Operating Systems: Internals and Design Principles

Chapter 4 Threads

Ninth Edition

By William Stallings

Processes and Threads

Resource Ownership

Process includes a virtual address space to hold the process image

- The OS performs a protection function to prevent unwanted interference between processes with respect to resources

Scheduling/Execution

Follows an execution path (trace) that may be interleaved with other processes

- A process has an execution state (Running, Ready, etc.) and a dispatching priority, and is the entity that is scheduled and dispatched by the OS

Processes and Threads

- The two characteristics (resource ownership and scheduling/execution) are independent and could be treated independently by the OS
- To distinguish the two characteristics, 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***
- ***Multithreading*** - The ability of an OS to support multiple, concurrent paths of execution within a single process

Single Threaded Approaches

A single thread of execution per process, in which the concept of a thread is not recognized, is referred to as a **single-threaded** approach

- MS-DOS: a single-user process and a single thread
- Some UNIX variants: multiple user processes but one thread per process

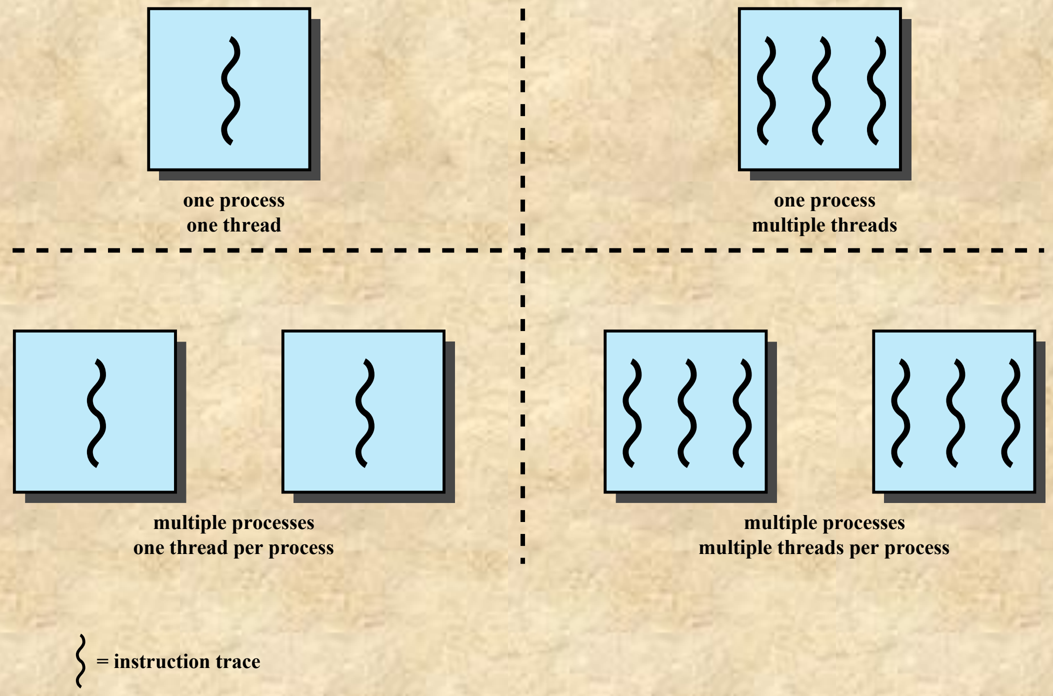


Figure 4.1 Threads and Processes

Multithreaded Approaches

The right half of Figure 4.1 depicts **multithreaded** approaches

- A Java runtime environment is an example of a system of one process with multiple threads
- Multiple processes, each of which supports multiple threads: Windows, Solaris, and many modern versions of UNIX

