A significant issue is whether the blocking of a thread results in the blocking of the entire process. In other words, if one thread in a process is blocked, does this prevent the running of any other thread in the same process, even if that other thread is in a ready state? Clearly, some of the flexibility and power of threads is lost if the one blocked thread blocks an entire process.

We will return to this issue subsequently in our discussion of user-level versus kernel-level threads, but for now, let us consider the performance benefits of threads that do not block an entire process. Figure 4.3 (based on one in [KLEI96]) shows a program that performs two remote procedure calls (RPCs)[2] to two different hosts to obtain a combined result. In a single-threaded program, the results are obtained in sequence, so the program has to wait for a response from each server in turn. Rewriting the program to use a separate thread for each RPC results in a substantial speedup. Note if this program operates on a uniprocessor, the requests must be generated sequentially and the results processed in sequence; however, the program waits concurrently for the two replies.

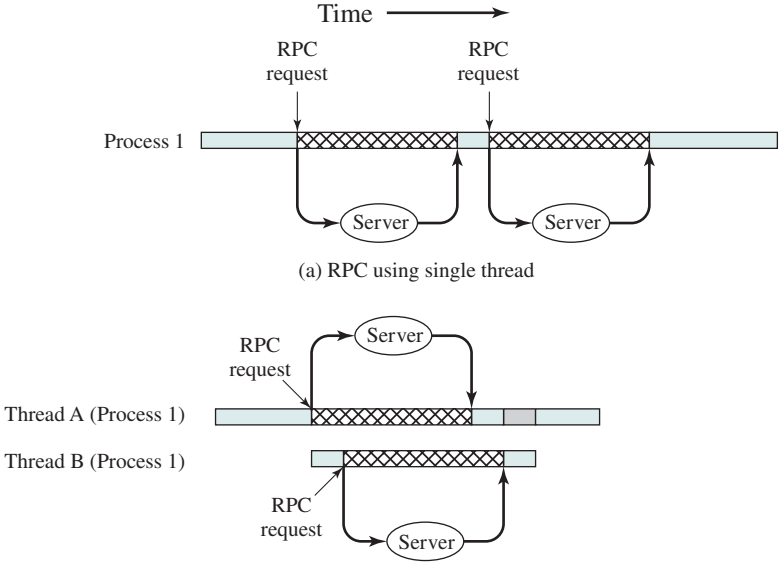

Figure 4.3 Remote Procedure Call (RPC) Using Threads

[2]An RPC is a technique by which two programs, which may execute on different machines, interact using procedure call/return syntax and semantics. Both the called and calling programs behave as if the partner program were running on the same machine. RPCs are often used for client/server applications and will be discussed in Chapter 16.

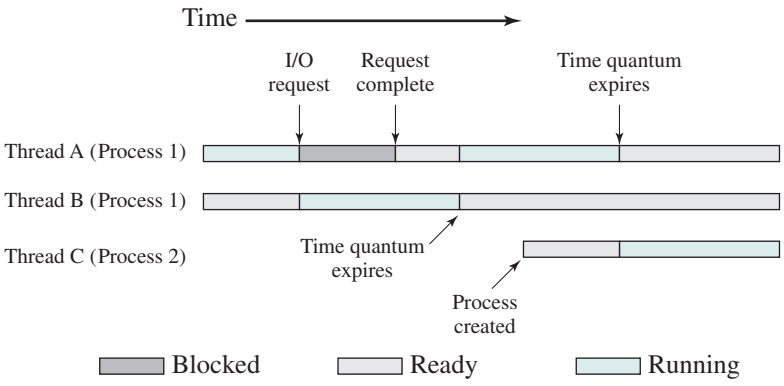

Figure 4.4 **Multithreading Example on a Uniprocessor**

On a uniprocessor, multiprogramming enables the interleaving of multiple threads within multiple processes. In the example of Figure 4.4, three threads in two processes are interleaved on the processor. Execution passes from one thread to another either when the currently running thread is blocked or when its time slice is exhausted.[3]

THREAD SYNCHRONIZATION All of the threads of a process share the same address space and other resources, such as open files. Any alteration of a resource by one thread affects the environment of the other threads in the same process. It is therefore necessary to synchronize the activities of the various threads so that they do not interfere with each other or corrupt data structures. For example, if two threads each try to add an element to a doubly linked list at the same time, one element may be lost or the list may end up malformed.

