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 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 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 is an example

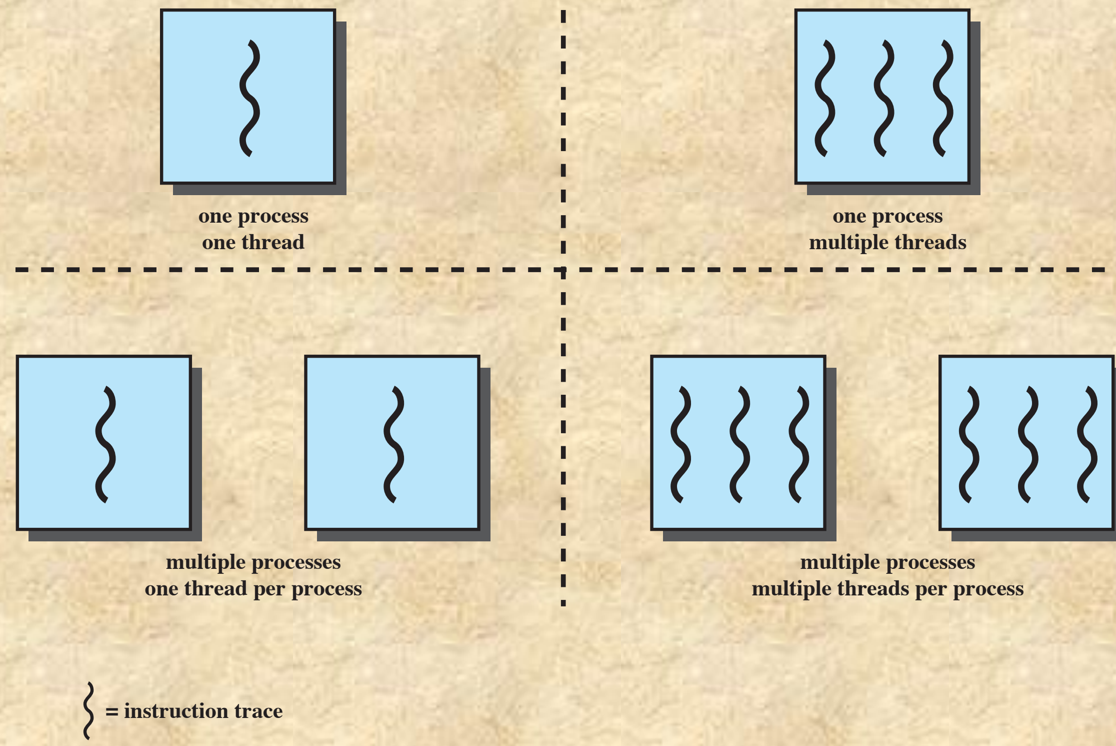


Figure 4.1 Threads and Processes

Multithreaded Approaches

- The right half of Figure 4.1 depicts multithreaded approaches
- A Java run-time environment is an example of a system of one process with multiple threads

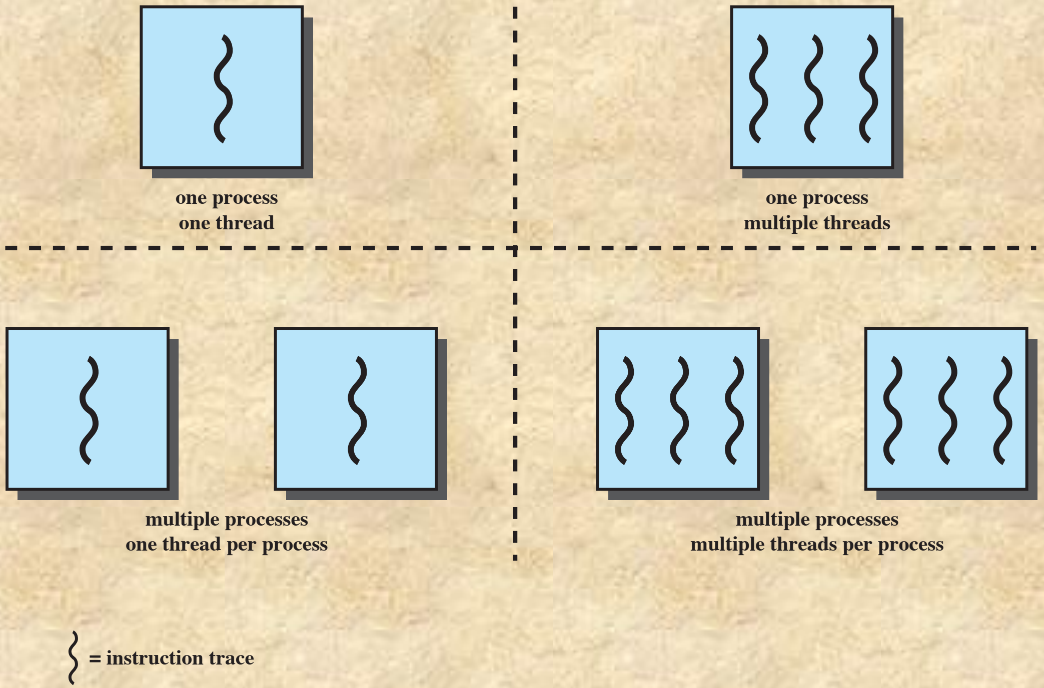


Figure 4.1 Threads and Processes

Process

- Defined in a multithreaded environment as "the unit of resource allocation and a unit of protection"
- Associated with processes:
  - A virtual address space that holds the process image
  - Protected access to:
    - Processors
    - Other processes (for interprocess communication)
    - Files
    - 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