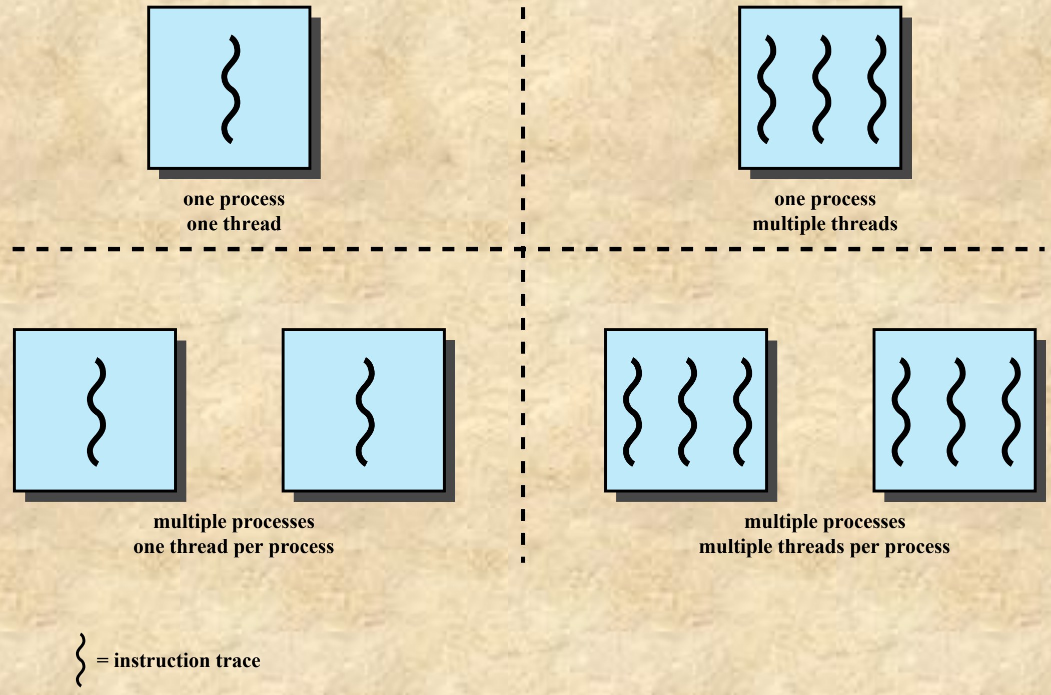


Figure 4.1 Threads and Processes

Process

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)

One or More Threads in a Process

Each thread has:

- An execution state (Running, Ready, etc.)
- A 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 processes, shared with all other threads in that process

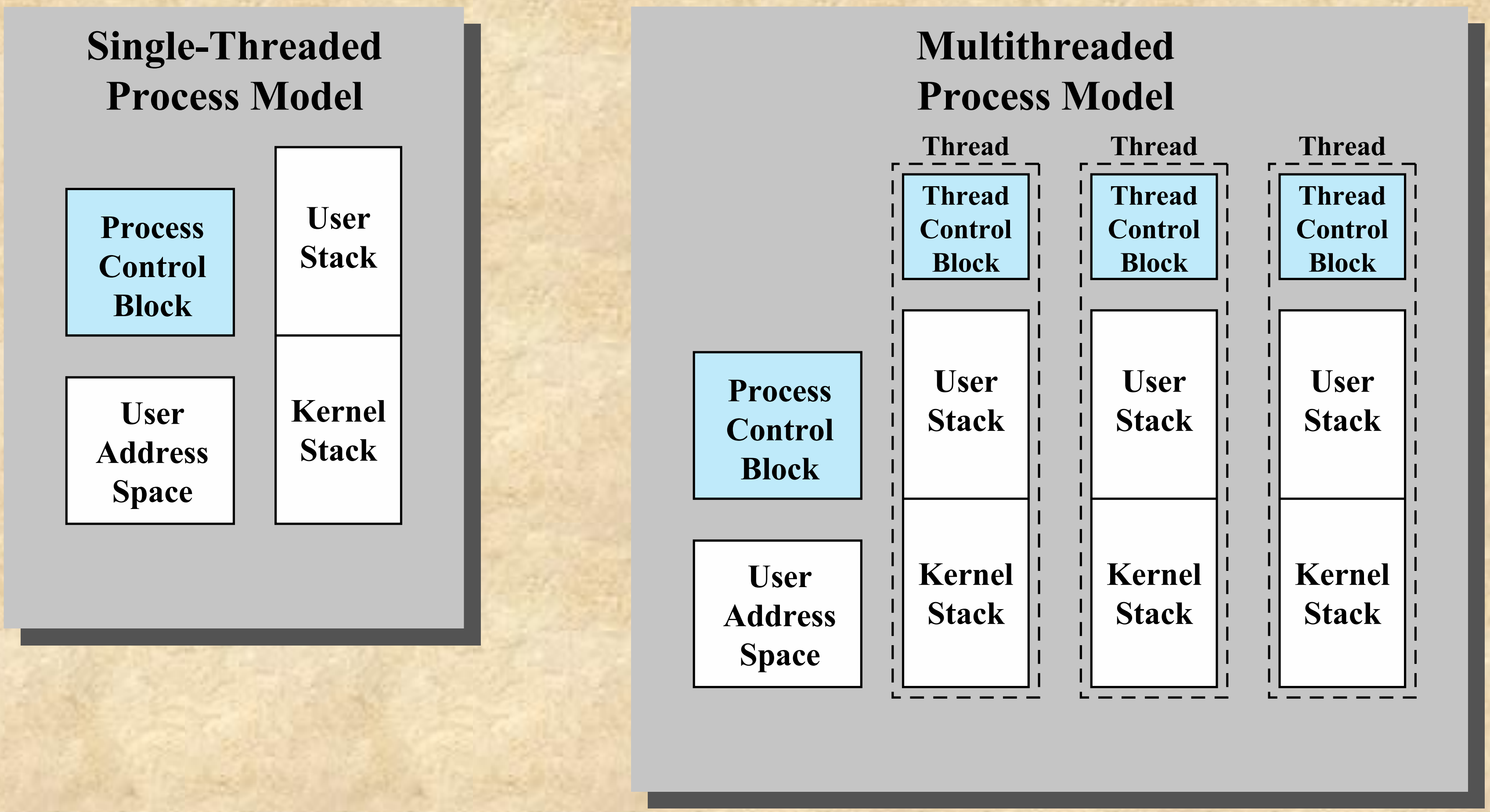


Figure 4.2 Single Threaded and Multithreaded Process Models

The **thread control block** contains register values, priority, and other thread-related state information

One or More Threads in a Process

- 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 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. All 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

Key Benefits of Threads

1. It takes far less time to create a new thread in an existing process, than to create a brand-new process
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**

