Threads

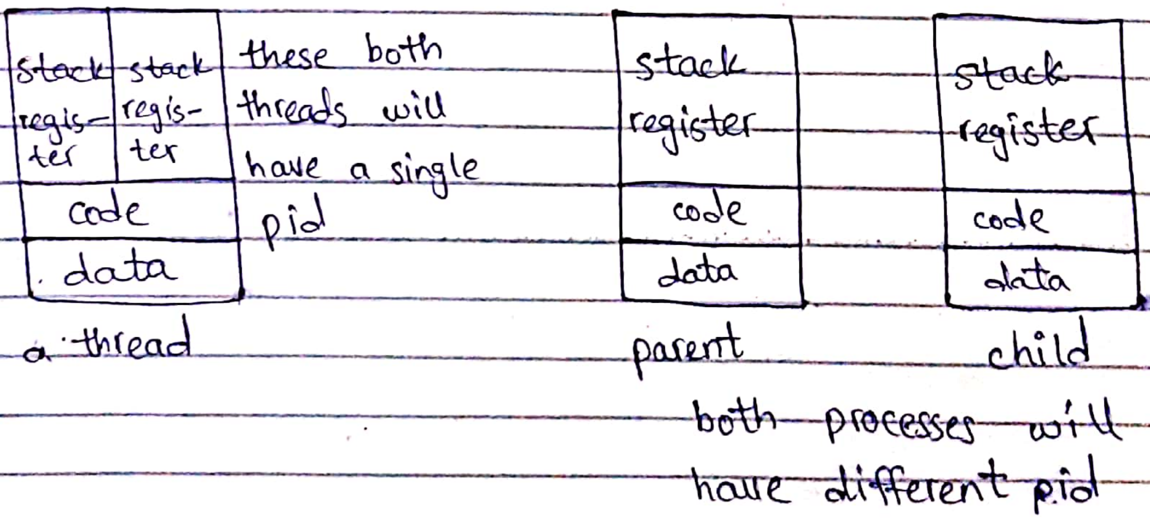
=> When we need to perform multitasking i.e. to perform multiple processes or multiple tasks at the same time, we can either do that with the help of multiple CPUs or with the using multiple threading.

=> A thread is just like a process but there are some differences b/w them.

=> There are system calls involved in a process for e.g. fork(), pipe() etc. but in threads there are no such system calls involved.

=> In a process, operating system treats each task differently. For instance, when fork [illegible] is called then the pid (process id) for the parent process and the child process will be different. But in threads, a thread is treated as a single task.

Below is an illustration of the above point :-



Different

=> A processes will have different copies of data, files and code while threads share same copy of code and data.

=> In processes context switching is slower as it takes a lot of time to move from one process to another but in threads they share same copy of code and data.

=> In processes, blocking one task will not block another task as they are independent while in threads, if we block a single thread then it will block the entire process as they are inter-dependent. For example, if ~~as~~ parent process is blocked due to a need of I/O then child process will work but in a thread if it is blocked then entire process is blocked.

Price Rs 140

Introduction to threads

=) A thread is a basic unit of CPU utilization. It is also called a lightweight process. It is a sequence of instructions within a process.

=) A thread behaves like a process within a process but it does not have its own Process controll block (PCB).

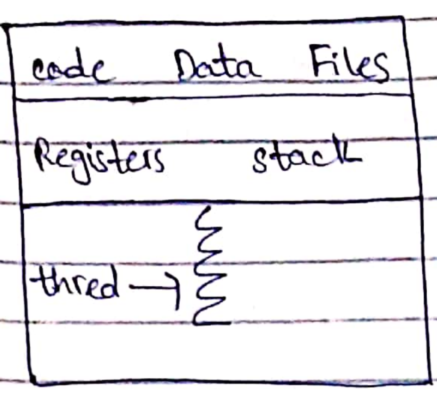
=) A thread comprises of a thread ID, a program counter, a register set and a stack.

=) Usually multiple threads are created within a process and multiple threads in a process allow multiple executions. A thread shares with other threads belonging to the same process its code section, data section and other operating-system resources, such as open files and signals.