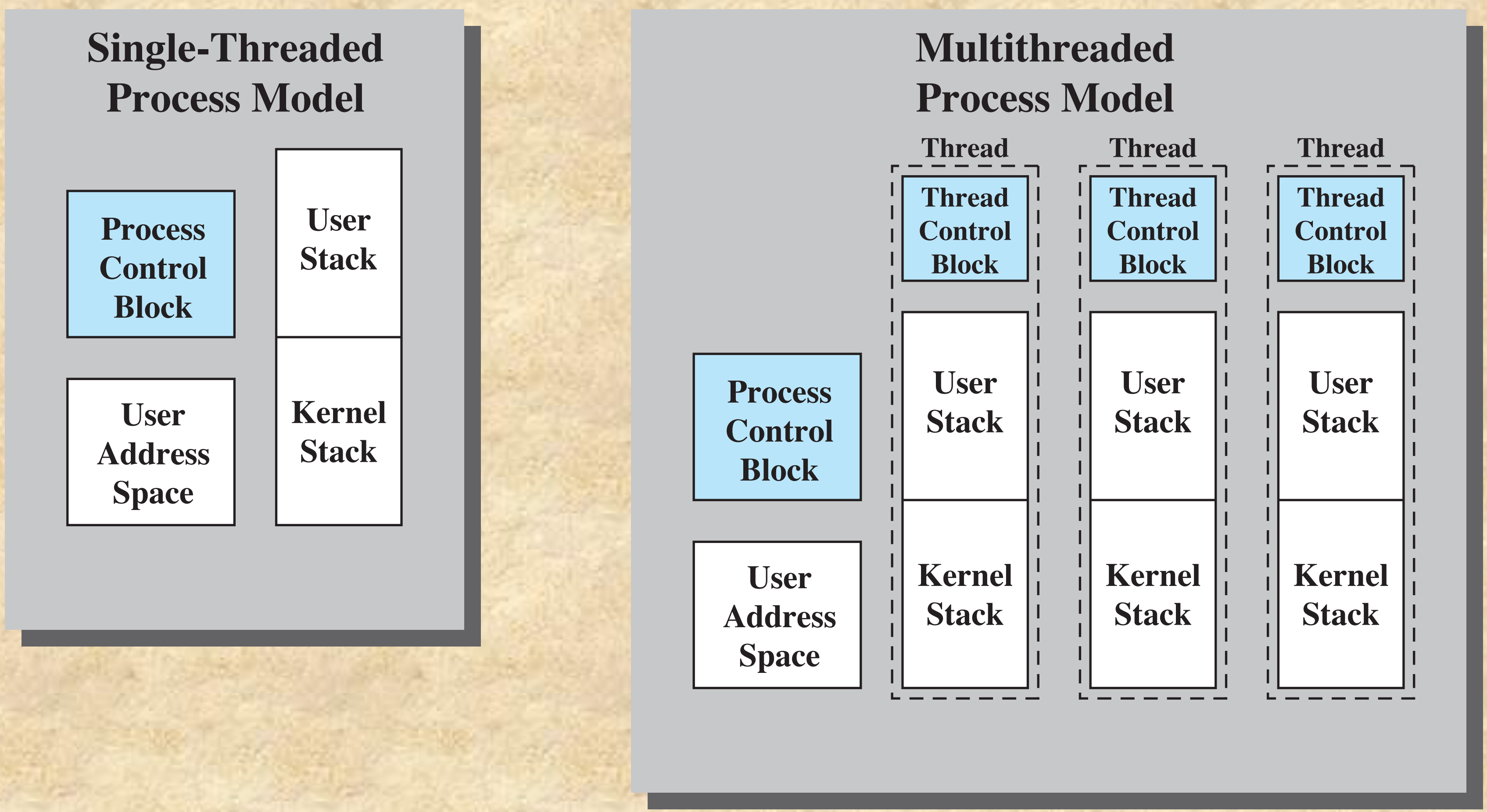


Figure 4.2 Single Threaded and Multithreaded Process Models

Key Benefits of Threads

- Takes less time to create a new thread than a process
- Less time to terminate a thread than a process
- Switching between two threads takes less time than switching between processes
- Threads enhance efficiency in communication between programs

Thread Use in a Single-User System

- Foreground and background work
- Asynchronous processing
- Speed of execution
- Modular program structure

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

  - Suspending a process involves suspending all threads of the process
  - Termination of a process terminates all threads within the process

Thread Execution States

The key states for a thread are:

- Running
- Ready
- Blocked

Thread operations associated with a change in thread state are:

- Spawn
- Block
- Unblock
- Finish

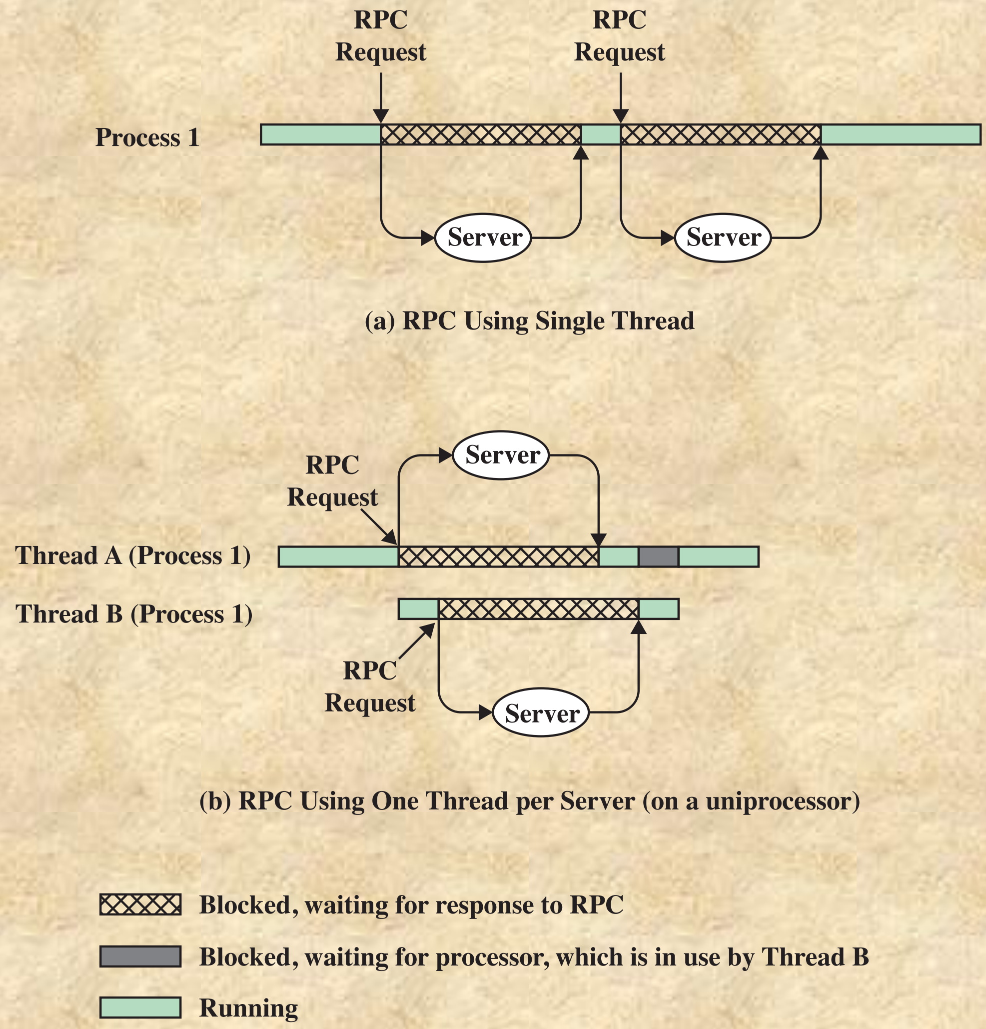


Figure 4.3 Remote Procedure Call (RPC) Using Threads

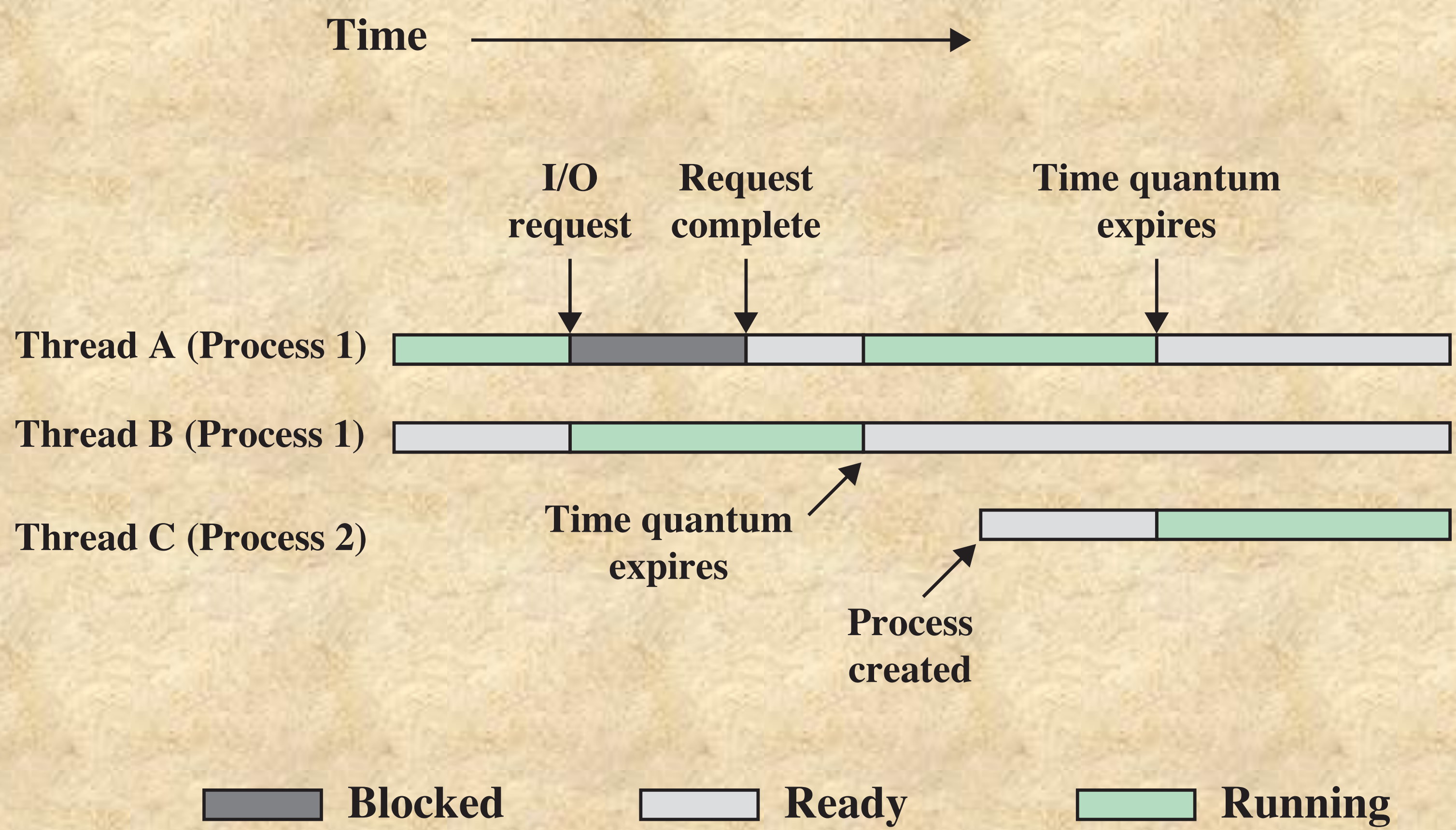
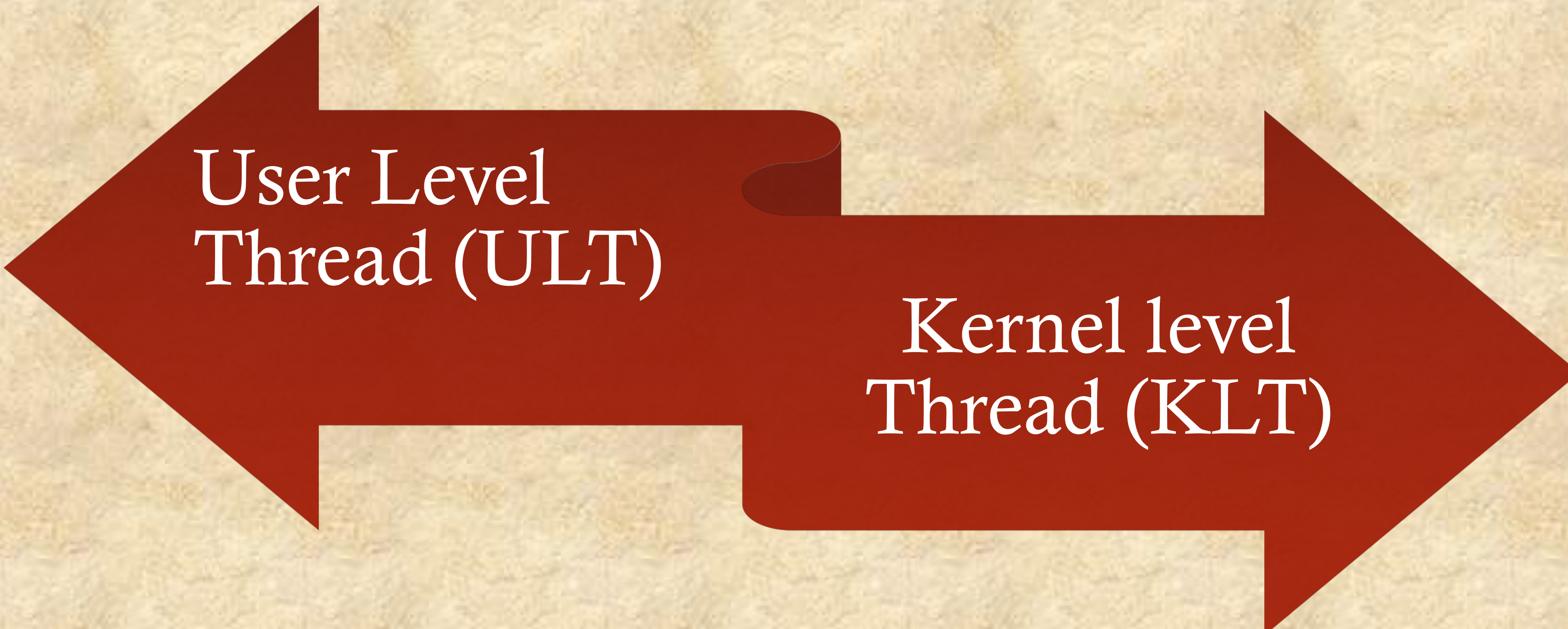