Thread Use in a Single-User System

- **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

- **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

Thread Use in a Single-User System

- **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
- **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

Application Benefits of Threads

- Consider an application that consists of several independent parts that do not need to run in sequence
- Each part can be implemented as a thread
- Whenever one thread is blocked waiting for I/O, execution could switch to another thread of the same application (instead of switching to another process)

Example of File Server

File Server on a LAN

- Needs to handle many file requests over a short period
- Threads can be created (and later destroyed) for each request
- If multiple processors: different threads could execute simultaneously on different processors

Threads

- In an OS that supports threads, scheduling and dispatching is done on a thread basis
- Most of the state information dealing with execution is maintained in thread-level data structures, but
    - Suspending a process involves suspending all threads of the process since all threads in a process share the same address space
    - Termination of a process terminates all threads within the process

Thread Execution States

The key states for a thread are:

- Running
- Ready
- Blocked

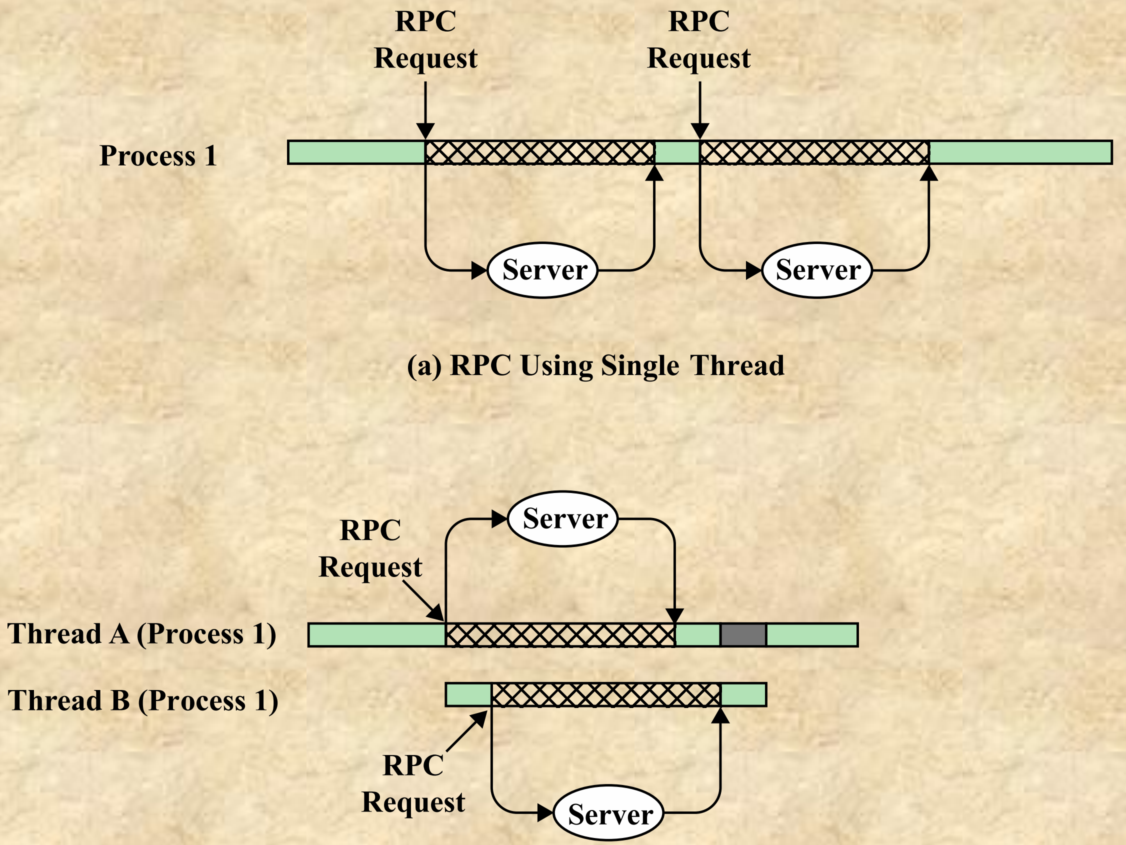
Generally, it does not make sense to associate suspend states with threads because such states are process-level concepts. If a process is swapped out, all of its threads are necessarily swapped out because they all share the address space of the process

Thread Execution States

Four basic thread operations associated with a change in thread state

- **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.** The 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 block (saving its user registers, program counter, and stack pointers). The processor may now turn to the execution of another ready thread in the same or a different process
- **Unblock:** When the event for which a thread is blocked occurs, the thread is moved to the Ready queue
- **Finish:** When a thread completes, its register context and stacks are deallocated

Figure 4.3 (based on one in shows a program that performs two remote procedure calls (RPCs) to two different hosts to obtain a combined result

