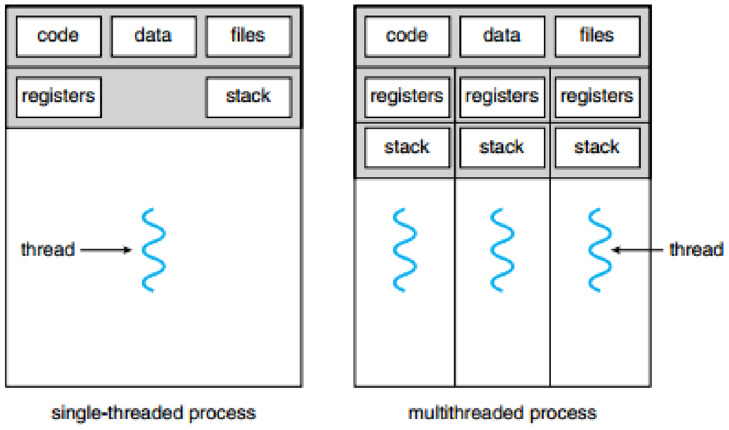
Threads

1 Introduction

A thread is a basic unit of CPU utilization; it comprises a **thread ID, a program counter, a register set, and a stack**. It 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.



Threads have same properties as of the process so they are called as **light-weight processes**. In some circumstances, each thread might need its own copy of certain data. We will call such data **thread-specific data**.

- A multitasking OS gives the perception of 2 or more jobs/processes running at the same time by dividing system resources amongst these tasks and switching rapidly between them.
- Thread-based multitasking involves running 2 or more threads concurrently to multi-task. Thread is the smallest unit and it requires less overhead to create and communicate compared to a process making it faster than the "heavy-weight" process-based multitasking.

2 Types

User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system.

User-Level Thread

It is implemented in the user level library and are not created using system calls. Kernel does not know about them. Thread switching are fast as no OS calls need to be made and requires no hardware support. If one thread performs blocking operation, the entire process is blocked. These threads are designed to be dependent unlike kernel threads.

Kernel-Level Thread

Kernel knows and manages the threads. Instead of thread table in each process, the kernel itself has thread table (a master one) that keeps track of all the threads in the system. In addition kernel also maintains the traditional process table to keep track of the processes. OS kernel provides system call to create and manage threads.

3 Benefits

- **Responsiveness**: Increased **throughput**. Completed executions are returned immediately.
- **Faster Context Switch:** Context switch time between threads is lower compared to process.
- **Economy:** Allocating memory and resources for process creation is costly. In Solaris, for example, creating a process is about thirty times slower than creating a thread, and context switching is about five times slower.
- **Scalability:** Multiple threads can run on multiple processors.
- **Communication:** Threads share resources like common address space so inter-communication is easier.

4 Multithreading models

Many-to-One

It maps many user-level threads to one kernel thread. Thread management is done by the thread library in user. space, so it is efficient; but the entire process will block if a thread makes a blocking system call. Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors.

One-to-One

It provides **more concurrency** than the many-to-one model by allowing another thread to run when a thread makes a blocking system call; it also allows multiple threads to run in parallel on multiprocessors. The drawback is that creating a user thread requires putting up with the overhead of creating the corresponding kernel thread. Therefore, most implementations of this model restrict the number of threads supported by the system.

Many-to-Many

The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads.Kernel level threads are specific to the machine. Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.