



Threads



Chapter 4

Organization

- ▶ **An Introduction** 
- ▶ **Multithreading Models**
- ▶ **Thread Libraries**
- ▶ **Threading Issues**



Two Characteristics of Processes

- ▶ **Two potentially independent characteristics embodied by the concept of process:**
 - ▶ **Resource ownership** - process is allocated a virtual address space to hold the process image
 - ▶ **Scheduling/execution** - follows an execution path that may be interleaved with other processes
- ▶ **These two characteristics are indeed the essence of a process**



Threads and Processes

- ▶ **A number of operating systems treat these two characteristics independently**
 - ▶ The unit of dispatching is referred to as a ***thread*** (or ***lightweight process***)
 - ▶ The unit of resource ownership is referred to as a ***process*** or ***task***



Thread

- ▶ *A thread is a basic unit of CPU utilization; It comprises of :*
 - ▶ *A thread ID*
 - ▶ *A program counter*
 - ▶ *A register set*
 - ▶ *A stack*
- ▶ *A thread shares with other threads belonging to the same process:*
 - ▶ *Code section*
 - ▶ *Data section*
 - ▶ *Other OS resources such as opened files and signals*



Example (1)

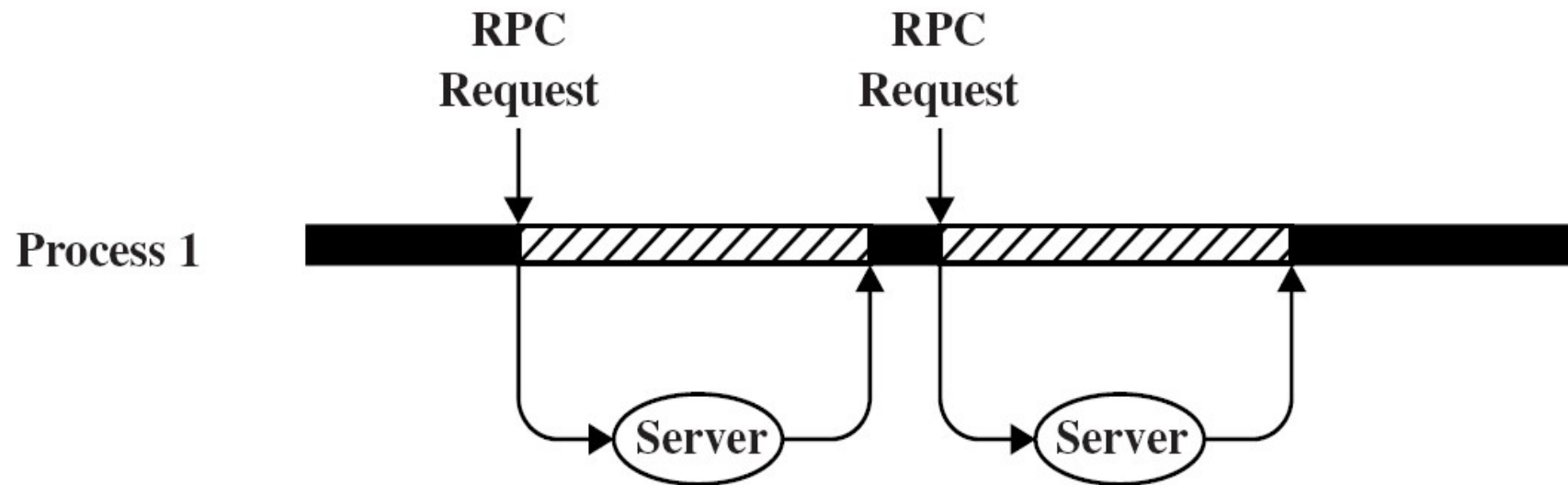
Scenario

- ▶ **A program performs two remote procedure calls (远程过程调用) to two different hosts to obtain a combined results**
- ▶ **Single-Threaded implementation**
 - ▶ The results are obtained in sequence → the program has to wait for a response from each host in turn
- ▶ **Multi-Threaded implementation**
 - ▶ Uses a separate thread for each RPC → results in a substantial speedup.



Example (2)

Single-Threaded Implementation



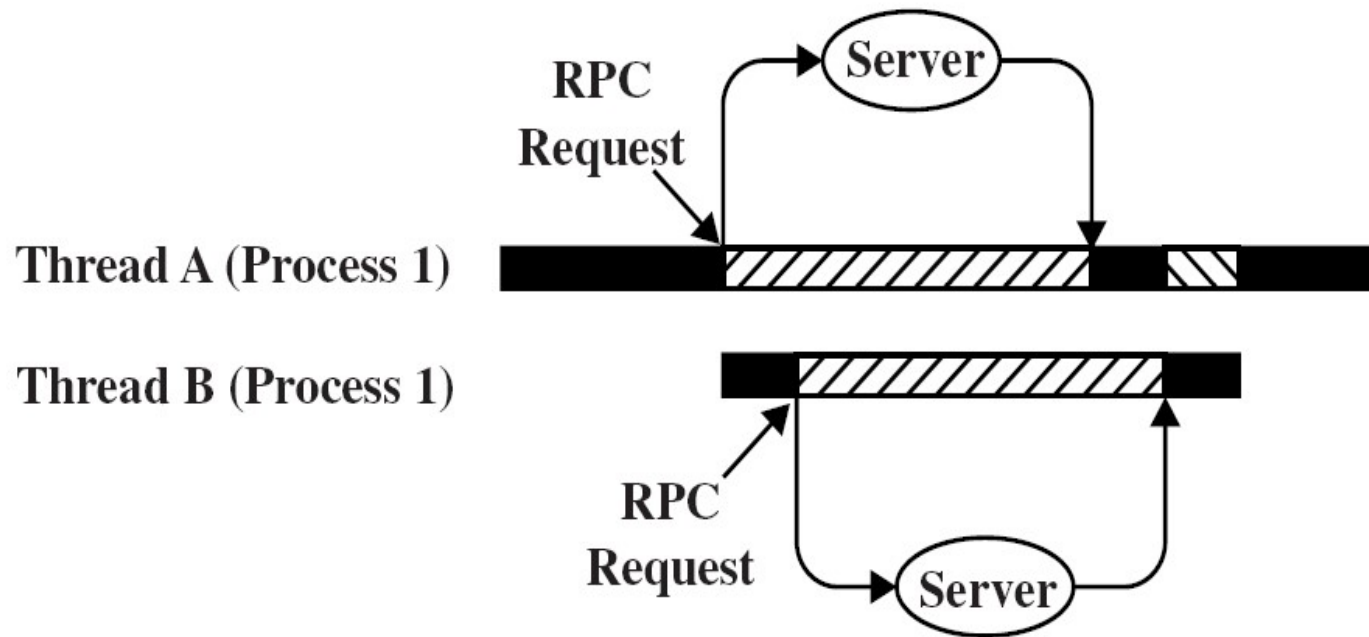
 Blocked, waiting for response to RPC

 Blocked, waiting for processor, which is in use by Thread B

 Running

Example (3)

