**Thread**

# Threads

The process and its associated threads are different concept.

Processes are used to group resources together and threads are the entities scheduled for execution on the CPU.

A thread is a single sequence stream within in a process. Because threads have some of the properties of processes, they are sometimes called lightweight processes.

Like a traditional process i.e., process with one thread, a thread can be in any of several states (Running, Blocked, Ready or Terminated).

Each thread has its own stack. Since thread will generally call different procedures and thus a different execution history. This is why thread needs its own stack.

**Threads share with other threads:**

* Code section
* Data section
* OS resources like open files and signals
* Process state
* Process id
* Address space
* Heap
* Static Data
* File locks

**A thread does not share:**

* Program counter (PC)
* Register set
* Stack pointer

# Processes Vs Threads

## Similarities

Like processes, threads share CPU and only one thread active (running) at a time.

Like processes, threads within a processes execute sequentially.

Like processes, thread can create children.

Like process, if one thread is blocked, another thread can run.

## Differences

Unlike processes, threads are not independent of one another.

Unlike processes, all threads can access every address in the task .

Unlike processes, thread are design to assist one other. Note that processes might or might not assist one another because processes may originate from different users.

# Why Threads?

1. A process with multiple threads make a great server for example printer server.
2. Because threads can share common data, they do not need to use interprocess communication.
3. Because of the very nature, threads can take advantage of multiprocessors.

Threads are cheap in the sense that

1. They only need a stack and storage for registers therefore, threads are cheap to create.
2. Threads use very little resources of an operating system in which they are working. That is, threads do not need new address space, global data, program code or operating system resources.
3. Context switching are fast when working with threads. The reason is that we only have to save and/or restore PC, SP and registers.

But this cheapness does not come free - the biggest drawback is that there is no protection between threads.

# Threads over Multiple Processes

## Advantages

**Context Switching:** Threads are very inexpensive to create and destroy, and they are inexpensive to represent. For example, they require space to store, the PC, the SP, and the general-purpose registers, but they do not require space to share memory information, Information about open files of I/O devices in use, etc. With so little context, it is much faster to switch between threads. In other words, it is relatively easier for a context switch using threads.

**Sharing:** Treads allow the sharing of a lot resources that cannot be shared in process, for example, sharing code section, data section, Operating System resources like open file etc.

## Disadvantage

**Blocking:** The major disadvantage if that if the kernel is single threaded, a system call of one thread will block the whole process and CPU may be idle during the blocking period.

**Security:** Since there is, an extensive sharing among threads there is a potential problem of security. It is quite possible that one thread over writes the stack of another thread (or damaged shared data) although it is very unlikely since threads are meant to cooperate on a single task.

# END