it is more efficient to have one process containing multiple threads serve the same purpose.
- Higher throughput. Due to this, a single user can own an individual task with multiple threads.
- Improved performance. That is while one server thread is blocked or waiting, a second thread in the same task will run.
- Some systems implement user-level threads in user-level libraries, rather than via system calls, so thread swithing does not need to call the operating system, and to cause an interrupt to the kernel.

Disadvantages of Threads

- There is no protection between threads.
- If the kernel is single-threaded, then any user-level thread executing a system call will cause the entire task to wait until the system call returns.

Source: http://www.go4expert.com/articles/scheduling-and-threads-t22306/