CSc33200: Operating Systems, CS-CCNY, Fall 2003 Jinzhong Niu October 1, 2003

Thread

1 Introduction

The characteristics of processes we have discussed so far may fall into two categories:

- **Resource ownership** From time to time, processes may be allocated control or ownership of various resources, including main memory, I/O devices, and files. The operating system may provide protection facilities to prevent illegal access to resources by non-owners.
- **Scheduling/execution** This aspect actually relates to the occupation of the processor, which is considered as a special resource. Every process has an execution path, and at any moment is associated with a state (Running, Ready, etc.). A process is the entity that is scheduled and dispatched by the operating system.

Why is the processor considered as a special resource? In my personal opinion, the first aspect involves the issues decided by the snapshots of a process, which may be thought static, while the second pays attention to the dynamic behavior of a process.

These two issues are taken apart not due to theoretical discussion purposes. Actually this point of view has been embodied in all kinds of realistic systems. In modern operating systems, the unit of dispatching is usually referred to as a **thread** or **lightweight process**. And the unit of resource ownership is still a **process**.

2 Multithreading

2.1 Concepts and examples

Almost all modern operating systems support the concept of thread by allowing multiple threads in a single process, which is referred to as **multithreading**, while the traditional approach of a single thread of execution per process, where the concept of thread is not embodied at all, is referred to as **single-threading**. These two approaches are depicted in Figure 1, in which a box represents a process and a downward curve a thread. As Figure 1 shows

- MS-DOS supports a single user process and a single thread.
- Some traditional UNIX systems are multiprogramming systems, thus support multiple user processes but only one execution path is allowed for each process.
- A *Java Virtual Machine* (JVM) is an example of a system of one process with multiple threads.
- Most modern operating systems, such as Windows 2000, Solaris, Linux, and OS/2, support multiple processes with multiple threads per process.

The last category is the concern of this section.

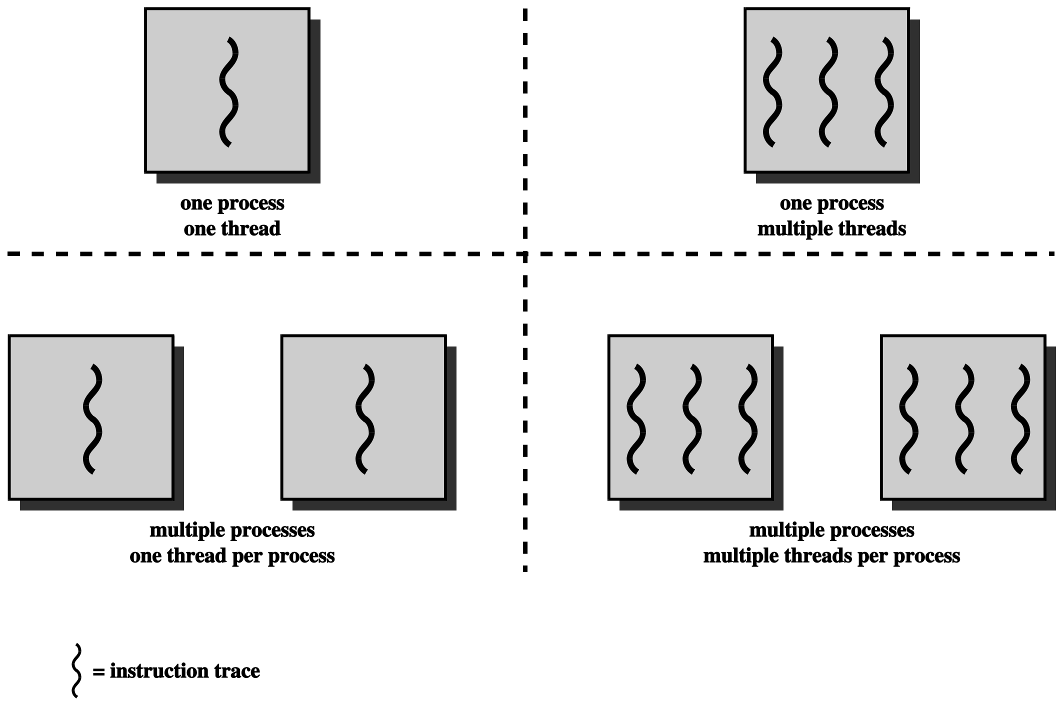

Figure 1: Threads and processes

2.2 Process model with consideration of threads

In a multithreaded environment, a process, as the unit of resource ownership and protection, is associated with:

- A virtual address space that holds the process image.