Figure 4.4 Multithreading Example on a Uniprocessor

Thread Synchronization

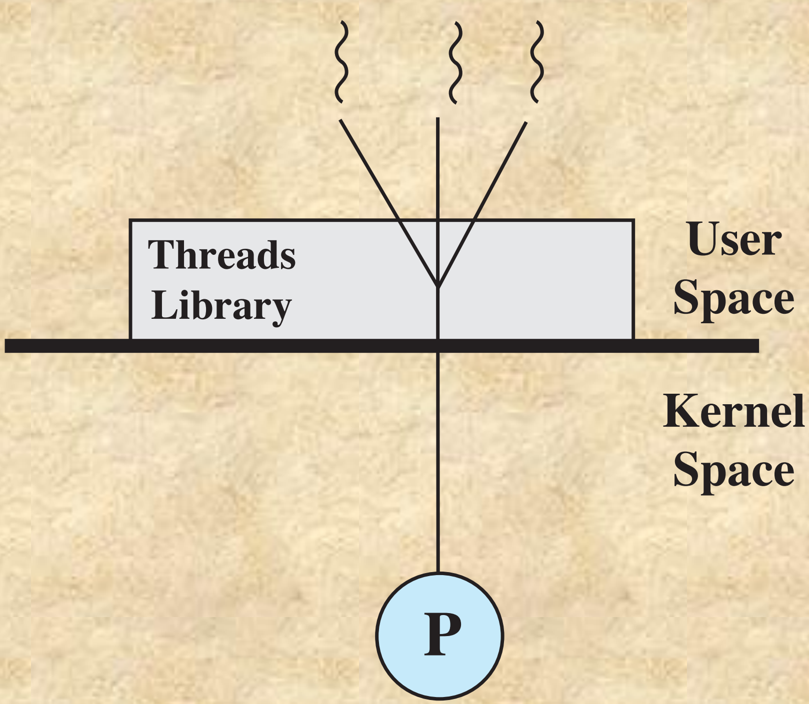
- It is necessary to synchronize the activities of the various threads
    - All threads of a process share the same address space and other resources
    - Any alteration of a resource by one thread affects the other threads in the same process

Types of Threads



User-Level Threads (ULTs)

- All thread management is done by the application
- The kernel is not aware of the existence of threads



(a) Pure user-level

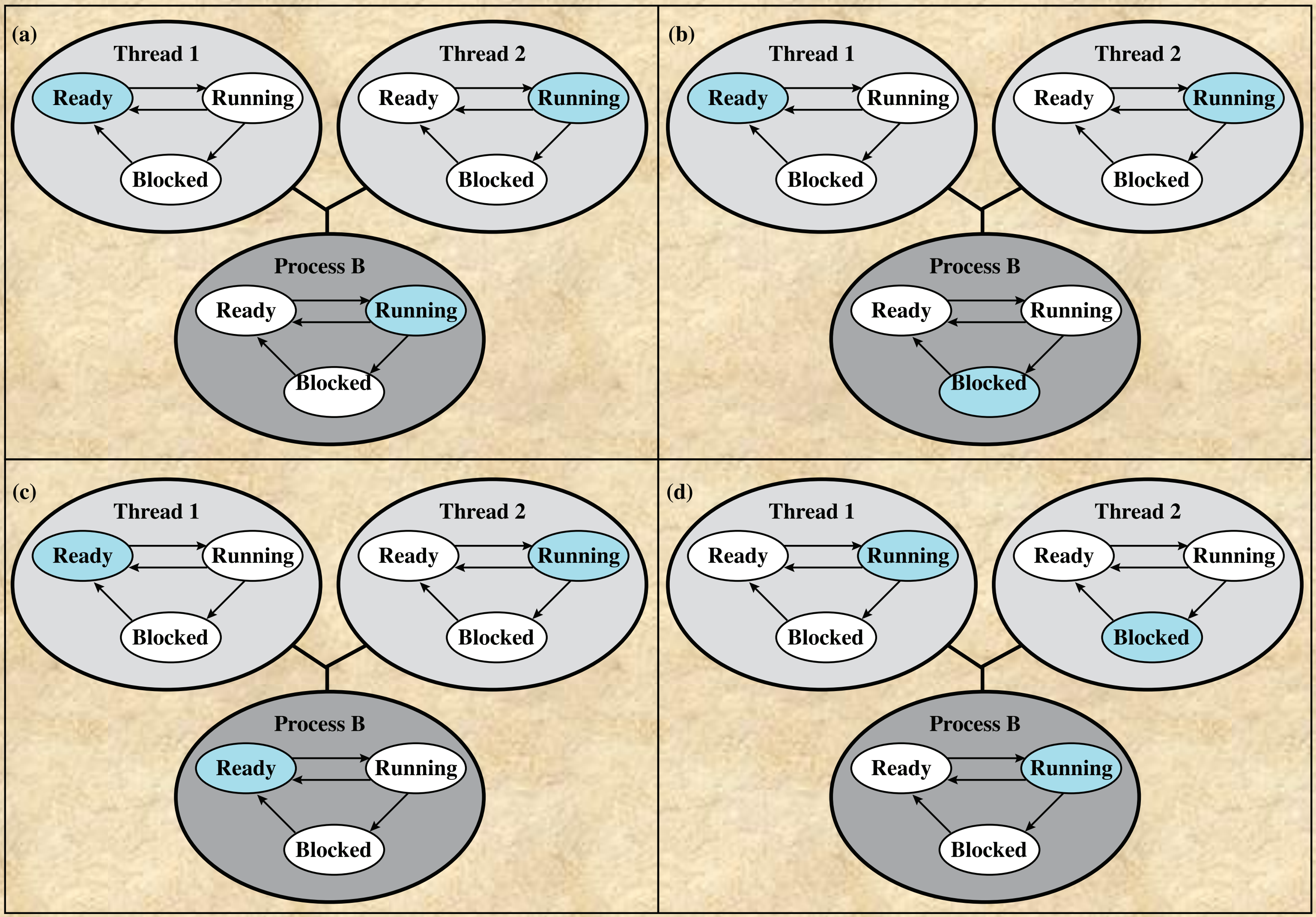


Figure 4.6 Examples of the Relationships Between User-Level Thread States and Process States