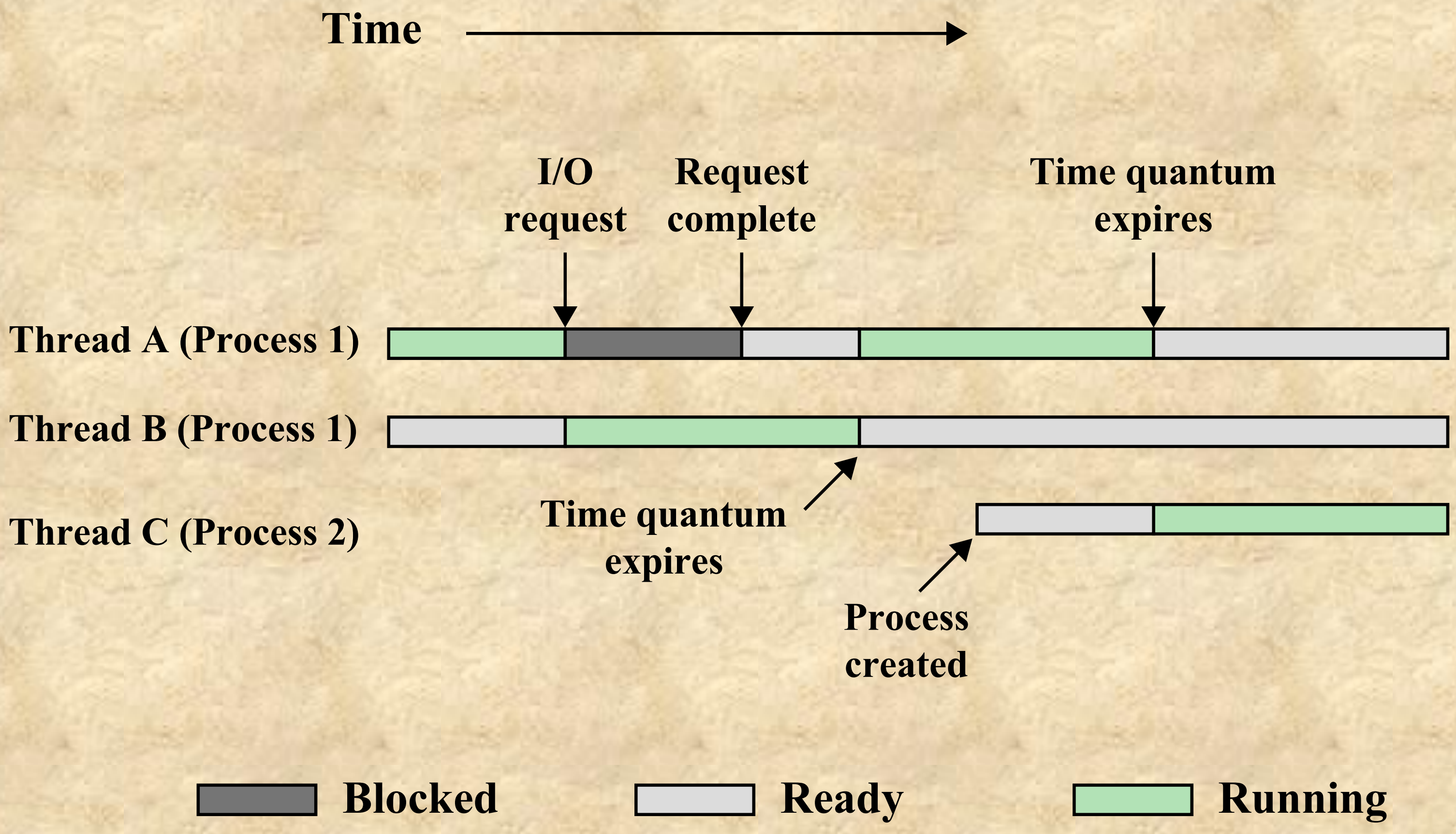


(a) RPC Using Single Thread

(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 4.3 Remote Procedure Call (RPC) Using Threads



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

Figure 4.4 Multithreading Example on a Uniprocessor

Thread Synchronization

Recall that

- All 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 other threads in the same process

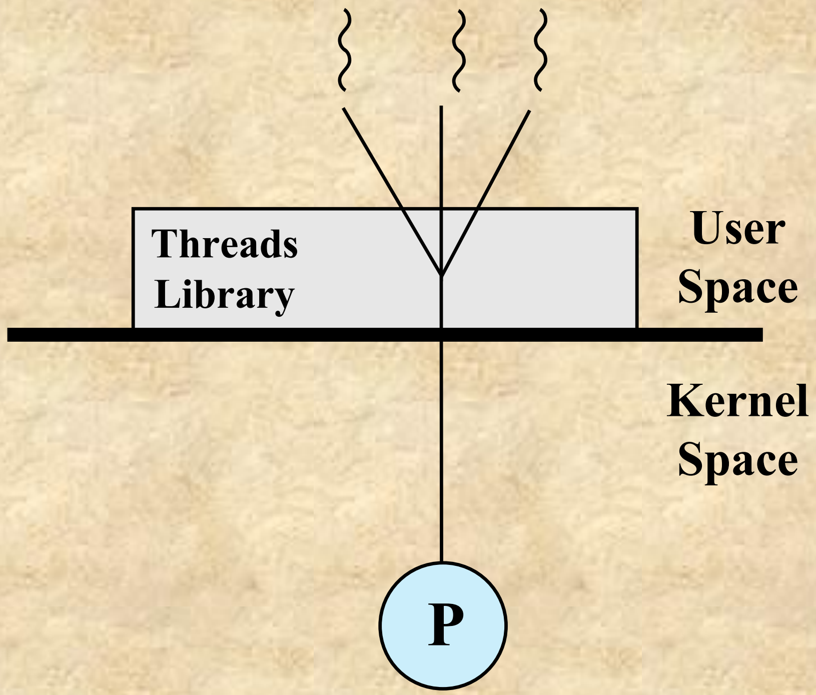
It is therefore necessary to synchronize the activities of the various threads. For example, if two threads each try to add an element to a doubly linked list at the same time, one element may be lost, or the list may end up malformed

