that process.

8.3 THREADS

Threads are a popular way to improve application performance through parallelism. In traditional operating systems the basic unit of CPU utilization is a process. Each process has its own program counter, its own register states, its own stack, and its own address space. On the other hand, in operating systems with threads facility, the basic unit of CPU utilization is a thread. In these operating systems, a process consists of an address space and one or more threads of control [Fig. 8.6(*b*)]. Each thread of a process has its own program counter, its own register states, and its own stack. But all the threads of a process share the same address space. Hence they also share the same global variables. In addition, all threads of a process also share the same set of operating system resources, such as open files, child processes, semaphores, signals, accounting information, and so on. Due to the sharing of address space, there is no protection between the threads of a process. However, this is not a problem. Protection between processes is needed because different processes may belong to different users. But a process (and hence all its threads) is always owned by a single user. Therefore, protection between multiple threads of a process is not necessary. If protection is required between two threads of a process, it is preferable to put them in different processes, instead of putting them in a single process.

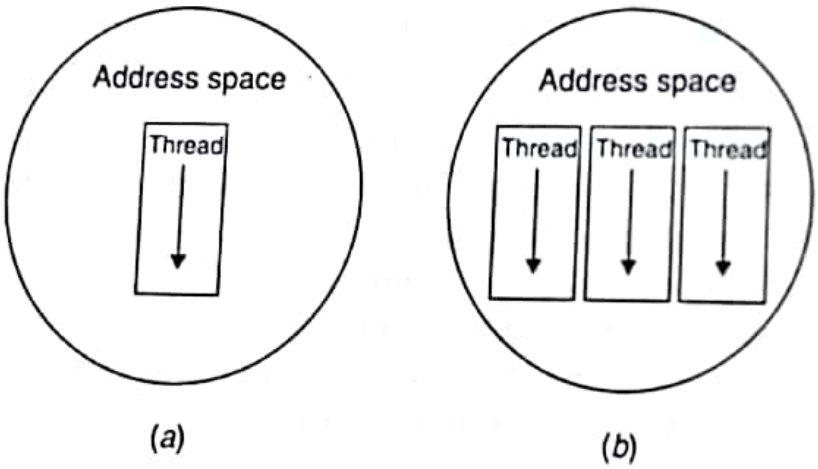


Fig. 8.6 (*a*) Single-threaded and (*b*) multithreaded processes. A single-threaded process corresponds to a process of a traditional operating system.

Threads share a CPU in the same way as processes do. That is, on a uniprocessor, threads run in quasi-parallel (time sharing), whereas on a shared-memory multiprocessor, as many threads can run simultaneously as there are processors. Moreover, like traditional processes, threads can create child threads, can block waiting for system calls to complete, and can change states during their course of execution. At a particular instance of time, a thread can be in any one of several states: running, blocked, ready, or terminated. Due to these similarities, threads are often viewed as miniprocesses. In fact, in operating systems

8.3.1 Motivations for Using Threads

The main motivations for using a multithreaded process instead of multiple single-threaded processes for performing some computation activities are as follows:

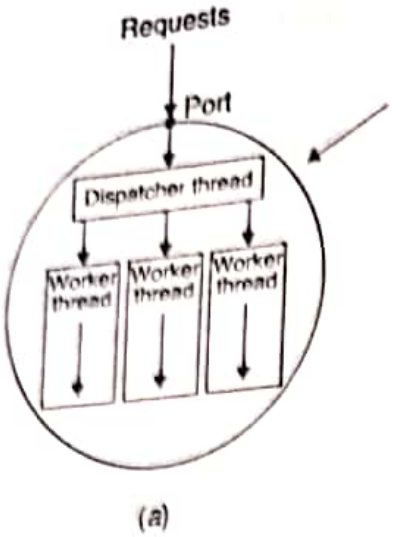
1. The overheads involved in creating a new process are in general considerably greater than those of creating a new thread within a process.
2. Switching between threads sharing the same address space is considerably cheaper than switching between processes that have their own address spaces.
3. Threads allow parallelism to be combined with sequential execution and blocking system calls [Tanenbaum 1995]. Parallelism improves performance and blocking system calls make programming easier.
4. Resource sharing can be achieved more efficiently and naturally between threads of a process than between processes because all threads of a process share the same address space.

These advantages are elaborated below.

The overheads involved in the creation of a new process and building its execution environment are liable to be much greater than creating a new thread within an existing process. This is mainly because when a new process is created, its address space has to be created from scratch, although a part of it might be inherited from the process's parent process. However, when a new thread is created, it uses the address space of its process that need not be created from scratch. For instance, in case of a kernel-supported virtual-memory system, a newly created process will incur page faults as data and instructions are referenced for the first time. Moreover, hardware caches will initially contain no data values for the new process, and cache entries for the process's data will be created as the process executes. These overheads may also occur in thread creation, but they are liable to be less. This is because when the newly created thread accesses code and data that have recently been accessed by other threads within the process, it automatically takes advantage of any hardware or main memory caching that has taken place.

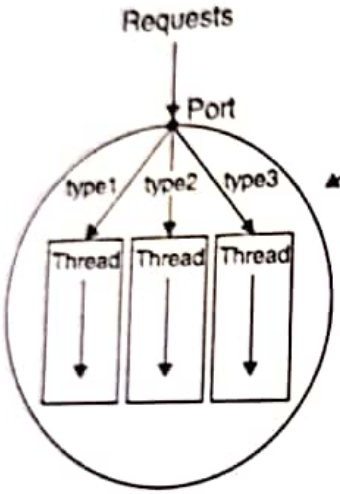
Threads also minimize context switching time, allowing the CPU to switch from one unit of computation to another unit of computation with minimal overhead. Due to the sharing of address space and other operating system resources among the threads of a process, the overhead involved in CPU switching among peer threads is very small as compared to CPU switching among processes having their own address spaces. This is the reason why threads are called lightweight processes.

To clarify how threads allow parallelism to be combined with sequential execution and blocking system calls, let us consider the different ways in which a server process can be constructed. One of the following three models may be used to construct a server process (e.g., let us consider the case of a file server) [Tanenbaum 1995]:



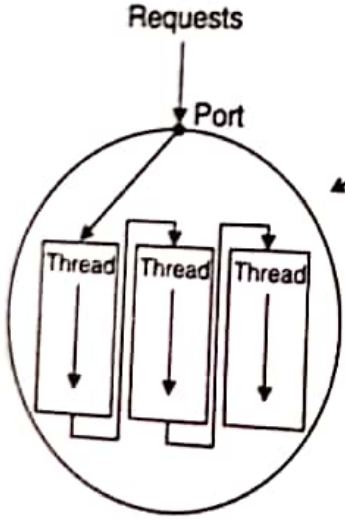
A server process for processing incoming requests

(a)



A server process for processing incoming requests that may be of three different types, each type of request being handled by a different thread

(b)



A server process for processing incoming requests, each request processed in three steps, each step handled by a different thread and output of one step used as input to the next step

(c)

Fig. 8.7 Models for organizing threads: (*a*) dispatcher-workers model; (*b*) team model; (*c*) pipeline model.