Advantages of ULTs

- Thread switching does not require kernel mode privileges
- Scheduling can be application specific
- ULTs can run on any OS

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

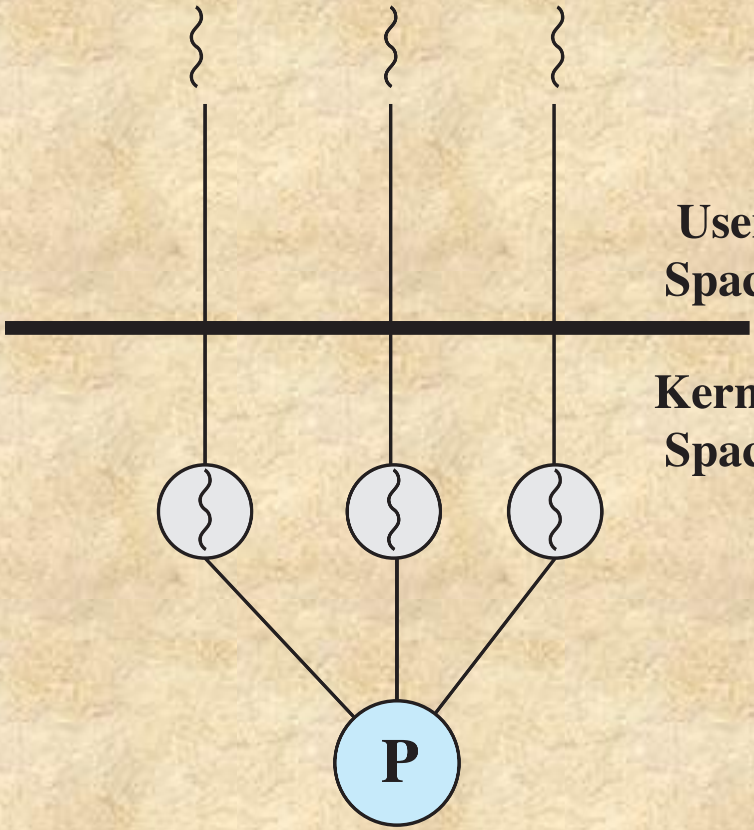
Jacketing

- Purpose is to convert a blocking system call into a non-blocking system call

Writing an application as multiple processes rather than multiple threads

- However, this approach eliminates the main advantage of threads

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

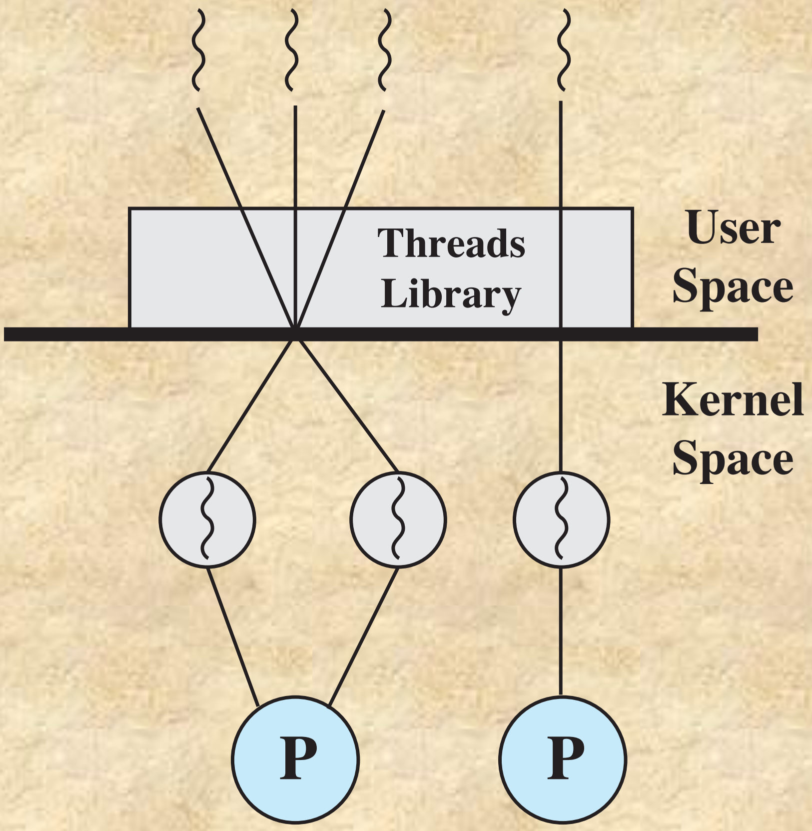
- The transfer of control from one thread to another within the same process requires a mode switch to the kernel

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)