- Protected access to I/O devices, files, and other resources that are owned by other processes.

A thread otherwise has:

- A thread execution state.
- A saved thread context when not running. A simple way to view a thread is as an independent program counter within a process, indicating the position of the instruction that the thread is working on.
- a stack tracing the execution path.
- some space for local variables.

Figure 2 illustrates the distinction between single-threaded process model and multithreaded process model. As we stated before, a process image includes its PCB, user address space for

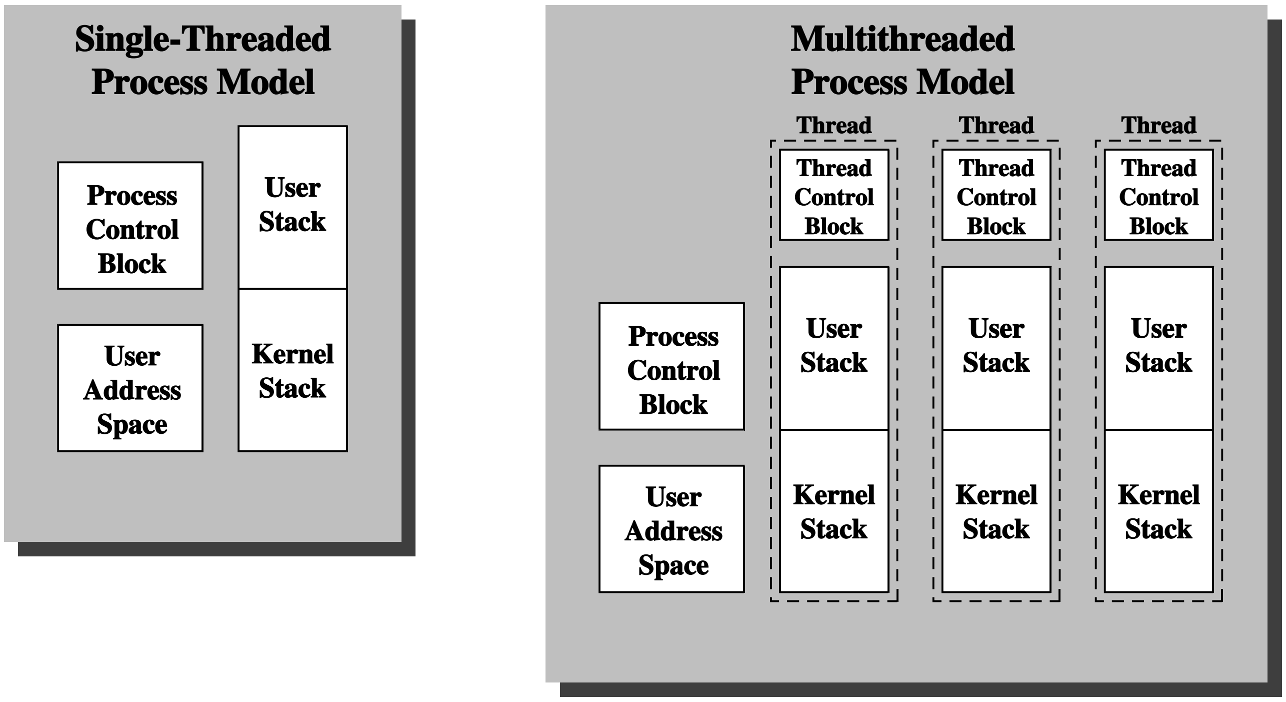

Figure 2: Single threaded and multithreaded process models

global variables and code, as well as user and kernel stacks tracing procedural invocations. In a multi-threading environment, there is still a single PCB and user address space associated with each single process, but multiple user stacks and kernel stacks are needed for multiple threads. A separate control block for each thread is also necessary to record the execution context, priority, and state information. Thus all threads within a process share the state of the process, reside in the same address space, and have access to the same collection of data. If the value of a global variable is altered by one thread, other threads can see the result by visiting the same variable.

2.3 Benefits of multi-threading

The introduction of thread is not to seek a perfection of a theoretical model, but does bring benefits:

- It takes far less time to create a new process in an existing process than to create a brand-new process.
- It takes less time to terminate a thread.
- It takes less time to do control switching between two threads within a same process than process switching.
- Threads make it easier to communicate between different execution traces. Sharing the same user address space enables the communication naturally but process communication demands special supports from the operating system.

Thus if there is a 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.

