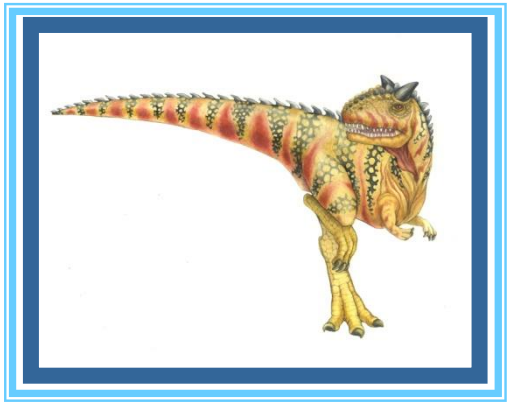
Threads

By,

Md. Biplob Hosen



Introduction to threads

- Thread is a lightweight process.
- Each thread runs strictly sequentially and has its own program counter and stack to keep track of where it is.
- Threads share the CPU just as processes do: first one thread runs, then another does.
- Threads can create child threads and can block waiting for system calls to complete.
- A traditional**(heavyweight)** process has single thread of control. If a process has multiple thread of control, it can perform more task at a time.

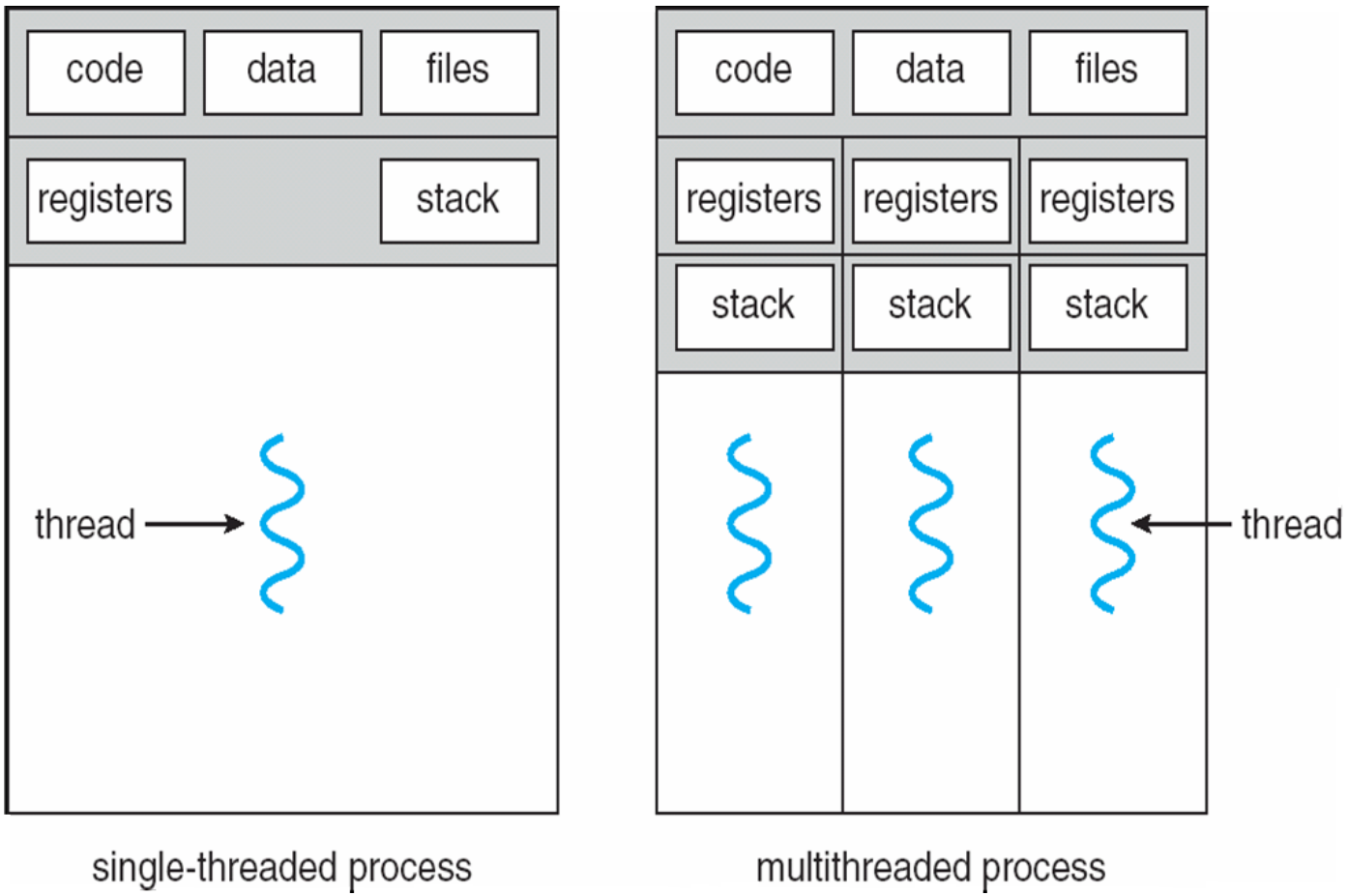
Introduction to threads…

- All threads have exactly the same address space. They share code section, data section, and OS resources (open files & signals). They share the same global variables. One thread can read, write, or even completely wipe out another thread's stack.
- Threads can be in any one of several states: running, blocked, ready, or terminated.