Combined Approaches

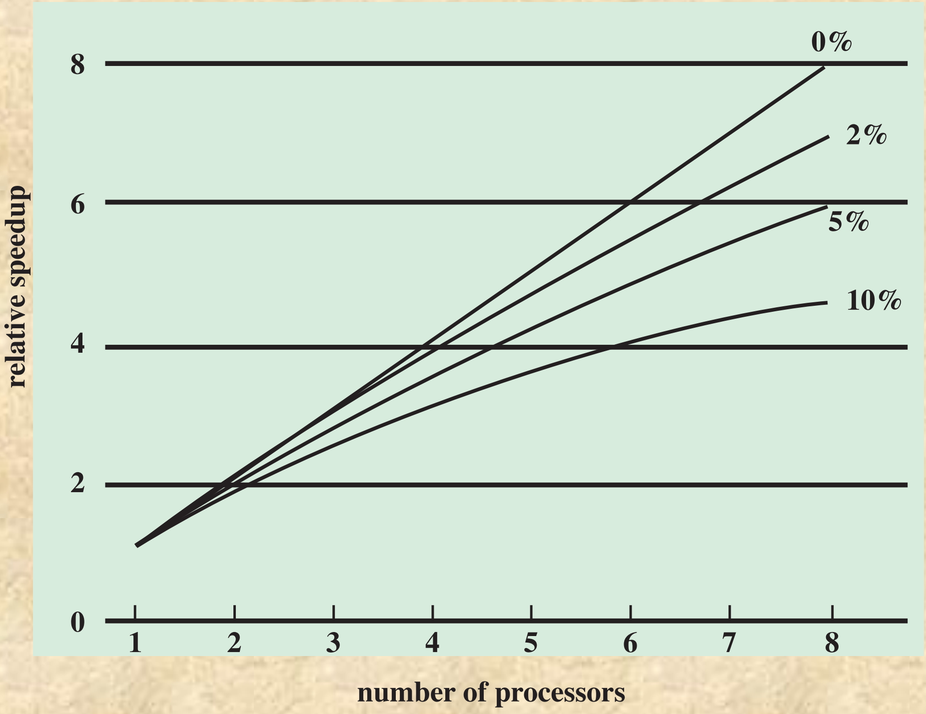
- Thread creation is done completely in the user space, as is the bulk of the scheduling and synchronization of threads within an application
- Solaris is a good example