Types of Threads

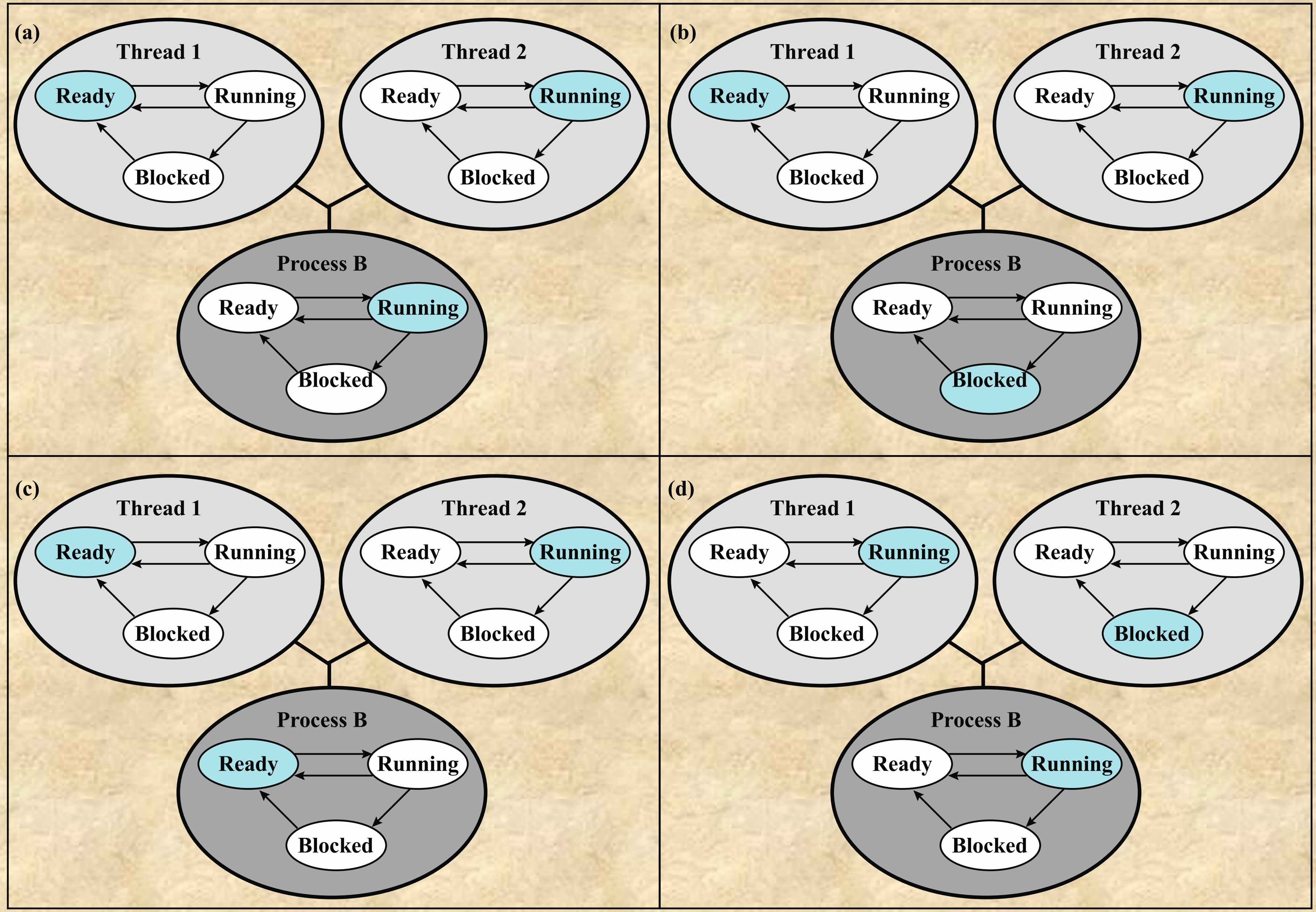
- There are two broad categories of thread implementation: **user-level threads (ULTs)** and **kernel-level threads (KLTs)**
- KLTs are also referred to in the literature as *kernel-supported threads or lightweight processes*

User-Level Threads (ULTs)

- In a pure ULT facility, all the work of thread management is done by the application, and the kernel is not aware of the existence of threads
- Any application can be programmed to be multithreaded by using a threads library. 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



(a) Pure user-level



Colored state
is current state

Figure 4.6 Examples of the Relationships Between User-Level Thread States and Process States

Transition from Figure 4.6(a) to Figure 4.6(b)

- 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 switches to another process. Meanwhile, according to the data structure maintained by the threads library, thread 2 of process B is still in the Running state
- It is important to note that thread 2 is not actually running in the sense of being executed on a processor; but it is perceived as being in the Running state by the threads library. Why?