What's more, many systems or facilities in the real world are in nature concurrent systems. That is different parts of a system run at the same time. But due to the limitation of early computer systems, the parallel structures are forced to be implemented in a serial fashion. The availability of threads provides more bullets for the computers to reflect the real world in a straightforward way.

2.4 Application of threads

Take web servers as an example. Let's consider the way how the client requests are processed and responded by a web server in terms of the model of the execution traces inside it. With the development of technologies, the following models have been practised in turn by a variety of web servers:

- **Blocked model**

The simplest way is to set up only one process, which processes one user request at a time. Thus if another request arrives over the Internet while the process is still busy processing and not listening, the new request will simply not be received. This blocked model is similar as the early batch-processing operating systems, where a job will not be processed until all previous ones have been finished.

- **Multiplexing model**

  To solve the problem in the blocked model, a single process may respond to multiple requests in a time-sharing fashion like a time-sharing operating system. However if the work-load of the server is heavy, then each single request will have to wait a long time until completely served.

- **Forking model**

  Since the requests come in independently, a natural way to respond is to create a new process for each newly coming request. This is usually done by the invocation of the UNIX routine `fork()`, so this model is called forking model. But this model also has its problems. Each process created requires its own address space but main memory is a limited resource. Though the appearance of virtual memory technique allows the suspension of processes, but a process has to be in main memory to be executed. The swapping out and in consumes much time. And the creation of process also takes much time, so the performance of such a web server may be greatly impaired.

- **Process-pool model**

  This model is an improvement to the forking model. Instead of creating a process when a request arrives and terminating it when the process finishes, a set of processes may be created in the first place when the web server is launched at the very beginning. These processes are managed as a process pool and whenever a process is needed for processing a request, the pool will allocate a process for the task. When the processing is finished, the process is returned to the pool for future requests. This model to much extent avoids the overhead of process creation and termination.

- **Process-pool with multithreading**

  The story has not ended yet. Limited resource of the computer system cannot afford a number of processes in the pool. With the introduction of multithreading, the process-pool model may be revised to let a single process responsible for multiple requests (e.g. from a same user) with one thread for each request. In this way, the number of processes needed in the pool may be reduced significantly. The sharing of user address space among the threads also fits well the relevance between the requests.

From this abstract example, we can see how processes and threads work together to solve real problems.

3 Thread functionality

3.1 Thread states

Although a thread embodies the dynamic behavior of processes, and is the unit of scheduling and dispatching, there are however several actions that affect all of the threads in a process and the operating system must manage at the process level.

- Suspension means swapping the process image out of main memory. Since all threads share the same address space, all the threads must enter a Suspend state at the same time.
- Termination of a process terminates all threads within that process. For example, if a Java thread calls `System.exit(0)`, then the JVM process will exit and all the threads will be terminated.

Thus the key states for a thread are Running, Ready, and Blocked. Accordingly there are events that cause a thread state transition:

- **Spawn**

  When a new process is created, typically a thread for this process is also spawned. Later on, other threads may be spawned within the same address space. Every new thread is provided with its own register context and stack space and placed on the ready queue.
- **Block**

  When a thread needs to wait for an event, it will be blocked and its thread context be saved in its thread control block. The processor may then schedule another thread to run.
- **Unblock**

  Similar to the case regarding process, when a desired event occurs, the waiting thread will then be unblocked and moved to the ready queue.
- **Finish**

  When a thread completes, the space allocated for it will be deallocated.

Note that with multithreading the Blocked state of a process makes no more sense. While a thread is blocked, another thread in the same process may still run on the processor. For example, Figure 3 shows a program that performs two *remote procedure call* (RPC) to two different hosts to obtain a combined result.