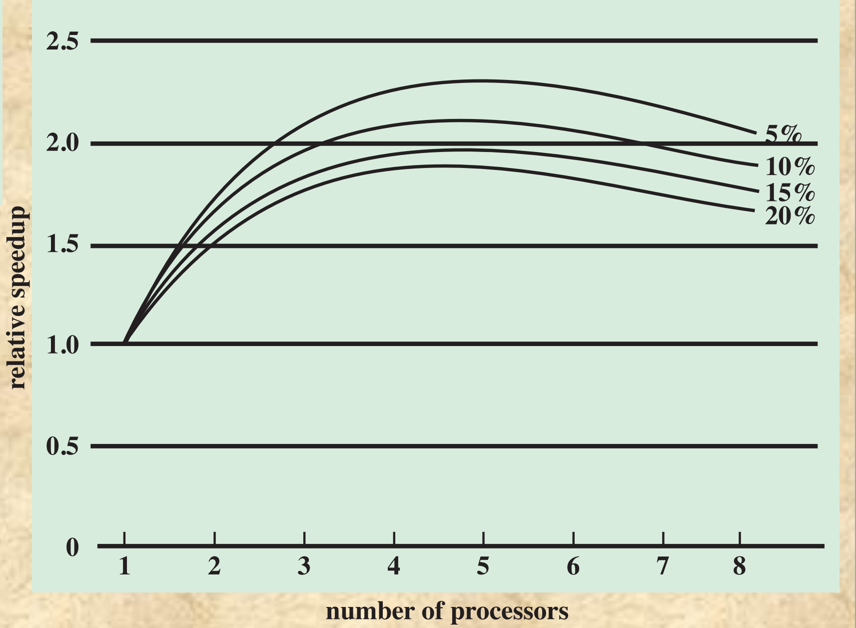
(c) Combined

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



(a) Speedup with 0%, 2%, 5%, and 10% sequential portions



(b) Speedup with overheads

Figure 4.7 Performance Effect of Multiple Cores

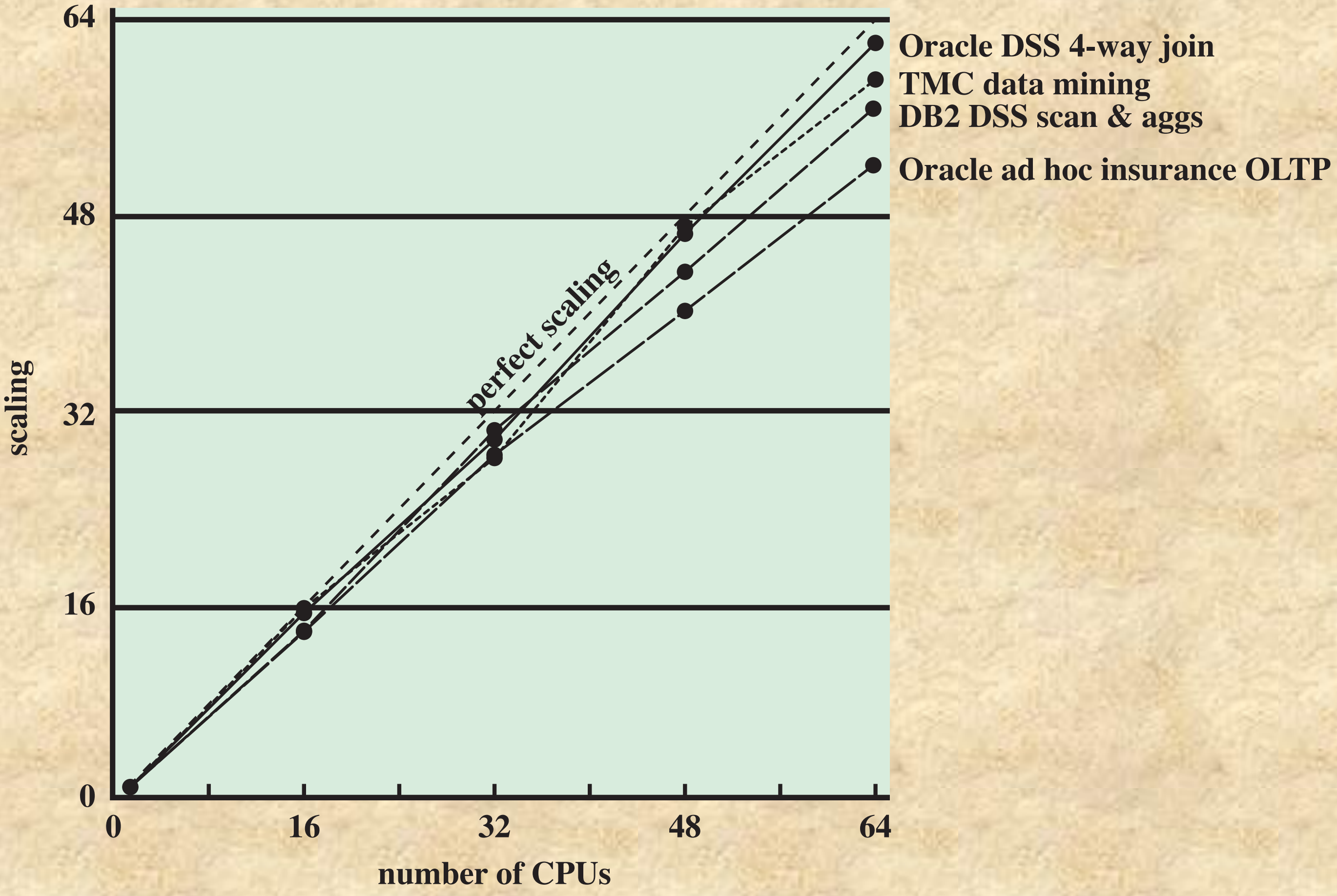


Figure 4.8 Scaling of Database Workloads on Multiple-Processor Hardware

Applications That Benefit

- 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

Multithreading

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

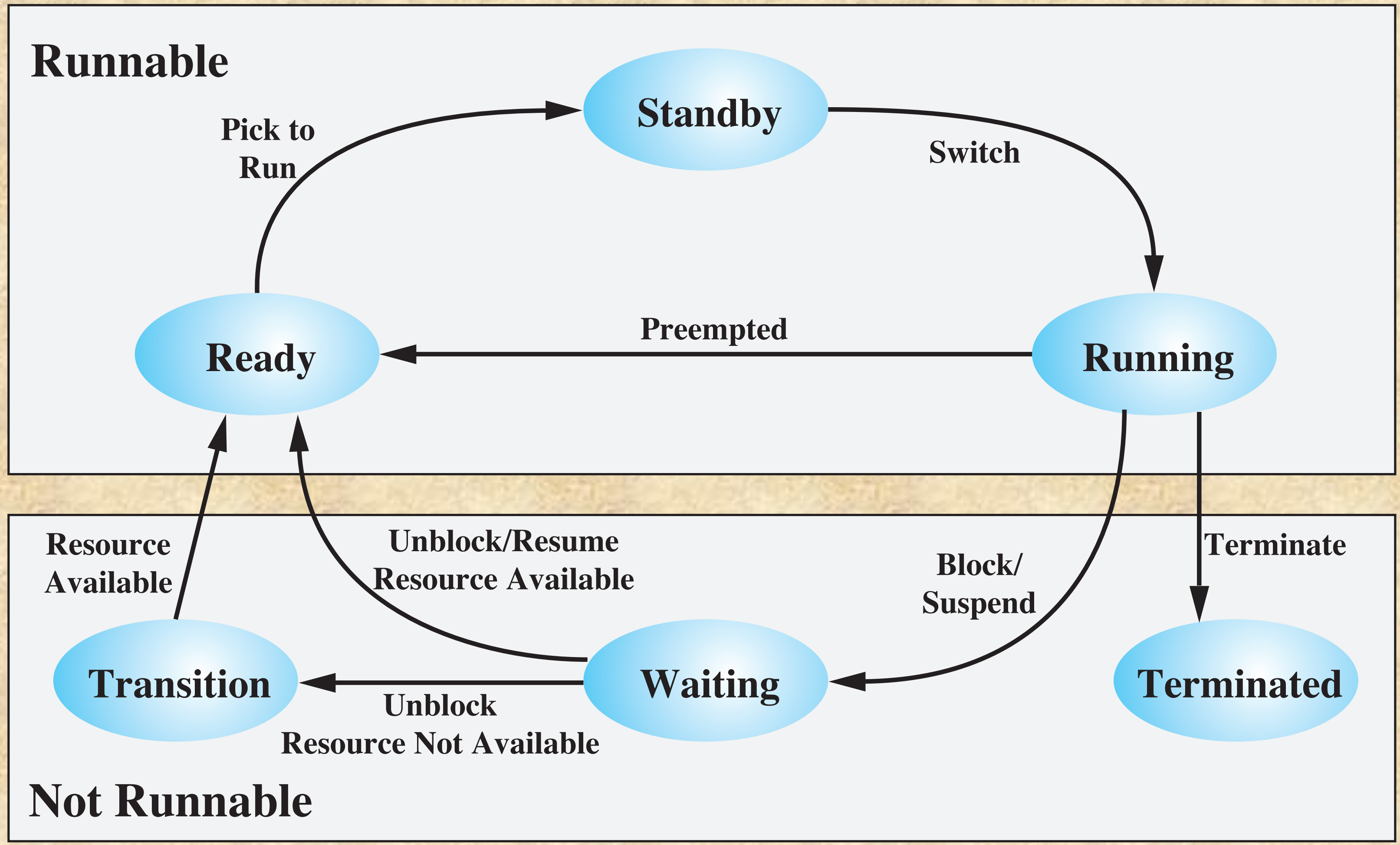


Figure 4.11 Windows Thread States

Solaris Process

- Makes use of four thread-related concepts:

Process	• Includes the user's address space, stack, and process control block
User-level Threads	• A user-created unit of execution within a process
Lightweight Processes (LWP)	• A mapping between ULTs and kernel threads
Kernel Threads	• Fundamental entities that can be scheduled and dispatched to run on one of the system processors