Transition from Figure 4.6(a) to Figure 4.6(c)

- A **clock interrupt** passes control to the kernel and the kernel determines that the currently running process (B) has exhausted its time slice. The kernel places process B in the Ready state and switches to another process. Meanwhile, according to the data structure maintained by the threads library, thread 2 of process B is still in the Running state

Transition from Figure 4.6(a) to Figure 4.6(d)

- Thread 2 has reached a point where it needs some action performed by thread 1 of process B. Thread 2 enters a Blocked state and thread 1 transitions from Ready to Running. The process itself remains in the Running state

Advantages of ULTs

There are a number of advantages to the use of ULTs instead of KLTs, including the following:

1. **Thread switching does not require kernel mode privileges** because all of the thread management data structures are within the user address space of a single process. Therefore, the process does not switch to the kernel mode to do thread management. This saves the overhead of two mode switches (user to kernel; kernel back to user)

2. **Scheduling can be application specific**. One application may benefit most from a simple round-robin scheduling algorithm, while another might benefit from a priority-based scheduling algorithm. The scheduling algorithm can be tailored to the application without disturbing the underlying OS scheduler

3. **ULTs can run on any OS**. No changes are required to the underlying kernel to support ULTs. The threads library is a set of application-level functions shared by all applications

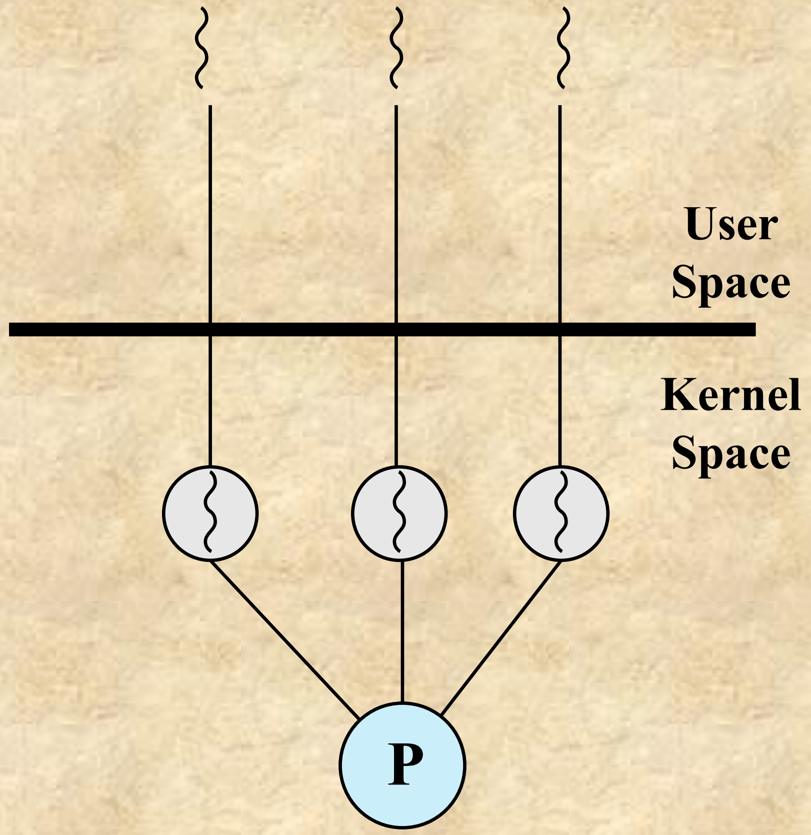
Disadvantages of ULTs

- In a typical OS many system calls are blocking
  - As a result, when a ULT executes a system call, not only is that thread blocked, but all of the threads within the process are blocked as well
- In a pure ULT strategy, a multithreaded application cannot take advantage of multiprocessing
  - A kernel assigns one process to only one processor at a time, therefore, only a single thread within a process can execute at a time

Overcoming ULT Disadvantages

- Approach 1: Writing an application as multiple processes rather than multiple threads
    - However, this approach eliminates the main advantage of threads
- Approach 2: Jacketing - convert a blocking system call into a non-blocking system call
    - For example, instead of directly calling a system I/O routine, a thread calls an application-level I/O jacket routine. Within this jacket routine is code that checks to determine if the I/O device is busy. If it is, the thread enters the Blocked state and passes control to another thread. When this thread later is given control again, the jacket routine checks the I/O device again

Kernel-Level Threads (KLTs)



(b) Pure kernel-level

Thread management is done by the kernel

- There is no thread management code in the application level, simply an application programming interface (API) to the kernel thread facility
- Windows is an example of this approach

Advantages of KLTs

- The kernel can 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
- Kernel routines themselves can be multithreaded

Disadvantage of KLTs

The transfer of control from one thread to another within the same process requires a mode switch to the kernel