Multithreaded Implementation

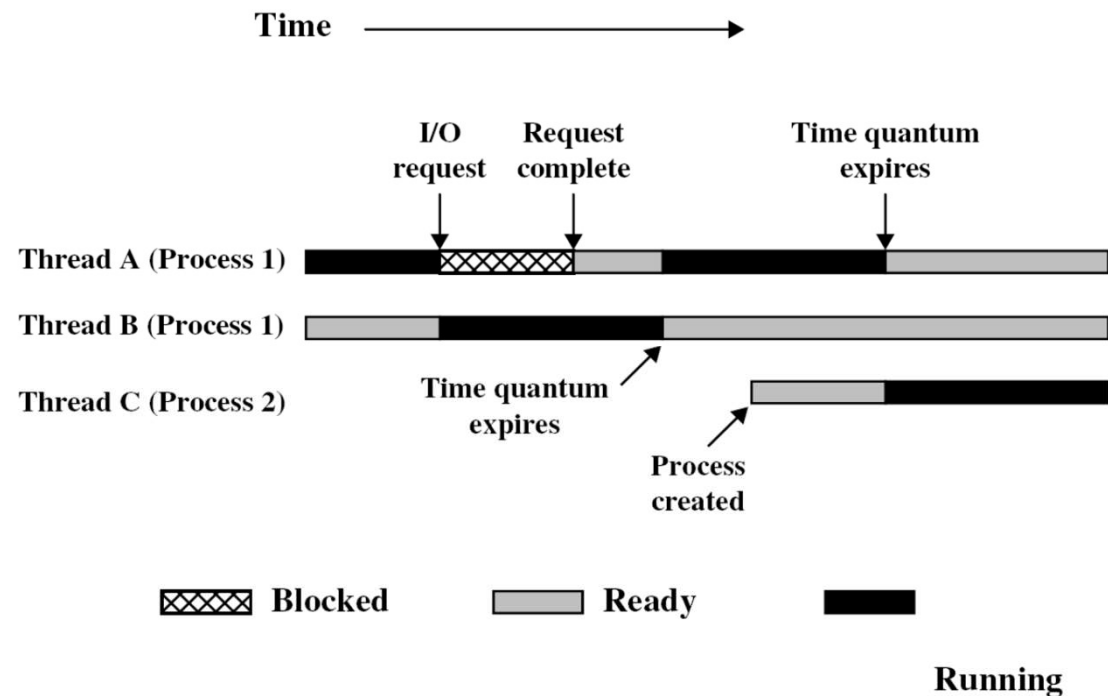


(b) RPC Using One Thread per Server (on a uniprocessor)



Interleaving Execution

- ▶ On a uniprocessor, multiprogramming enables the interleaving of multiple threads within multiple processes



Benefits of Muti-threaded Programming

- ▶ **Responsiveness**
- ▶ **Resource Sharing**
- ▶ **Economy**
- ▶ **Utilization of Multiprocessor Architectures**



Benefits of Threads (Compared to Processes)

Benefits, compared with multiple processes

- ▶ Less time to **create** a new thread than a process
- ▶ Less time to **terminate** a thread than a process
- ▶ Less time to **switch** between two threads within the same process
- ▶ Since threads within the same process share memory and files, they can **communicate** with each other without invoking the kernel
- ▶ May **speed up** the application!!
 - ▶ (Multiple CPUs) or (overlap I/O and computing)

Benefits same as multiple processes

Multithreading (1)

多线程

- ▶ **What is Multithreading**

- ▶ The ability to support multiple threads of execution within a single process
- ▶ In a multithreaded environment, a process is defined as the unit of resource allocation and a unit of protection.

- ▶ **Protection between threads within a single process?**

- ▶ Not necessary!! Why?

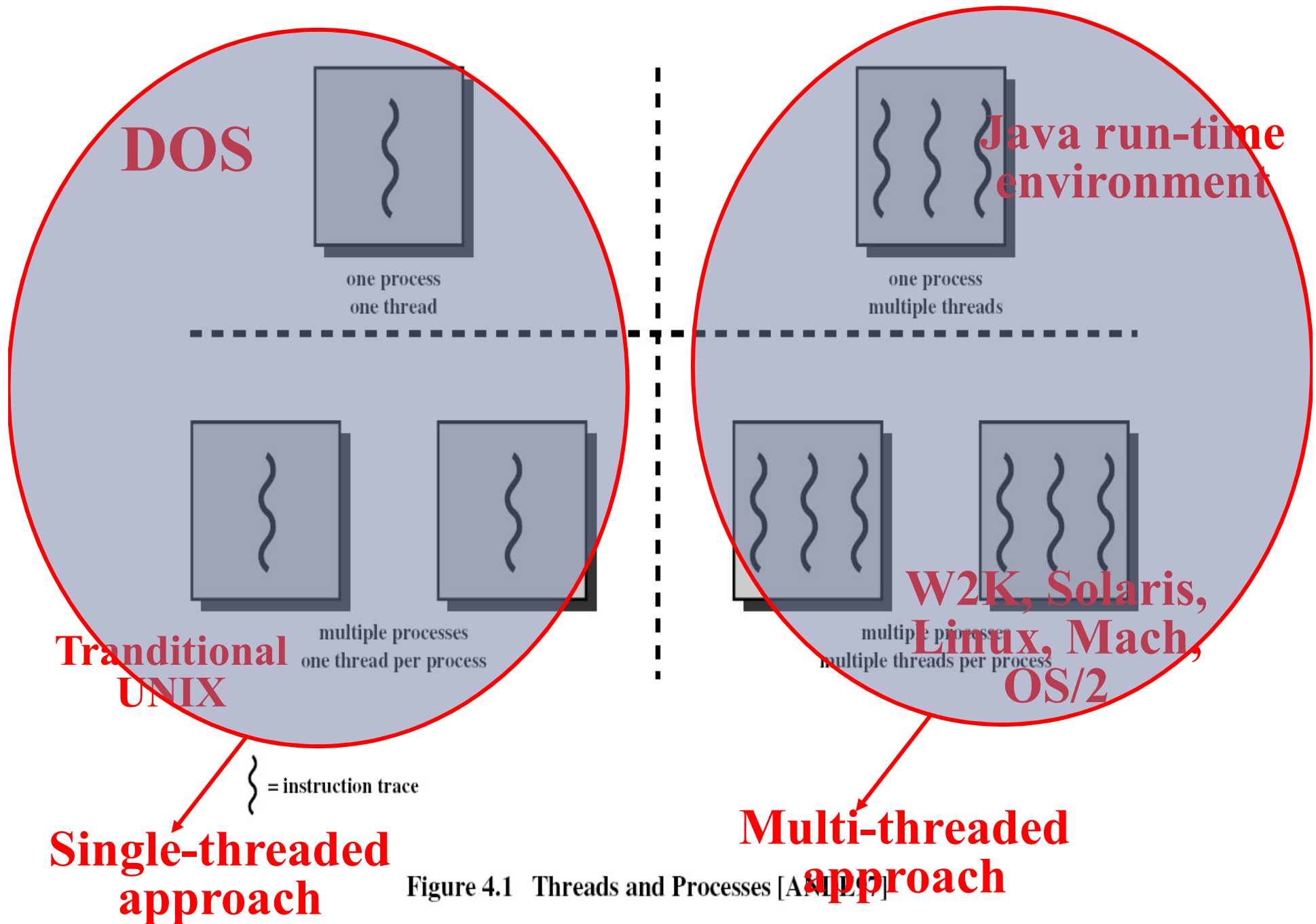


Multithreading (2)

- ▶ **Classification**

- ▶ **How many address spaces does the system have?**
 - ▶ **One or Many?**
- ▶ **How many threads in each address space?**
 - ▶ **One or Many?**





What are Associated with Processes?