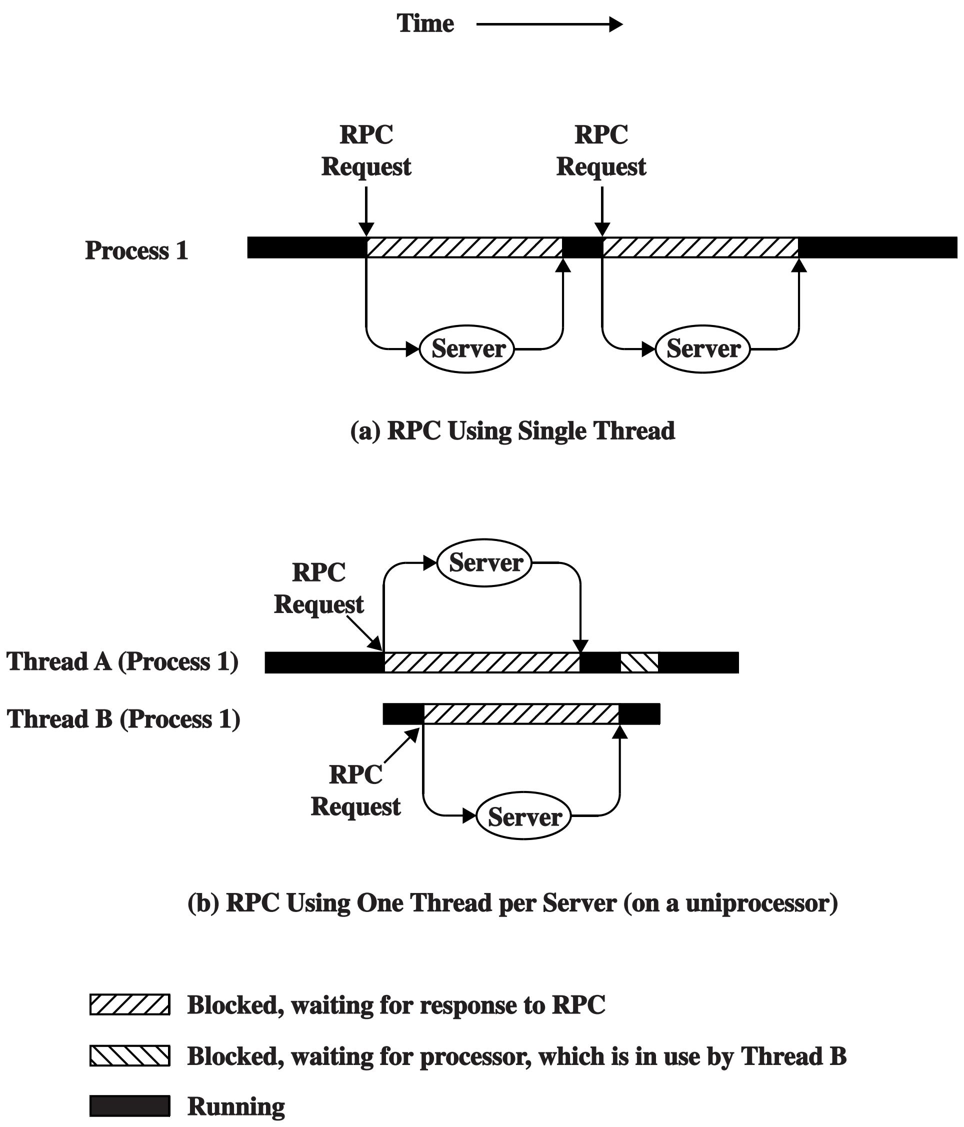
Time
RPC Request
RPC Request
Process 1
Server
Server
(a) RPC Using Single Thread
RPC Request
Server
Thread A (Process 1)
Thread B (Process 1)
RPC Request
Server
(b) RPC Using One Thread per Server (on a uniprocessor)
Blocked, waiting for response to RPC
Blocked, waiting for processor, which is in use by Thread B
Running

Figure 3: Remote procedure call using threads

An RPC is similar to a regular procedure call except that the RPC caller and the RPC callee may reside on different hosts. The transfer of parameters and results involves data transmission over the network instead merely local stack push and pop operations. The RPC mechanism is illustrated in Figure 4.

In a single-threaded system, the two requests have to be done in sequence, while in a multi-threaded environment, each request may be made by separate threads. Although on a uniprocessor system at any moment only one thread may be running, and the requests still have to be generated sequentially, the process however waits concurrently for the two replies.

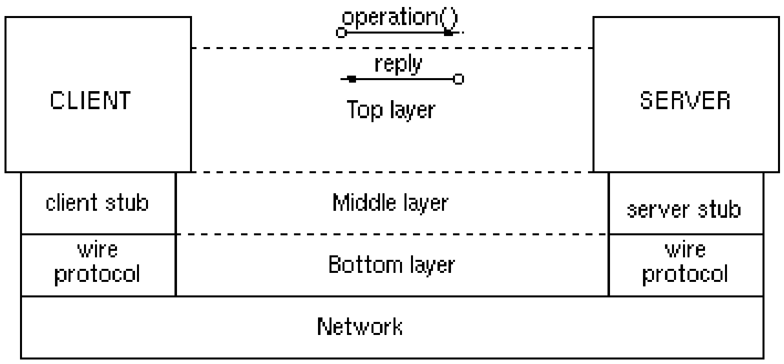

Figure 4: Remote procedure call mechanism

3.2 Thread synchronization

Since threads in a processor share the same address space, the result of a thread's operation is visible to other threads. This enables the communication between threads, but may cause problems as well. The issues raised here, concurrency and synchronization, will be discussed later.