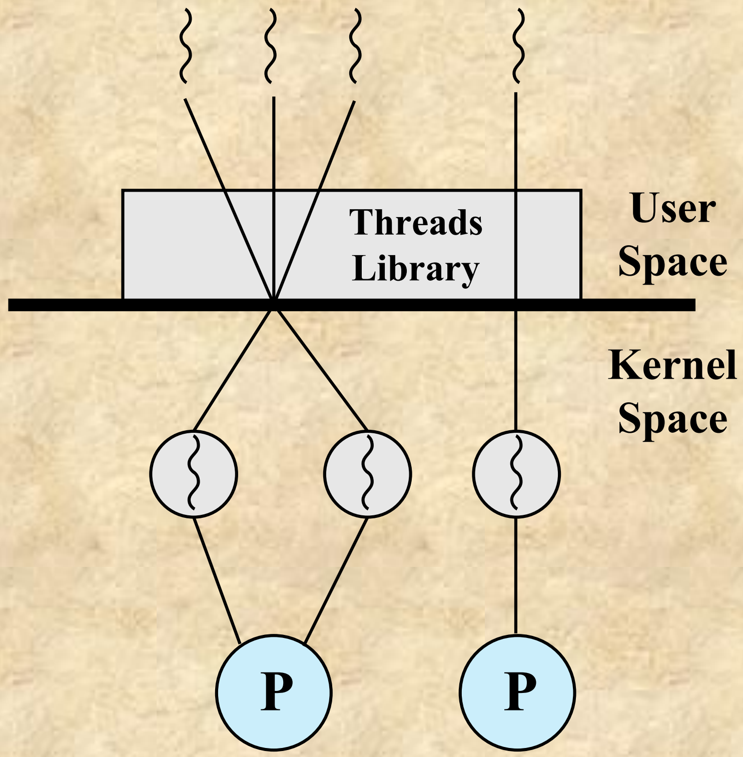
Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

Table 4.1
Thread and Process Operation Latencies (µs)

Null Fork measures the overhead of forking a process/thread
Signal-Wait measures the overhead of synchronizing two processes/threads together

Combined Approaches

- Thread creation is done completely in the user space, as is the bulk of the scheduling and synchronization of threads within an application
- The multiple ULTs from a single application are mapped onto some (smaller or equal) number of KLTs
- In a combined approach, multiple threads within the same application can run in parallel on multiple processors, and a blocking system call need not block the entire process. If properly designed, this approach should combine the advantages of the pure ULT and KLT approaches while minimizing the disadvantages.
- Solaris is a good example using the combined approach (one to one)



(c) Combined

Threads:Processes	Description	Example Systems
1:1	Each thread of execution is a unique process with its own address space and resources.	Traditional UNIX implementations
M:1	A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.	Windows NT, Solaris, Linux, OS/2, OS/390, MACH
1:M	A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.	Ra (Clouds), Emerald
M:N	Combines attributes of M:1 and 1:M cases.	TRIX

Table 4.2
Relationship between Threads and Processes

Applications That Benefit from Multicore

- Multithreaded native applications
  - Characterized by having a small number of highly threaded processes
- Multiprocess applications
  - Characterized by the presence of many single-threaded processes
- Java applications
  - All applications that use a Java 2 Platform, Enterprise Edition application server can immediately benefit from multicore technology
- Multi-instance applications
  - Multiple instances of the application in parallel

Windows Process and Thread Management

- An **application** consists of one or more processes
- Each **process** provides the resources needed to execute a program
- A **thread** is the entity within a process that can be scheduled for execution
- A **job object** allows groups of processes to be managed as a unit
- A **thread pool** is a collection of worker threads that efficiently execute asynchronous callbacks on behalf of the application
- A **fiber** is a unit of execution that must be manually scheduled by the application
- User-mode scheduling (UMS) is a lightweight mechanism that applications can use to schedule their own threads

Management of Background Tasks and Application Lifecycles

- Beginning with Windows 8, and carrying through to Windows 10, developers are responsible for managing the state of their individual applications
- Previous versions of Windows always give the user full control of the lifetime of a process
- In the new **Metro interface**, Windows takes over the process lifecycle of an application
  - A limited number of applications can run alongside the main app in the Metro UI using the SnapView functionality
  - Only one Store application can run at one time
- Live Tiles give the appearance of applications constantly running on the system
  - In reality they receive push notifications and do not use system resources to display the dynamic content offered

Metro Interface

- Foreground application in the Metro interface has access to all of the processor, network, and disk resources available to the user
  - All other apps are suspended and have no access to these resources
- When an app enters a suspended mode, an event should be triggered to store the state of the user's information
  - This is the responsibility of the application developer
- Windows may terminate a background app
  - You need to save your app's state when it's suspended, in case Windows terminates it so that you can restore its state later
  - When the app returns to the foreground another event is triggered to obtain the user state from memory

Windows Process

Important characteristics of Windows processes are the following:

- Windows processes are implemented as objects
- A process can be created as a new process or as a copy of an existing process
- An executable process may contain one or more threads
- Both process and thread objects have built-in synchronization capabilities

Process and Thread Objects

Windows makes use of two types of process-related objects:

Processes	Threads
• An entity corresponding to a user job or application that owns resources	• A dispatchable unit of work that executes sequentially and is interruptible

Multithreading in Windows

A multithreaded process achieves concurrency without the overhead of using multiple processes

- Threads within the same process can exchange information through their common address space and have access to the shared resources of the process
- Threads in different processes can exchange information through shared memory that has been set up between the two processes