- ▶ In multithreaded environment, a process is the unit of resource allocation and a unit of protection. It has
 - ▶ A ***virtual address space*** which holds the process image
 - ▶ ***Protected access to*** processors, other processes, files, and I/O resources



Items Private to a Thread

- ▶ An **execution state** (running, ready, etc.)
- ▶ Saved **thread context** when not running
- ▶ An execution **stack**
- ▶ Some per-thread **static storage** for local variables
- ▶ Access to the memory and resources of its process
 - ▶ all threads of a process share this

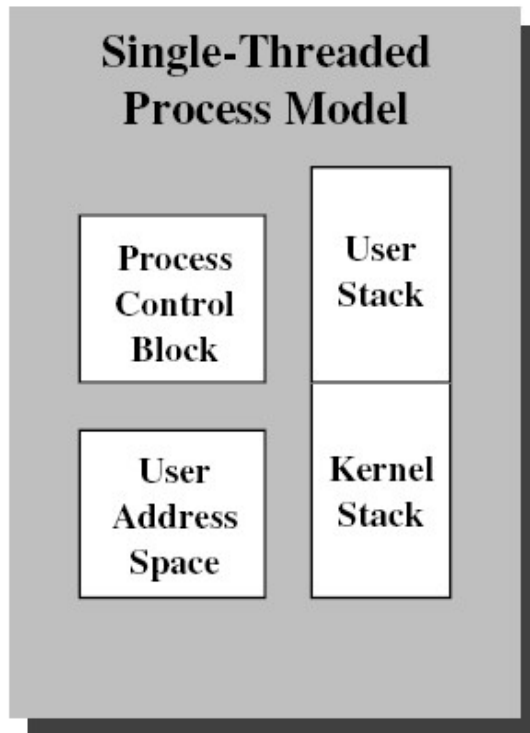


A Summary

Per process items	Per thread items
Address Space Global Variables Open files Child Processes Pending alarms Signals and signal handlers Accounting information	Program Counter Registers Stack State



单线程进程模型



多线程进程模型

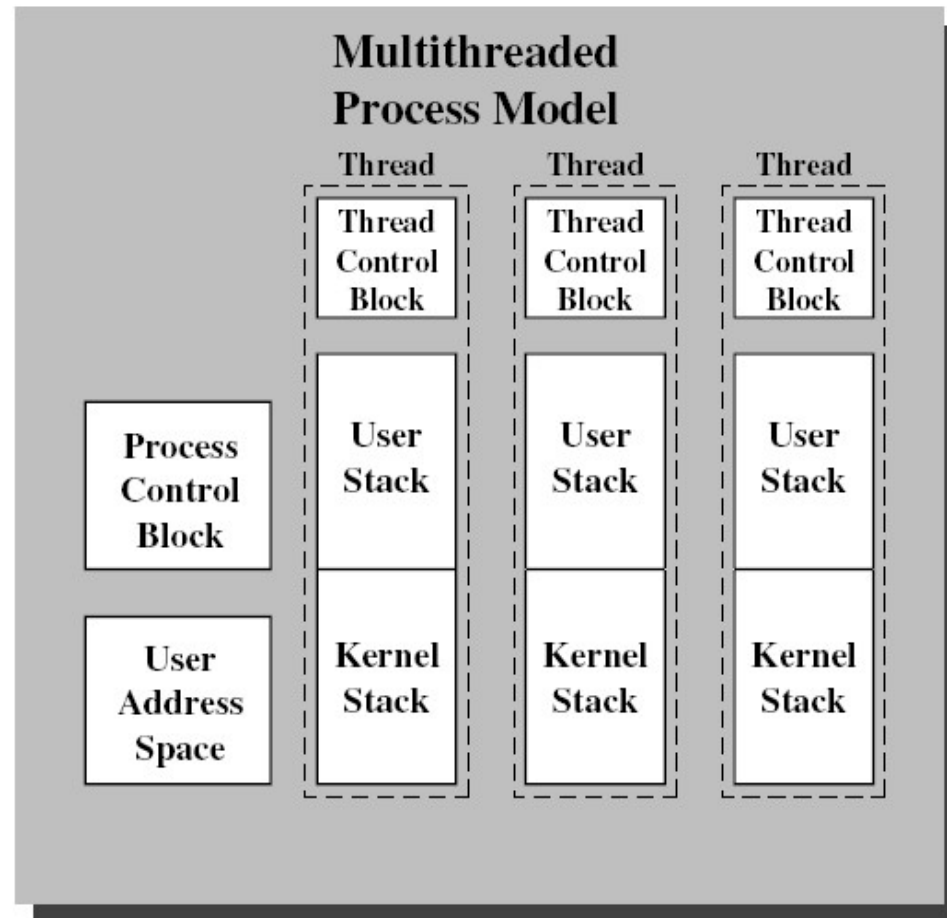


Figure 4.2 Single Threaded and Multithreaded Process Models

Each Thread Has Its Own Stack

- ▶ Each thread's stack contains one frame for each procedure called but not yet returned from.
 - ▶ This frame contains the **procedure's local variables** and the **return address** to use when the procedure call has finished.
 - ▶ Each thread will generally call different procedures and thus have a different execution history.



Scheduling and Dispatching

- ▶ Scheduling and dispatching is done on thread basis, in the OS that supports threads
 - ▶ Most of the state information dealing with execution is maintained in thread-level data structures
- ▶ Some actions that affect all of the threads in a process:
 - ▶ Suspension (swapping the address space out of main memory)
 - ▶ Termination of a process (terminates all threads within the process)

These actions must be managed by OS in process-level



Thread Functionality (1)

Thread States

- ▶ **Thread States:** Running, Ready, Blocked
- ▶ Operations associated with a change in thread state
 - ▶ **Spawn (派生):** Spawn another thread
 - ▶ **Block (阻塞):** when a thread needs to wait for an event, it will block.
 - ▶ **Unblock (解除阻塞):** when a event for which a thread is blocked occurs, the thread is moved to the ready queue
 - ▶ **Finish (结束):** Deallocate register context and stacks




Thread Functionality (2)

Thread Synchronization

- ▶ **All of the threads of a process share the same address space and other resources**
 - ▶ Any alteration of a resource by one thread affects the environment of the other threads in the same process
 - ▶ Therefore, it is necessary to synchronize the activities of the various threads so that they do not interfere with each other.
- ▶ **Details will be discussed in the next two chapters.**



Organization

- ▶ **An Introduction**
- ▶ **Multithreading Models** 
- ▶ **Thread Libraries**
- ▶ **Threading Issues**



User-Level and Kernel-Level Threads

- ▶ Two broad categories of thread implementation:
 - ▶ *User-level threads* (ULTs)
 - ▶ *Kernel-level threads* (KLTs)
 - ▶ Also called as kernel-supported threads or lightweight processes.

