modern versions of UNIX, among others. In this section, we give a general description of multithreading; the details of the Windows, Solaris, and Linux approaches will be discussed later in this chapter.

In a multithreaded environment, a process is defined as the unit of resource allocation and a unit of protection. The following are associated with processes:

- A virtual address space that holds the process image
- Protected access to processors, other processes (for interprocess communication), files, and I/O resources (devices and channels)

Within a process, there may be one or more threads, each with the following:

- A thread execution state (Running, Ready, etc.)
- A saved thread context when not running; one way to view a thread is as an independent program counter operating within a process
- An execution stack
- Some per-thread static storage for local variables
- Access to the memory and resources of its process, shared with all other threads in that process

Figure 4.2 illustrates the distinction between threads and processes from the point of view of process management. In a single-threaded process model (i.e., there is no distinct concept of thread), the representation of a process includes its process control block and user address space, as well as user and kernel stacks to manage the call/return behavior of the execution of the process. While the process is running, it controls the processor registers. The contents of these registers are saved when the process is not running. In a multithreaded environment, there is still a single process

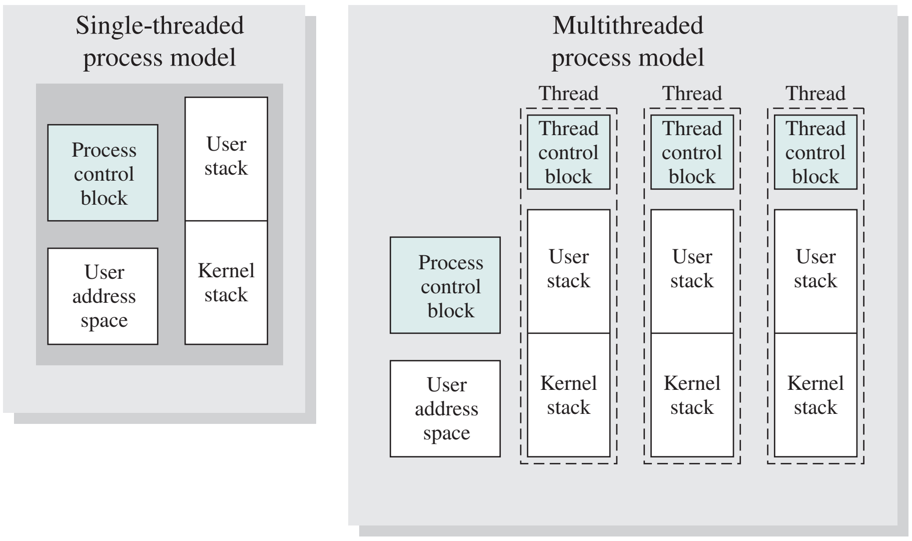

Figure 4.2 Single-Threaded and Multithreaded Process Models

control block and user address space associated with the process, but now there are separate stacks for each thread, as well as a separate control block for each thread containing register values, priority, and other thread-related state information.

Thus, all of the threads of a process share the state and resources of that process. They reside in the same address space and have access to the same data. When one thread alters an item of data in memory, other threads see the results if and when they access that item. If one thread opens a file with read privileges, other threads in the same process can also read from that file.

The key benefits of threads derive from the performance implications:

1. It takes far less time to create a new thread in an existing process, than to create a brand-new process. Studies done by the Mach developers show that thread creation is ten times faster than process creation in UNIX [TEVA87].
2. It takes less time to terminate a thread than a process.
3. It takes less time to switch between two threads within the same process than to switch between processes.
4. Threads enhance efficiency in communication between different executing programs. In most operating systems, communication between independent processes requires the intervention of the kernel to provide protection and the mechanisms needed for communication. However, because threads within the same process share memory and files, they can communicate with each other without invoking the kernel.

Thus, if there is an application or function that should be implemented as a set of related units of execution, it is far more efficient to do so as a collection of threads, rather than a collection of separate processes.

An example of an application that could make use of threads is a file server. As each new file request comes in, a new thread can be spawned for the file management program. Because a server will handle many requests, many threads will be created and destroyed in a short period. If the server runs on a multiprocessor computer, then multiple threads within the same process can be executing simultaneously on different processors. Further, because processes or threads in a file server must share file data and therefore coordinate their actions, it is faster to use threads and shared memory than processes and message passing for this coordination.

The thread construct is also useful on a single processor to simplify the structure of a program that is logically doing several different functions.

[LETW88] gives four examples of the uses of threads in a single-user multiprocessing system:

1. **Foreground and background work:** For example, in a spreadsheet program, one thread could display menus and read user input, while another thread executes user commands and updates the spreadsheet. This arrangement often increases the perceived speed of the application by allowing the program to prompt for the next command before the previous command is complete.
2. **Asynchronous processing:** Asynchronous elements in the program can be implemented as threads. For example, as a protection against power failure, one can design a word processor to write its random access memory (RAM)

buffer to disk once every minute. A thread can be created whose sole job is periodic backup and that schedules itself directly with the OS; there is no need for fancy code in the main program to provide for time checks or to coordinate input and output.

3. **Speed of execution:** A multithreaded process can compute one batch of data while reading the next batch from a device. On a multiprocessor system, multiple threads from the same process may be able to execute simultaneously. Thus, even though one thread may be blocked for an I/O operation to read in a batch of data, another thread may be executing.
4. **Modular program structure:** Programs that involve a variety of activities or a variety of sources and destinations of input and output may be easier to design and implement using threads.

In an OS that supports threads, scheduling and dispatching is done on a thread basis; hence, most of the state information dealing with execution is maintained in thread-level data structures. There are, however, several actions that affect all of the threads in a process, and that the OS must manage at the process level. For example, suspension involves swapping the address space of one process out of main memory to make room for the address space of another process. Because all threads in a process share the same address space, all threads are suspended at the same time. Similarly, termination of a process terminates all threads within that process.

Thread Functionality

Like processes, threads have execution states and may synchronize with one another. We look at these two aspects of thread functionality in turn.

Thread States As with processes, the key states for a thread are Running, Ready, and Blocked. Generally, it does not make sense to associate suspend states with threads because such states are process-level concepts. In particular, if a process is swapped out, all of its threads are necessarily swapped out because they all share the address space of the process.

There are four basic thread operations associated with a change in thread state [ANDE04]:

1. **Spawn:** Typically, when a new process is spawned, a thread for that process is also spawned. Subsequently, a thread within a process may spawn another thread within the same process, providing an instruction pointer and arguments for the new thread. The new thread is provided with its own register context and stack space and placed on the Ready queue.
2. **Block:** When a thread needs to wait for an event, it will block (saving its user registers, program counter, and stack pointers). The processor may then turn to the execution of another ready thread in the same or a different process.
3. **Unblock:** When the event for which a thread is blocked occurs, the thread is moved to the Ready queue.
4. **Finish:** When a thread completes, its register context and stacks are deallocated.