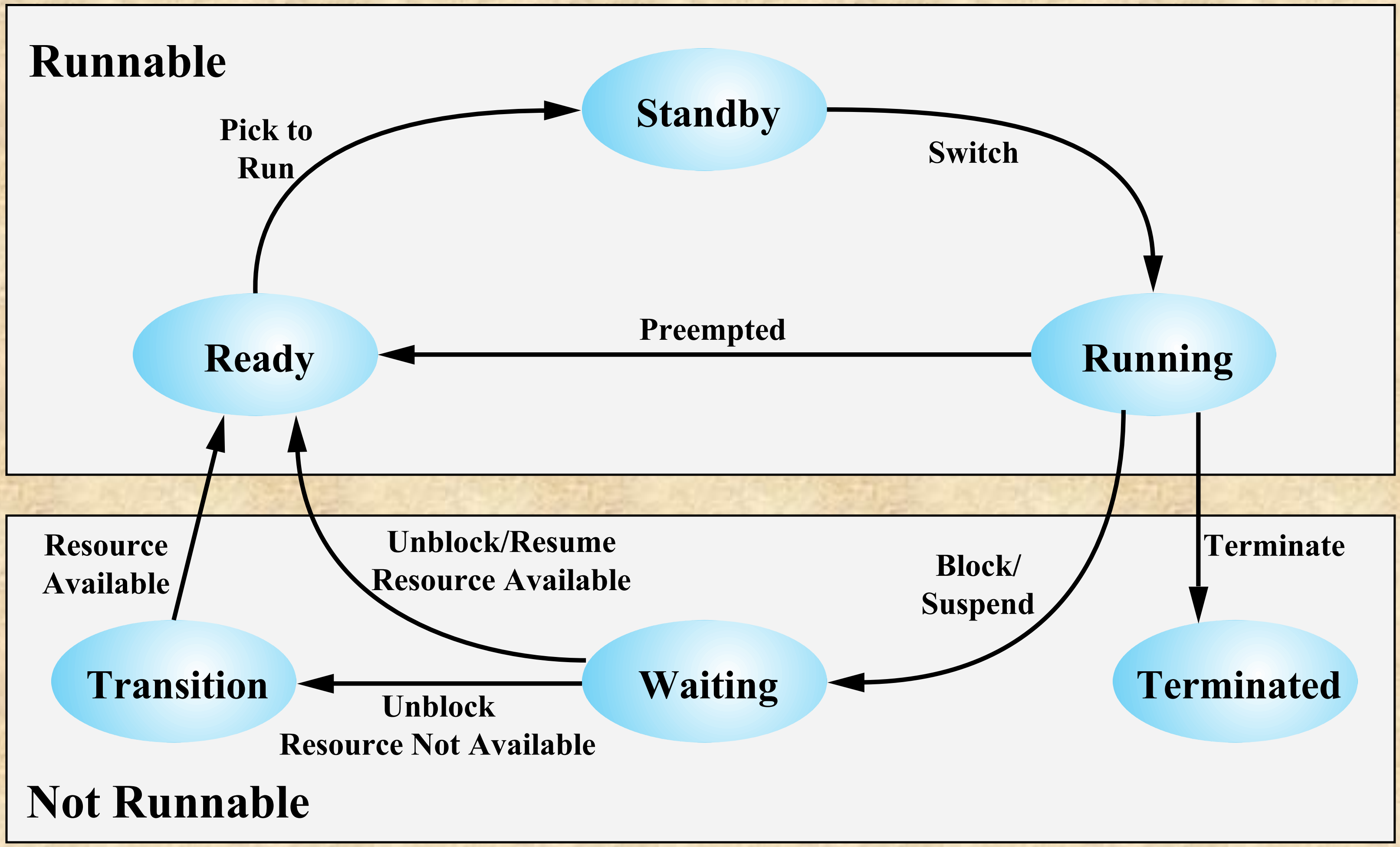


Figure 4.11 Windows Thread States

Thread States in Windows

Ready: A ready thread may be scheduled for execution. The Kernel dispatcher keeps track of all ready threads and schedules them in priority order

Standby: A standby thread has been selected to run next on a particular processor. The thread waits in this state until that processor is made available. If the standby thread's priority is high enough, the running thread on that processor may be preempted in favor of the standby thread. Otherwise, the standby thread waits until the running thread blocks or exhausts its time slice

Running: Once the Kernel dispatcher performs a thread switch, the standby thread enters the Running state and begins execution and continues execution until it is preempted by a higher-priority thread, exhausts its time slice, blocks, or terminates. In the first two cases, it goes back to the Ready state

Thread States in Windows

Waiting: A thread enters the Waiting state when (1) it is blocked on an event (e.g., I/O), (2) it voluntarily waits for synchronization purposes, or (3) an environment subsystem directs the thread to suspend itself. When the waiting condition is satisfied, the thread moves to the Ready state if all of its resources are available.

Transition: A thread enters this state after waiting if it is ready to run, but the resources are not available. For example, the thread's stack may be paged out of memory. When the resources are available, the thread goes to the Ready state.

Terminated: A thread can be terminated by itself, by another thread, or when its parent process terminates