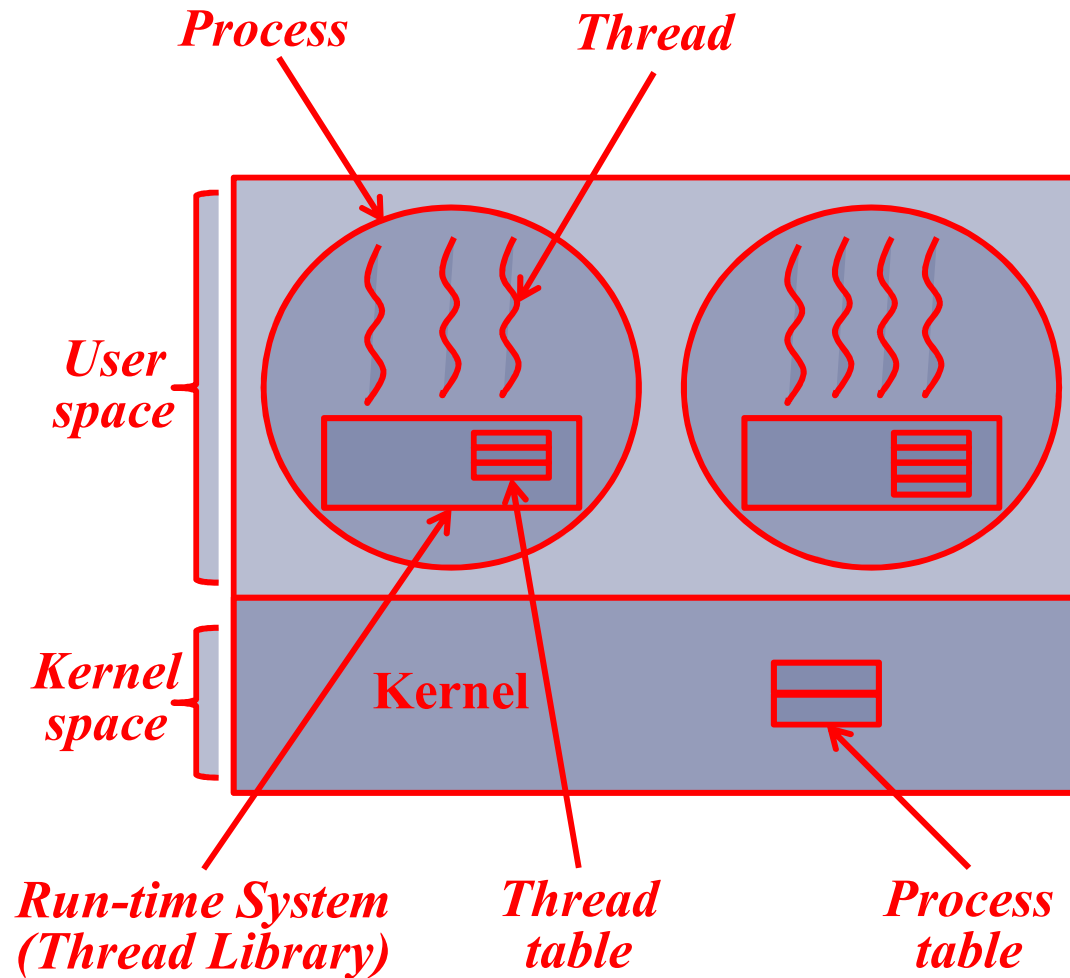
A hybrid implementation is also available



User-Level Threads

- ▶ **Pure ULT:** All thread management is done by the application
 - ▶ The kernel is not aware of the existence of threads
- ▶ **Threads library:** 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
 - ▶ For **saving and restoring thread context**





Each process has its own private thread table to keep track of the threads in that process:

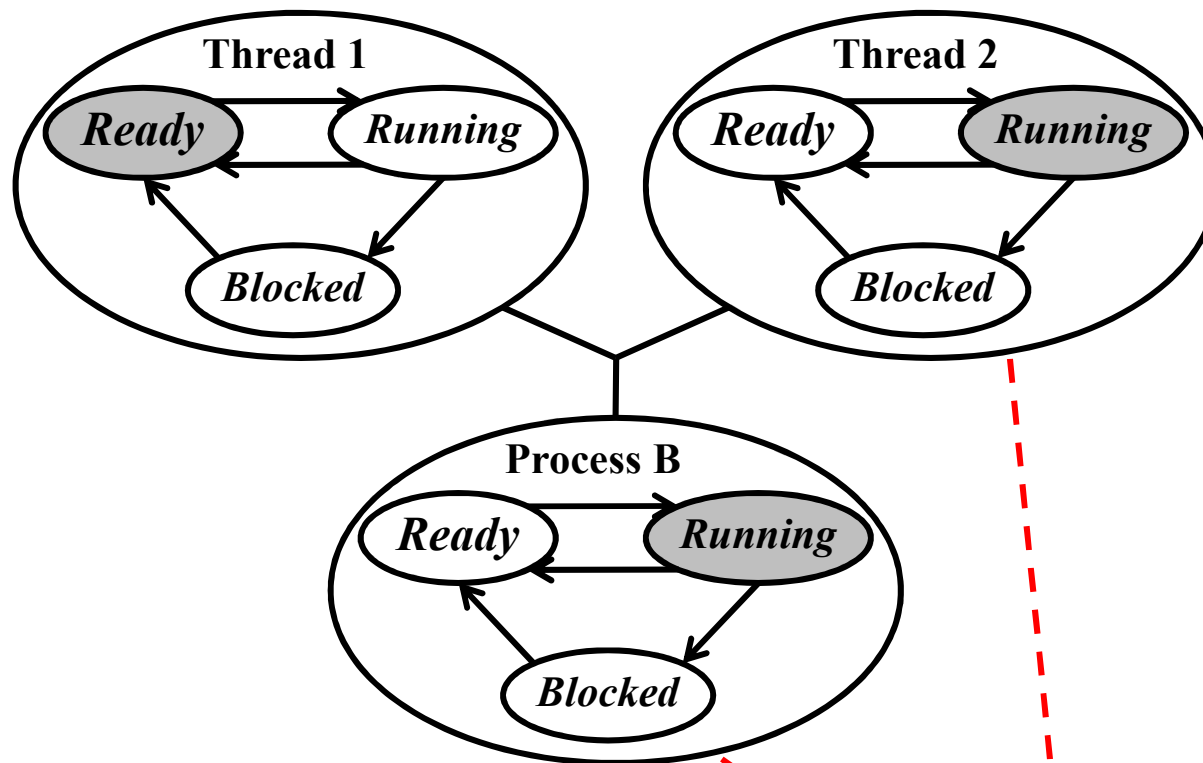
It keeps track of the per-thread properties: each thread's

► *program counter, stack pointer, registers, state,*

The Relationships between User-Level Thread States and Process States

- ▶ **An example: there are two user-level threads in process B.**
 - ▶ Initially, process B is executing in its thread 2.