4 User-level and kernel-level threads

We move on to consider the implementation issues of threads. There are two common ways to support threads: **user-level threads** and **kernel-level threads**. Figure 5 (a), (b), and (c) respectively show the two approaches and their combination.

4.1 User-level threads

In this approach, all of the work of thread management is done by the user application, and the operating system kernel is not aware of the existence of threads. Typically a *thread library* is available providing routines for manipulating threads (creating, terminating, scheduling, and saving and restoring thread context) and communication between threads as well. JVM implementations on Windows platform fall into this category.

Same as we have said, when a user process is created, a single thread is spawned and begins running in the process space. This thread may spawn a new thread some time in the same

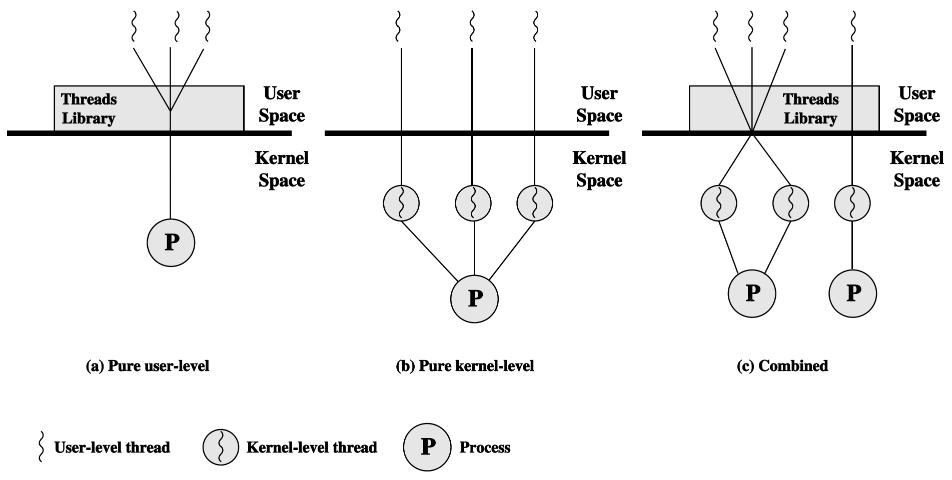

Figure 5: User-level and kernel-level threads

process by invoking the spawn utility in the threads library. The threads library creates a data structure for the new thread and then dispatch control to one of the threads in the currently running process. When the control switches between individual threads and the library, the thread context is also saved or restored accordingly.

The above activities all happen within the user address space. The kernel has no idea of these and always schedule the process as a unit and assign a single execution state to it. But how the thread state transition and process state transition work together consistently? Figure 6 gives an example. Suppose process B is executing in thread 2; the state of the process and the two threads are shown in Figure 6 (a). Now let's see what will happen in the following three occasions:

1. **Thread 2 makes a system call that blocks process B, e.g. an I/O call.** This causes control to transfer to the kernel. The kernel then starts the I/O operation, places process B in the Blocked state, and switches control to another process. Meanwhile the information the threads maintained by the threads library remains unchanged, so thread 2 is still perceived as running though it is not. The new snapshot is illustrated in Figure 6 (b).
2. **A timeout event occurs.** The kernel determines that process B has exhausted its time slice, so places it in the Ready state and dispatches another process. Similar to the first case, the thread states are still the same.
3. **Thread 2 has reached a point where it has to wait until thread 1 has performed a specific action.** Thus thread 2 enters a Blocked state and thread 1 becomes active and begins to run. The process itself remains in the Running state.