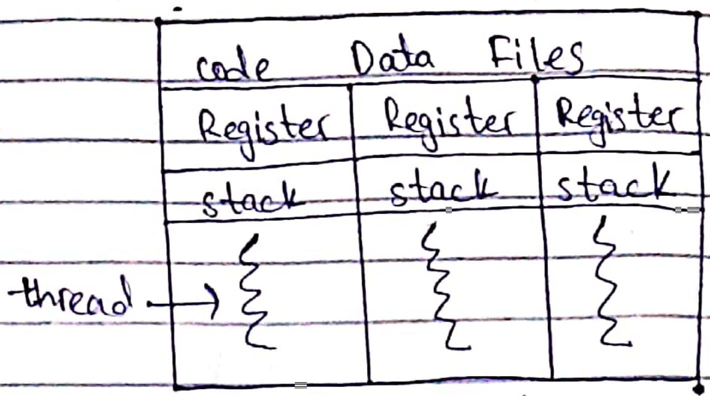
=) A traditional | heavyweight process has a single thread of control but if a process has multiple threads of control, it can perform more than one task at a time.

=) If a process has more than single thread Then each ~~process~~ thread will be assigned a different task. Hence, the process can perform more than one task at a time.

Below are two diagrams for better understanding:



single threaded process



multi-threaded process

this ~~one~~ process has only a single thread which means it can only perform one task at a time

this process has multiple threads so it can perform multiple tasks at a time

Benefits of multithreaded programming:

⇒ are broken down into four major categories:

① Responsiveness:

Multithreading increases responsiveness to the user. A process consists of more than one thread. If one thread is blocked or busy in a lengthy calculation, some other thread may still be executing. So the user gets more response from the executing process.

② Resource sharing:

All threads of one process share the memory and resources of that process. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

③ <u>Economy</u>:
Allocation of memory and resources for process creation is costly. All threads in a process share the resources of that process so it is more economical to create and context switch the threads.

④ <u>Utilization of multiprocessor architectures</u>:
Multiprocessor architecture allows the facility of parallel processing. It is the most efficient way of processing. A single threaded process can be executed on one CPU even if there are more processors. Multithreading on a multiprocessor system increases concurrency. Different parts of a multi-threaded process can be executed simultaneously on different processors.

Multi-threading Models

⇒ There are two types of threads: User threads and kernel threads.

User threads:

User level threads are implemented in user level libraries instead of system calls. The thread switching does not need to call operating system. It does not call interrupt to the kernel. The kernel knows nothing about user-level threads. It manages these threads as single-threaded processes. The user level threads are very fast.

Kernel threads:

Kernel level threads are supported within the kernel of the operating system. They allow the kernel to perform multiple tasks and to service multiple kernel system calls simultaneously.

Threads in coding

```
#include <pthread.h>

void* routine ()
{
    cout << " Hello! " << endl;
    sleep(3);
    cout << " end" << endl;
}
```

this function is a pointer type because in thread creation we have to give the address of the pointer function

sleep(3); → it will sleep for 3 seconds

```
int main (int argc, char* argv[])
{
    pthread_t t1;
```

this is a variable but the type is pthread_t

In order to start creating a thread we first have to define a sort of place where the application can store some information about the thread.

```
pthread_create(&t1, NULL, &routine, NULL);
```

To initialize a thread we need to call the function pthread_create. The first parameter is the pointer to the t1 variable, second is always a Null, third is the pointer to the function and last is the arguments of the function which is called in the third parameter.

```
pthread_join(t1, Null);
```

After creating a thread, we need to wait so that the thread does not execute after the process has finished as we have to first execute the thread and then the process. The first parameter is the pthread_t variable and second is some pointer that gets the result from the thread.

```
return 0;
}
```

<u>For creating multiple threads:</u>

```
pthread_t t1, t2;

pthread_create(&t1, Null, &routine, Null);
pthread_create(&t2, Null, &routine, Null);

pthread_join(t1, Null);
pthread_join(t2, Null);
```

=) pthread_create and pthread_join returns zero if the execution was successful.