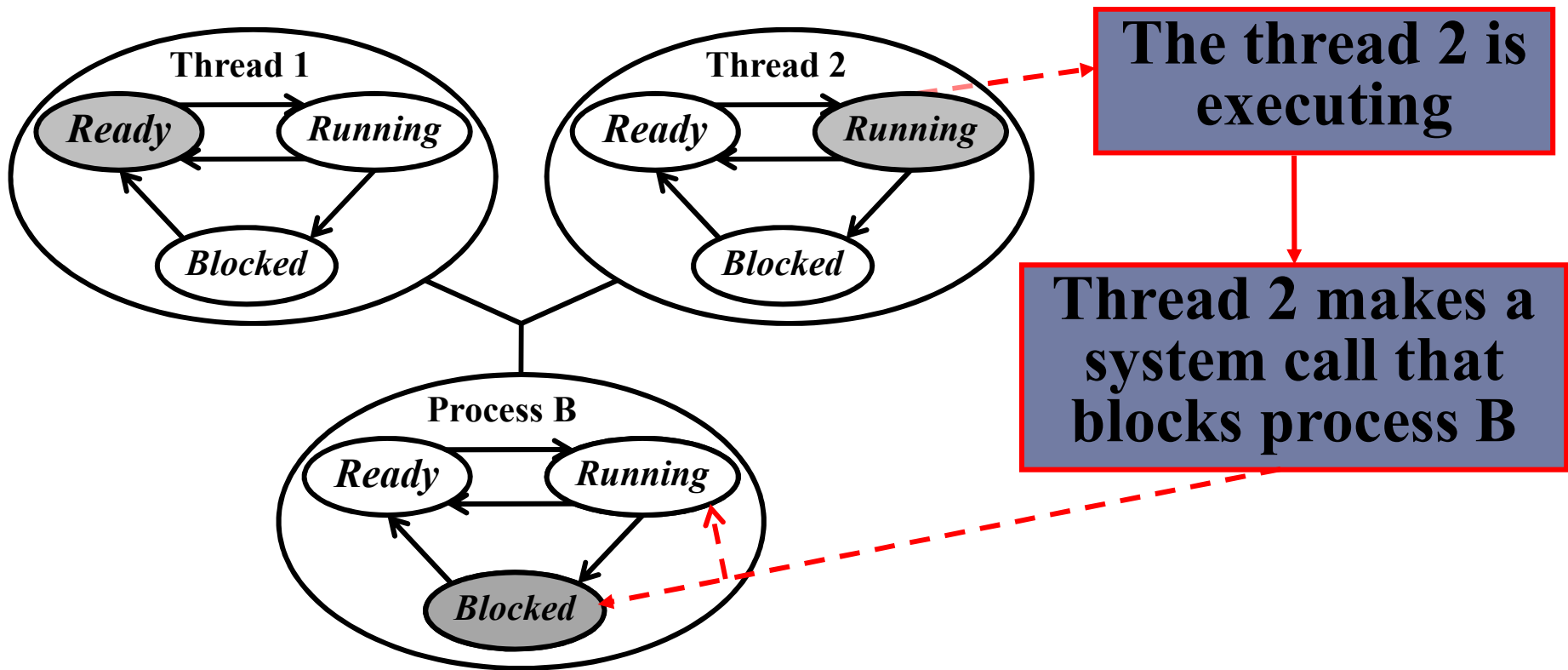


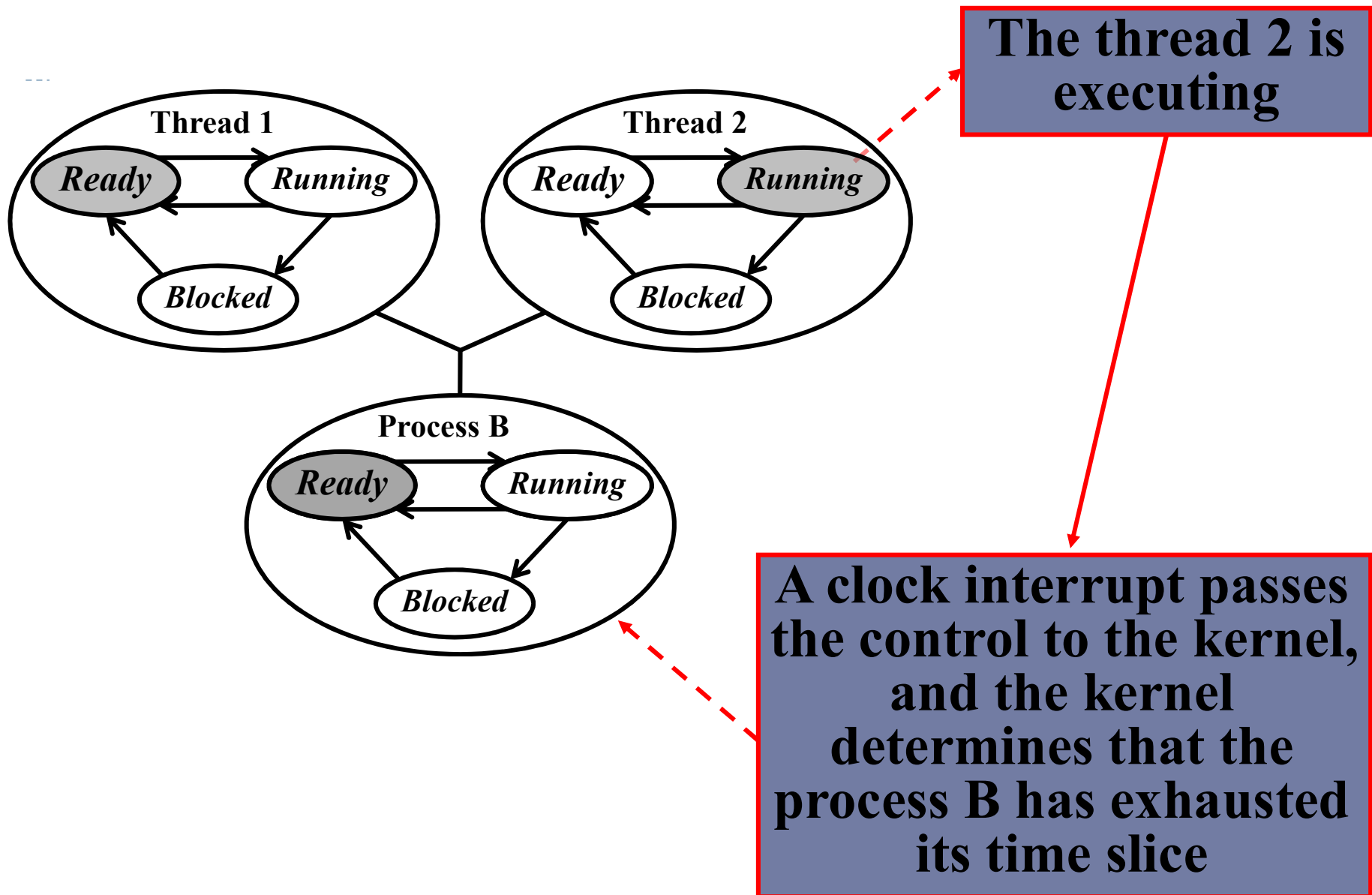
Process B is executing in its thread 2

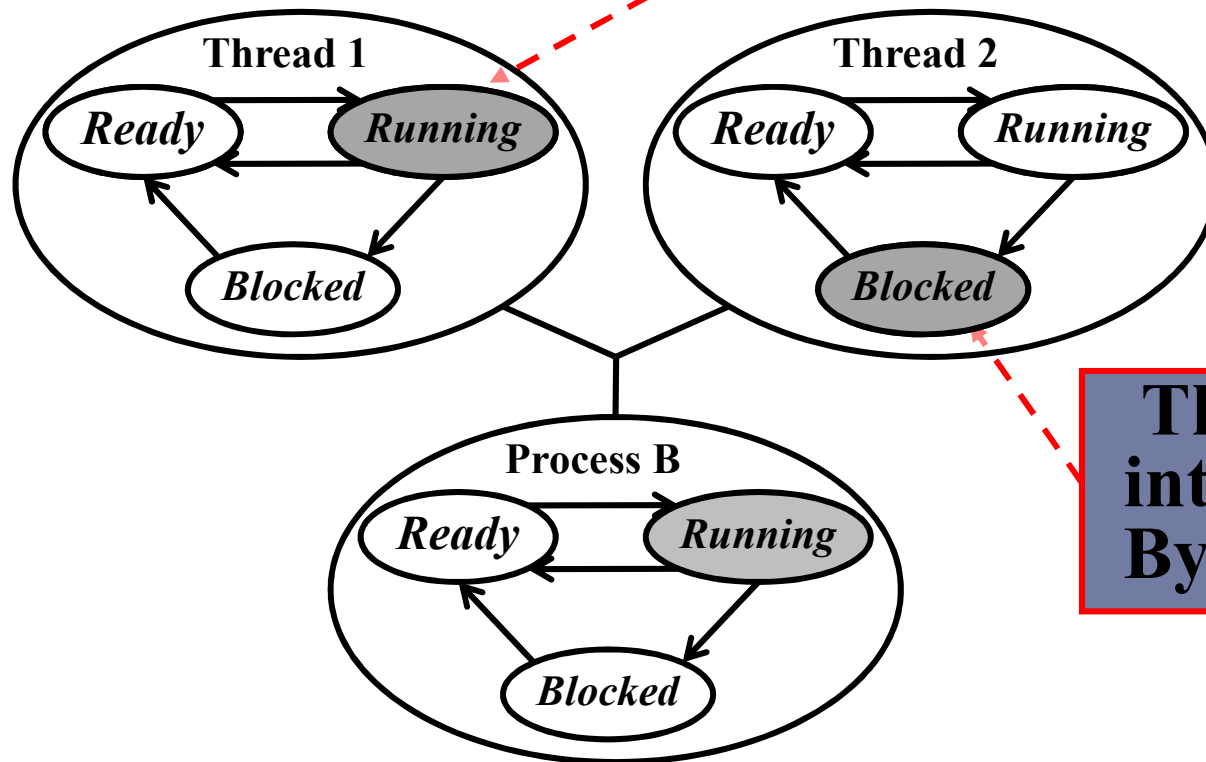
Problems

- ▶ ***What happens for the following situations respectively?***
 - ▶ *Thread 2 makes a blocking system call*
 - ▶ *A clock interrupt passes control to the kernel and the kernel determines the currently-running process has exhausted its time slice.*