Example

- **A process as a house**
  - A house is really a container, with certain attributes (such as the amount of floor space, the number of bedrooms, and so on).
  - the house really doesn't actively *do* anything on its own — it's a passive object. This is effectively what a process is.
- **The occupants as threads**
  - The people living in the house are the *active* objects — they're the ones using the various rooms, watching TV, cooking, taking showers, and so on.
  - The occupants of a house can be considered as thread.
- **Single threaded**
  - If you've ever lived on your own, then you know what this is like — you *know* that you can do *anything* you want in the house at *any* time, because there's nobody else in the house. If you want to turn on the stereo, use the washroom, have dinner — whatever — you just go ahead and do it.
- **Multi threaded**
  - Things change dramatically when you add another person into the house. Let's say you get married, so now you have a spouse living there too.

Single and Multithreaded Processes



Motivation

- In certain situation, a single application may need to perform several similar tasks. For example:
  - A web server accepts client request for web pages, images, and so on.
  - A busy web server may have several (perhaps thousand) of clients concurrently accessing it.
  - If the web server ran as a traditional single-threaded process, it would be able to service only one client at a time.
  - Therefore, the amount of time that a client might have to wait for its request to be serviced could be enormous.
  - One solution is to have the server run as a single process that accepts requests. When the server receives a request, it creates a separate process to service that request.
  - But process creation is time consuming and resource intensive.
  - It is generally more efficient to use one process that contains multiple threads.
  - In the case of web server, instead of creating separate process, a server would create separate thread that would listen for client request; when a request was made, rather then creating another process, the server would create another thread to service the request.

Benefits of using Thread

- Responsiveness:
  - Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. For example, a multithreaded web browser could still allow user interaction in one thread while an image was being loaded in another thread.
- Resource Sharing:
  - By default, threads share the memory and the resources of the process to which they belong. 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.

Benefits of using Thread…

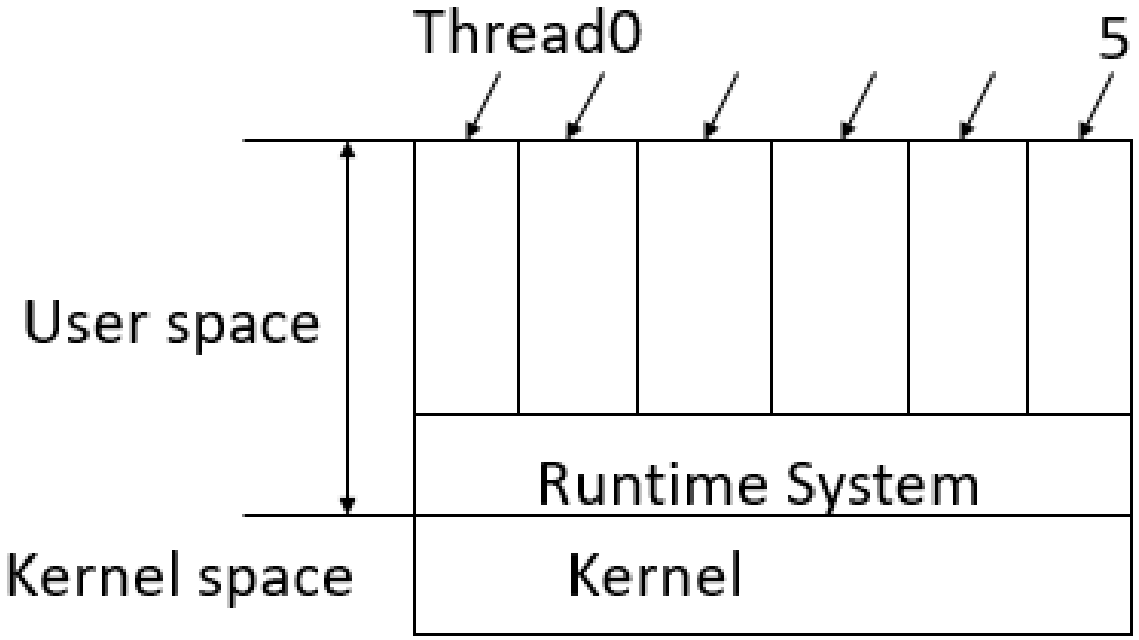
- Economy:
  - Allocating memory and resources for process creation is costly. As thread shares resource of the process to which they belong, it is more economical to create and context-switch threads than processes.
- Utilization of multiprocessor architectures:
  - The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors.