The issues raised and the techniques used in the synchronization of threads are, in general, the same as for the synchronization of processes. These issues and techniques will be the subject of Chapters 5 and 6.

4.2 TYPES OF THREADS

User-Level and Kernel-Level Threads

There are two broad categories of thread implementation: user-level threads (ULTs) and kernel-level threads (KLTs).[4] The latter are also referred to in the literature as *kernel-supported threads* or *lightweight processes*.

USER-LEVEL THREADS In a pure ULT facility, all of the work of thread management is done by the application and the kernel is not aware of the existence of threads.

[3]In this example, thread C begins to run after thread A exhausts its time quantum, even though thread B is also ready to run. The choice between B and C is a scheduling decision, a topic covered in Part Four.

[4]The acronyms ULT and KLT are not widely used, but are introduced for conciseness.

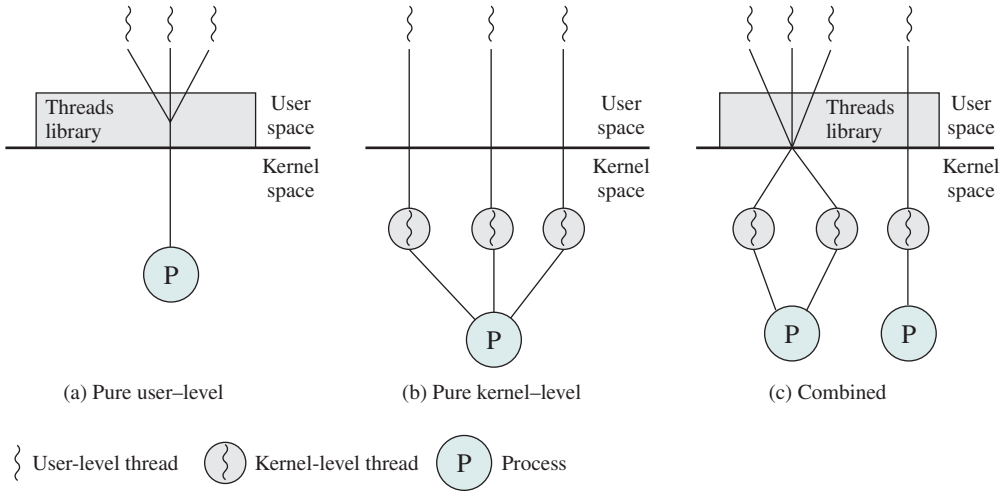

Figure 4.5 User-Level and Kernel-Level Threads

Figure 4.5a illustrates the pure ULT approach. Any application can be programmed to be multithreaded by using a threads library, which is a package of routines for ULT management. The threads library contains code for creating and destroying threads, for passing messages and data between threads, for scheduling thread execution, and for saving and restoring thread contexts.

By default, an application begins with a single thread and begins running in that thread. This application and its thread are allocated to a single process managed by the kernel. At any time that the application is running (the process is in the Running state), the application may spawn a new thread to run within the same process. Spawning is done by invoking the spawn utility in the threads library. Control is passed to that utility by a procedure call. The threads library creates a data structure for the new thread and then passes control to one of the threads within this process that is in the Ready state, using some scheduling algorithm. When control is passed to the library, the context of the current thread is saved, and when control is passed from the library to a thread, the context of that thread is restored. The context essentially consists of the contents of user registers, the program counter, and stack pointers.

All of the activity described in the preceding paragraph takes place in user space and within a single process. The kernel is unaware of this activity. The kernel continues to schedule the process as a unit and assigns a single execution state (Ready, Running, Blocked, etc.) to that process. The following examples should clarify the relationship between thread scheduling and process scheduling. Suppose process B is executing in its thread 2; the states of the process and two ULTs that are part of the process are shown in Figure 4.6a. Each of the following is a possible occurrence:

1. The application executing in thread 2 makes a system call that blocks B. For example, an I/O call is made. This causes control to transfer to the kernel. The kernel invokes the I/O action, places process B in the Blocked state, and