 - ▶ *Thread 2 has reached a point where it needs some action performed by thread 1 of process B.*









Thread 1 goes from Ready to Running

The thread 2 gets into Blocked state, By threads library

The thread 2 is waiting for some action performed by thread 1.

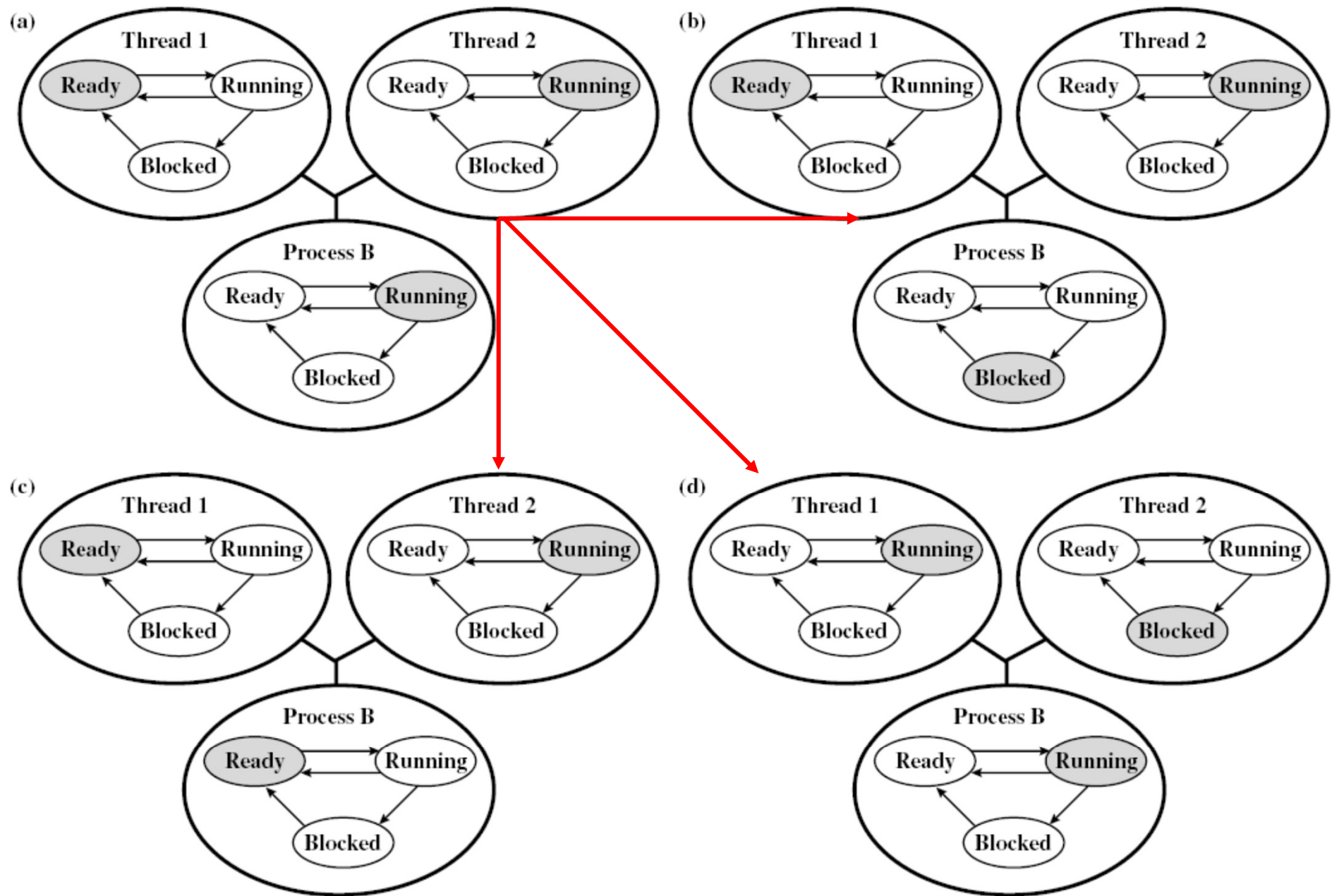


Figure 4.7 Examples of the Relationships Between User-Level Thread States and Process States