Thread Library

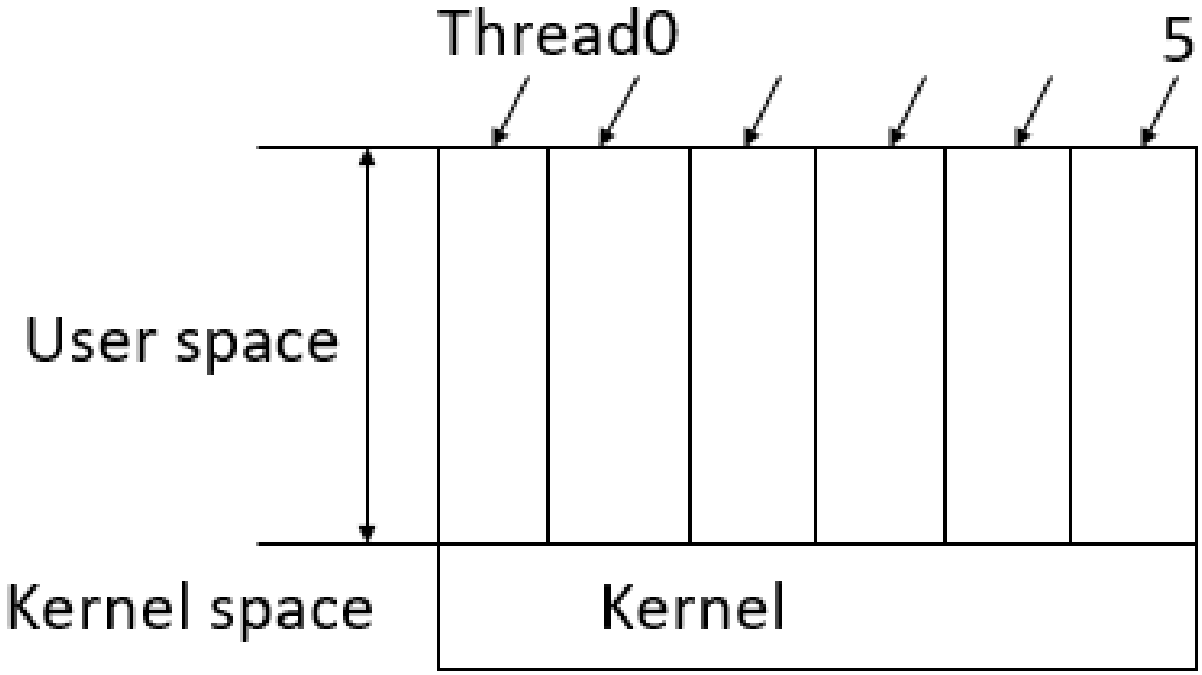
- A thread library provides the programmer an API for creating and managing threads.
- There are two primary ways of implementing thread libraries:

1. **User Thread:**
   - Thread management done by user-level threads library: The library provides support for thread creation, scheduling, and management.
   - All codes and data structures for the library exists in user space. Therefore, invoking a function in the library results in a local function call in user space and not a system call.
2. **Kernel Thread:**
   - Supported by the Kernel directly. Kernel performs the thread creation, scheduling and management inside the kernel
   - In this case, code and data structures for library exists in kernel space. Invoking a function in the API for the library typically results in a system call to the kernel.

Implementing threads in user space

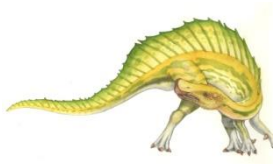


Implementing threads in kernel space



Advantage of User Thread

- It is cheap to create and destroy threads because all thread administration is kept in the user's address space.
- User-level threads package can be implemented on an operating system that does not support threads. For example, the UNIX system.
- Switching thread context can often be done in just a few instructions.
  - Basically only the values of CPU registers need to be stored and subsequently reloaded with the previously stored values of the thread to which it is being switched.
  - There is no need to change MMU, flush the TLB so on.
- User-level threads scale well. Kernel threads require some table space and stack space in the kernel, which can be a problem if there are a very large number of threads.



Dis-advantage of User Thread

- Blocking system calls are difficult to implement. Letting one thread make a system call that will block the thread will stop all the threads.
- Page faults. If a thread causes a page faults, the kernel does not know about the threads. It will block the entire process until the page has been fetched, even though other threads might be runnable.
- If a thread starts running, no other thread in that process will ever run unless the first thread voluntarily gives up the CPU.
- For the applications that are essentially CPU bound and rarely block, there is no point of using threads. Because threads are most useful if one thread is blocked, then another thread can be used.

Advantage of Kernel Thread

- The kernel knows about and manages the threads. No runtime system is needed. When a thread wants to create a new thread or destroy an existing thread, it makes a kernel call, which then does the creation and destruction.
- To manage all the threads, the kernel has one table per process with one entry per thread.
- When a thread blocks, the kernel can run either another thread from the same process or a thread from a different process.

Dis-advantage of Kernel Thread

- Here, every thread operation (creation, deletion, synchronization etc.) will have to be carried out by the kernel which requires a system call.
- Again switching thread context here is as expensive as switching process context.
- Therefore, most of the benefits of using threads instead of processes disappears.