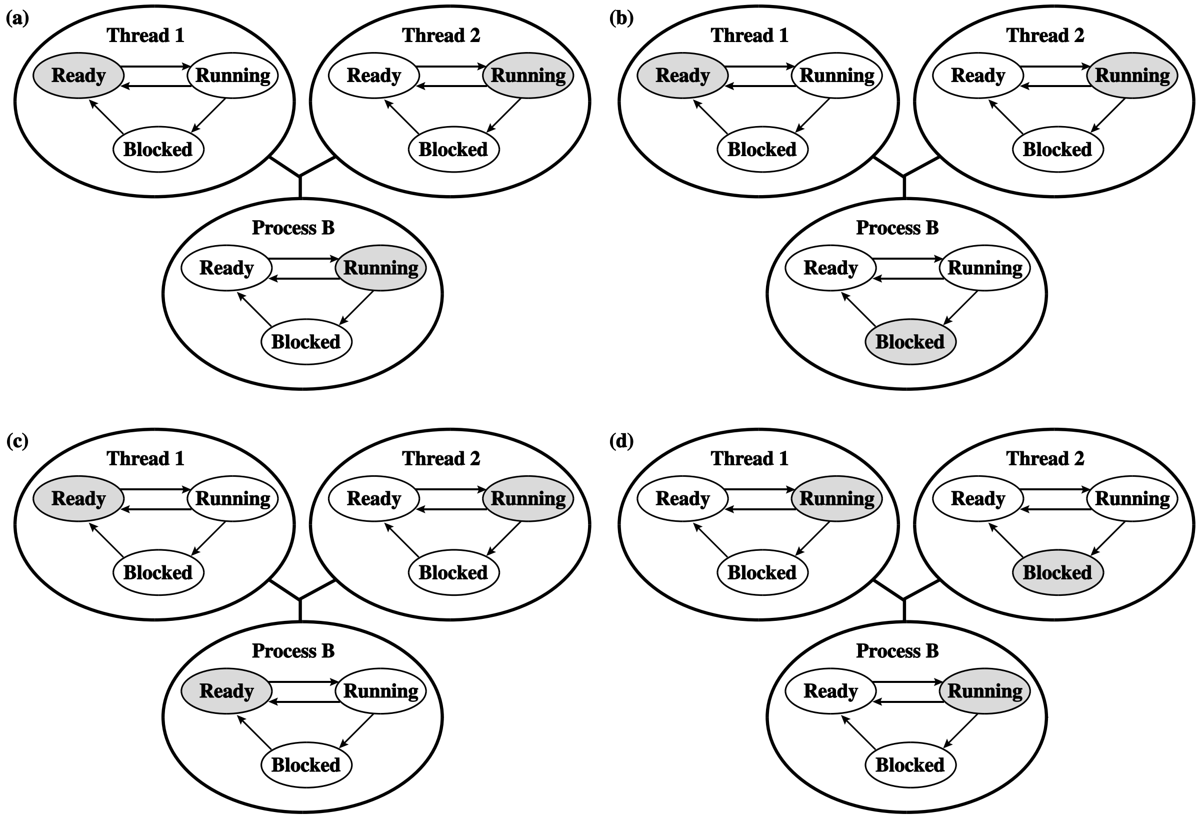

Figure 6: Examples of the relationships between user-level thread states and process states

4.2 Kernel-level threads

With a kernel-level thread implementation, all of the thread management work is done by the kernel. The operating system may also provides an application programming interface (API) to the user application, similar to the above threads library, but there is no thread management code in user address space. The examples include Windows 2000, Linux, and some implementation of JVM on Solaris.

Figure 5 (b) depicts this approach. The kernel maintains context information for every process and for individual threads within processes. Scheduling by the kernel is done on a thread basis.

4.3 Comparison between user-level and kernel-level threads

Compared with the kernel-level implementation, The user-level one has the following advantages:

1. **No mode switching**

Thread switching does not require mode switching, whose overhead is huge. The threads library works in the user address and there is no need to switch to the kernel mode. As described in the textbook, some experiments have shown that there may be an order of magnitude or more of difference between user-level threads and kernel threads regarding the time needed for various thread operations.

2. **Customizable thread scheduling**

   Thread scheduling can be tailored according to the nature of the application so that better performance may be achieved.

3. **System independent**

   User-level implementation may be easily ported to other platforms since the threads library is a set of application-level utilities.

On the other side of the story, user-level threads also possess explicit disadvantages:

1. **Process-level blocking**

   A blocking system call made by a thread in a process will cause all the threads in the process to be blocked. The textbook proposed two solutions to solve this problem, but seems useless.

2. **Undistributable threads**

   In a multi-processor system, the kernel assigns one process to only one processor at a time. Thus the multiple threads in a single process have no chance to run simultaneously.

Obviously the kernel-level threads do not have the above problems, but thread switching causes mode switching and may greatly impair the performance.

Thus some systems tried to combine the two approaches together, e.g. Solaris and some good results have been achieved in this direction.