Advantages of ULTs

- ▶ ***Thread switching does not require switching to the kernel mode***
 - ▶ *Because all of the thread management data structures are within the user address space.*
- ▶ ***Scheduling can be application-specific***
- ▶ ***ULTs can run on any OS.***



Disadvantages of ULTs

- ▶ **When ULT executes a system call and gets blocked, all of the threads in the process are blocked**
 - ▶ *Many system calls are blocking, in a typical OS.*
- ▶ **The problem of page faults**
 - ▶ *when a thread causes a page fault and other threads in the process might be runnable*
- ▶ **A multithreaded application can not take advantage of multiprocessing**
 - ▶ *Only a single thread within a process can execute at a time*

The management of multiple processes within a **multiprocessor**



Blocking System Calls

- ▶ Make system calls to be nonblocking
 - ▶ For example: a read on the keyboard would just return a 0 bytes if no characters were already buffered.
- ▶ Jacketing or Wrapper: to tell in advance if a system call will block. (... next slide)



Jacketing Technique

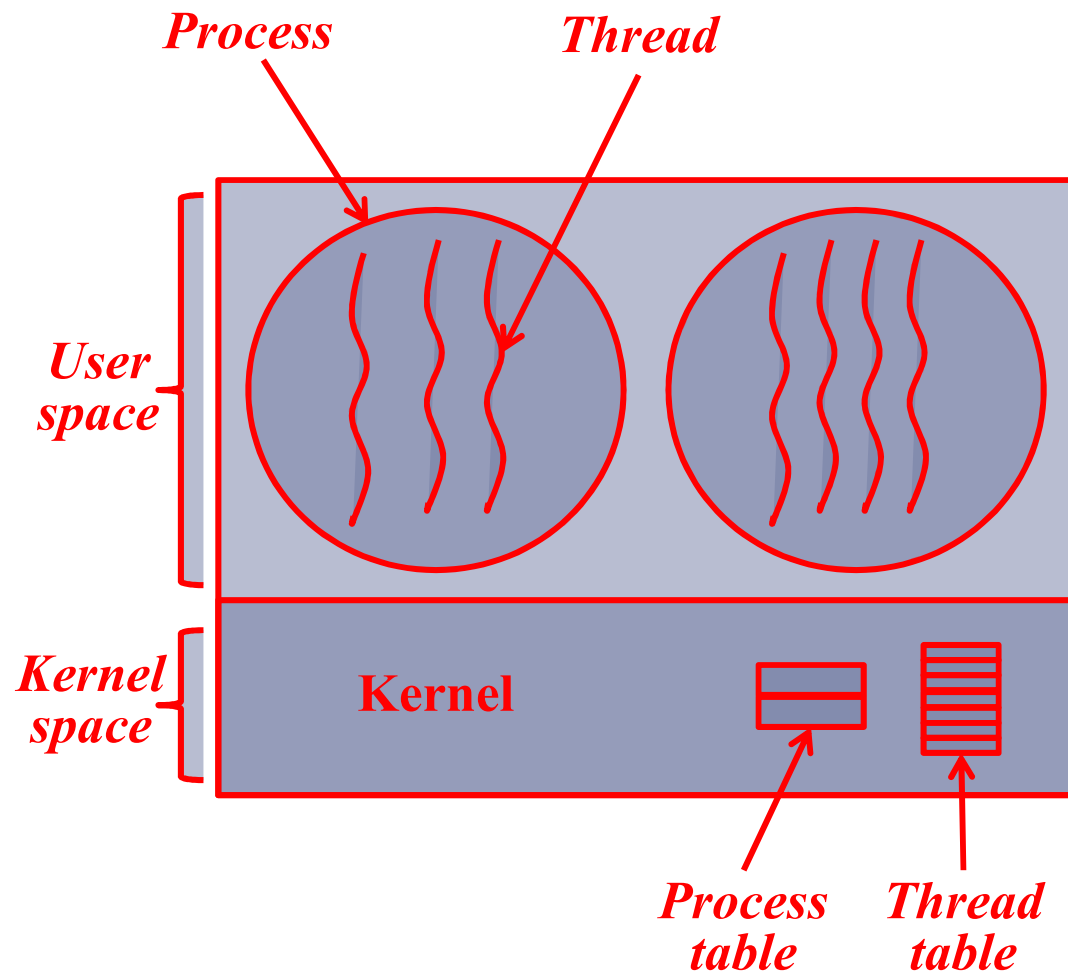
- ▶ Jacketing is one way to overcome the problem of blocking threads
- ▶ The purpose of jacketing is to convert a blocking system call into a nonblocking system call
 - ▶ A thread call an application-level I/O jacket routine, instead of calling system I/O routine
 - ▶ The jacket routine checks to determine if the I/O device is busy
 - ▶ If yes, this thread enters Ready state, and passes control to another thread
 - ▶ This thread checks the I/O device again when it is given control again.



Kernel-Level Threads

- ▶ **Pure KLT: All of the work of thread management is done by the kernel**
 - ▶ *W2K, Linux, and OS/2 are examples of this approach*
- ▶ **Kernel maintains context information for the process and the threads**
 - ▶ *There is no thread table in each process*
- ▶ **Scheduling is done on a thread basis**





Advantages of KLTs

- ▶ *Simultaneously schedule multiple threads from the same process on **multiple processors***
- ▶ *If one thread in a process is blocked, the kernel can schedule another thread of the same process*
- ▶ *The kernel routines themselves can be multithreaded*



Disadvantages of KLTs

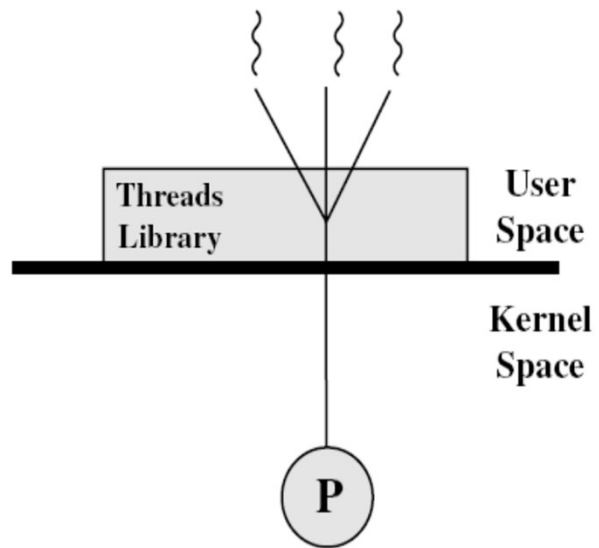
- ▶ *All calls that might block a thread are implemented as system calls*
 - ▶ *Thus have considerably **greater cost** than a call to a run-time system procedure*
- ▶ *Transfer of control from one thread to another within the same process requires a **mode switch** to the kernel.*



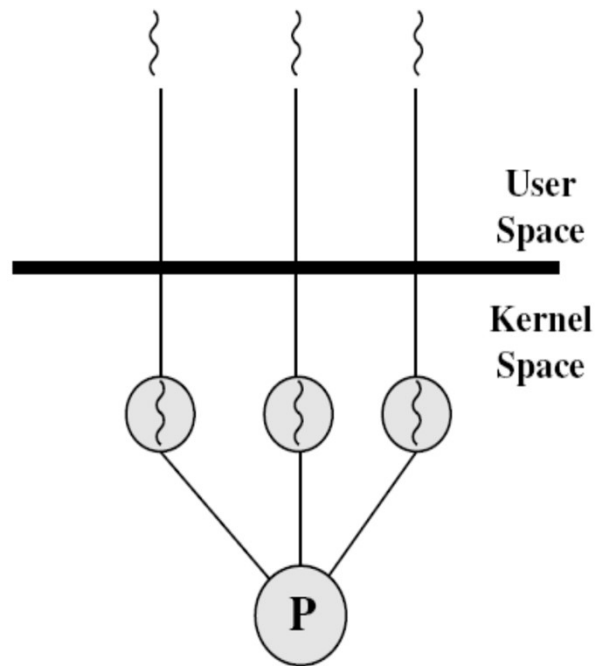
Combined Approaches

- ▶ **Combined ULT/KLT:**
 - ▶ Thread creation done in the user space
 - ▶ Bulk of scheduling and synchronization of threads done in the user space
- ▶ **Example is Solaris**

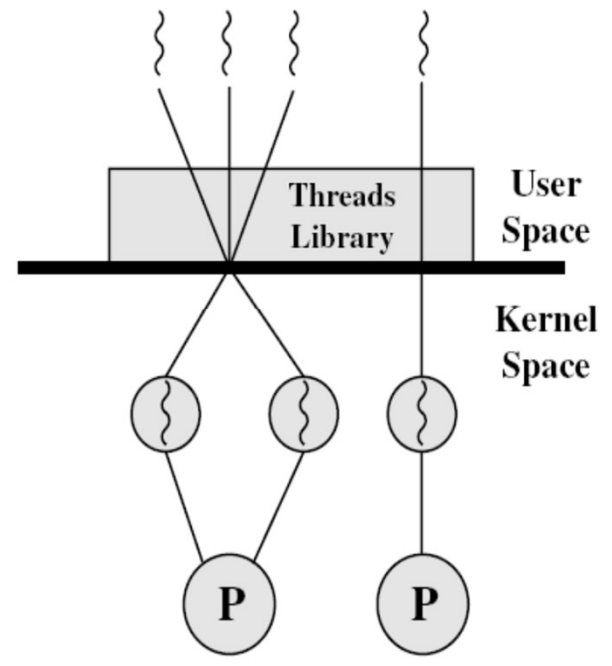




(a) Pure user-level




(b) Pure kernel-level



(c) Combined



Organization

- ▶ **An Introduction**
- ▶ **Multithreading Models**
- ▶ **Thread Libraries** 
- ▶ **Threading Issues**



Thread Library

- ▶ **Thread library: provides the programmer an API for creating and managing threads**
- ▶ **Two primary kinds of implementation:**
 - ▶ Provide a library entirely in user space with no kernel support
 - ▶ Implement a kernel-level library supported directly by the OS



Main Thread Libraries

- ▶ **Today, three main thread libraries are in use**
 - ▶ **POSIX Pthreads**
 - ▶ The threads extension of the **POSIX** standard: may be either a user- or kernel-level library
 - ▶ **Win32**
 - ▶ A kernel-level library available on **Windows** systems
 - ▶ **Java**
 - ▶ The **Java** thread **API** is typically implemented using a thread library available on the host system



```
int sum;
void *runner(void *param)

int main(int argc, char *argv[])
{
    pthread_t tid; /*the thread identifier
pthread_attr_t attr; /*set of thread attributes
if ( argc!=2 || atoi(argv[1])<0 ){
    //display error message and return;
}
pthread_attr_init(&attr); //get the default attributes
pthread_create(&tid, &attr, runner, argv[1]); //create the thread
pthread_join(tid, NULL); //wait for the thread to exit
printf("sum = %d\n", sum);
}

void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;
    for (i=0; i<=upper; i++) sum+=i;
    pthread_exit(0);
}
```

Pthreads: An Example

```

DWORD Sum; //this data is shared by the thread(s)
DWORD WINAPI Summation(LPVOID Param){
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i=0; i<=upper; i++) Sum+=i;
    return 0;
}

int main(int argc, char *argv[]){
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;
    if ( argc!=2 || atoi(argv[1])<0 ){
        //display error message and return;
    }
    Param = atoi(argv[1]);
    ThreadHandle = CreateThread(
        NULL, //default security attributes
        0, //default stack size
        Summation, //thread function
        &Param, //parameter to thread function;
        0, //default creation flags
        &ThreadId); //return the thread identifier
    if (ThreadHandle!=NULL){
        //wait for the thread to finish
        WaitForSingleObject(ThreadHandle, INFINITE);
        CloseHandle(ThreadHandle);
        printf("sum = %d\n", Sum);
    }
}

```

Win32 Threads: An Example

Organization

- ▶ **An Introduction**
- ▶ **Multithreading Models**
- ▶ **Thread Libraries**
- ▶ **Threading Issues** 



Threading Issues

- ▶ **The `fork()` system call**
- ▶ **Cancellation**
- ▶ **Thread Pools**
- ▶ **Scheduler Activation**



The fork () System Call

- ▶ **Two choices:**
 - ▶ Does the new process duplicate all threads?
 - ▶ Or is the new process duplicate only the thread that invoked fork () system call?
- ▶ **The decision depends on the application**
 - ▶ Whether the exec () is called immediately after forking



Cancellation

- ▶ **Two different scenarios:**
 - ▶ **Asynchronous cancellation** - One thread immediately terminates the target thread
 - ▶ **Deferred cancellation** - The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion



Thread Pools

- ▶ **Basic idea of thread pool:**
 - ▶ To create a number of threads at process startup and place them into a pool, where the threads sit and wait for work
 - ▶ When a server receives a request, it awakens a thread from the pool and passes it the request to service
 - ▶ Once the thread complete its service, it returns to the pool and awaits more work
- ▶ **Benefits:**
 - ▶ Servicing a request with an existing thread is usually faster than waiting to create a thread
 - ▶ A thread pool limits the number of threads that exist at any one point



Scheduler Activation (1)

- ▶ *Many-to-Many Model:*
 - ▶ *The kernel provides an application with a set of virtual processors (LWPs), and*
 - ▶ *The application can schedule user threads onto an available virtual processors*



Scheduler Activation (2)

Upcall

- ▶ *Scheduler activation: for communication between the user-thread library and the kernel*
 - ▶ *The kernel provides an application with a set of LWPs (or virtual processors)*
 - ▶ *The application can schedule user threads onto an available virtual processor.*
 - ▶ *The kernel must inform an application about certain events, which is called an upcall.*
 - ▶ *The thread library handles upcalls with an upcall handler*



Scheduler Activation (2)

- ▶ *The steps:*

- ▶ *One event that triggers an upcall occurs **when an application thread is about to block**. Then the kernel makes an upcall to the application*
- ▶ *The kernel then allocate a new virtual processor to the application. The application runs an upcall handler on it.*
- ▶ *The upcall handler saves the state of the blocking thread and relinquishes the virtual processor of the blocking thread*
- ▶ *The upcall handler then schedules another thread that is eligible to run on the virtual processor.*
- ▶ *When the event that the blocking thread was waiting for occurs, the kernel makes another upcall to the thread library, informing it that the previously blocked thread is now eligible to run.*