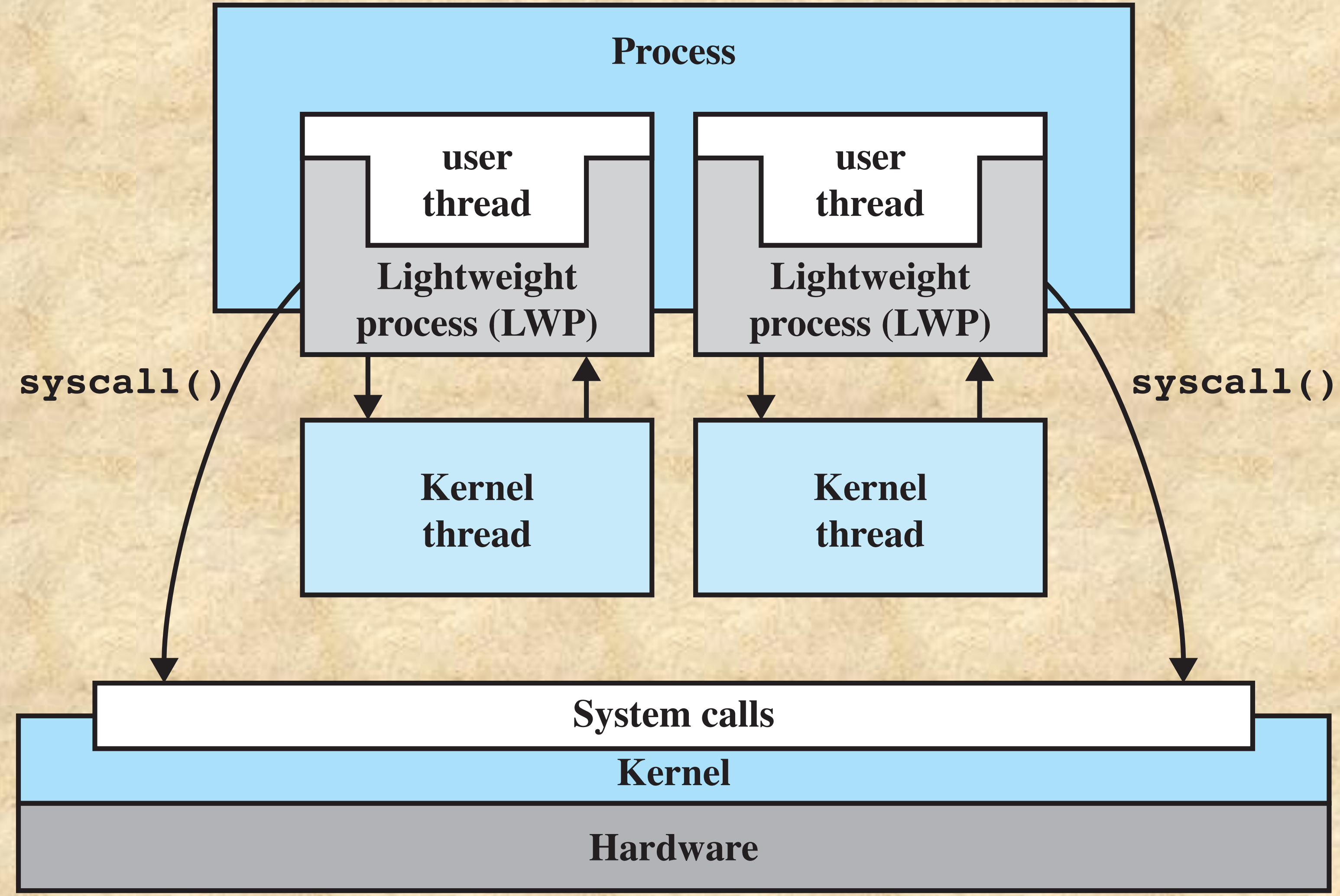


Figure 4.12 Processes and Threads in Solaris

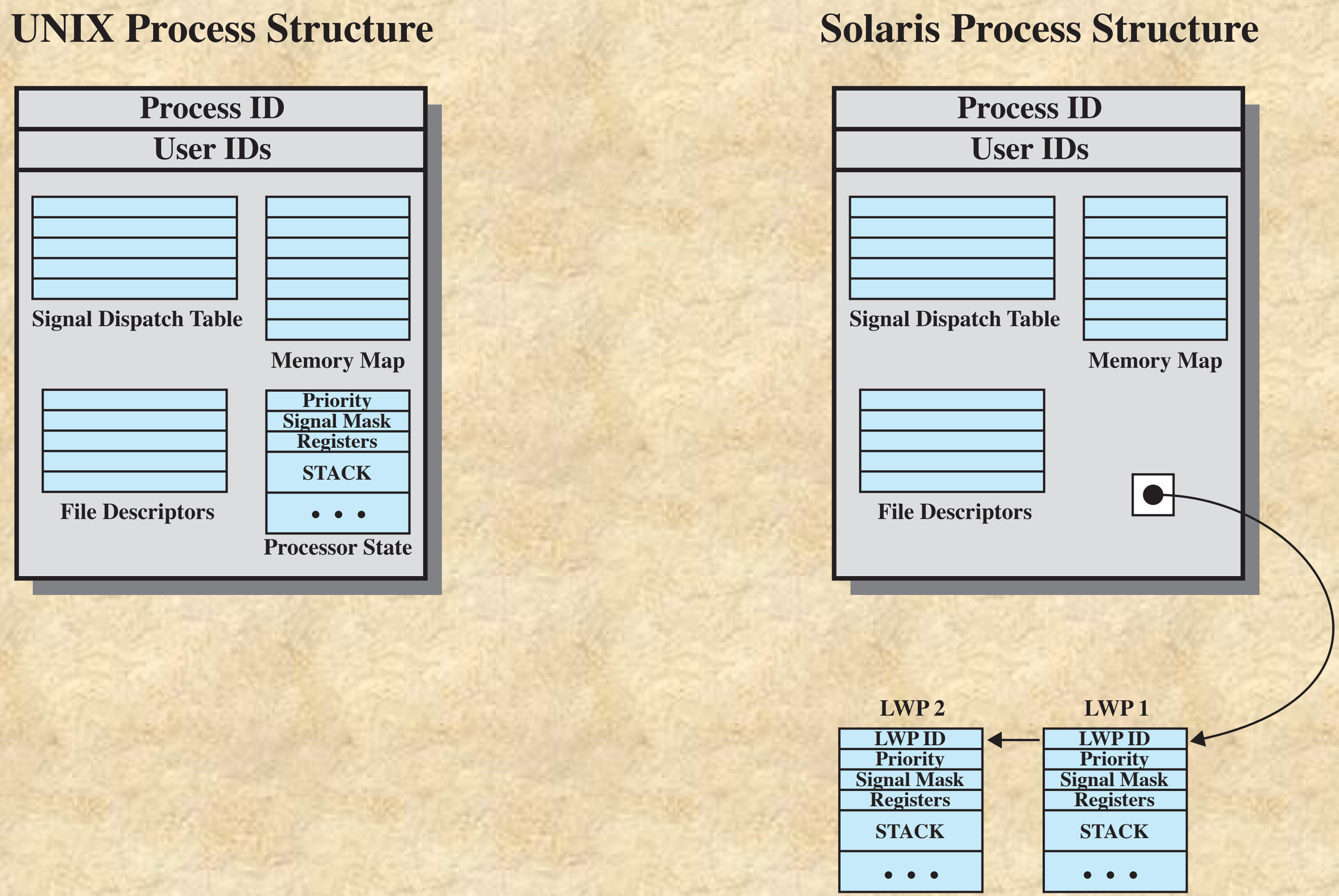


Figure 4.13 Process Structure in Traditional UNIX and Solaris [LEWI96]

A Lightweight Process (LWP) Data Structure Includes:

- An LWP identifier
- The priority of this LWP and hence the kernel thread that supports it
- A signal mask that tells the kernel which signals will be accepted
- Saved values of user-level registers
- The kernel stack for this LWP, which includes system call arguments, results, and error codes for each call level
- Resource usage and profiling data
- Pointer to the corresponding kernel thread
- Pointer to the process structure

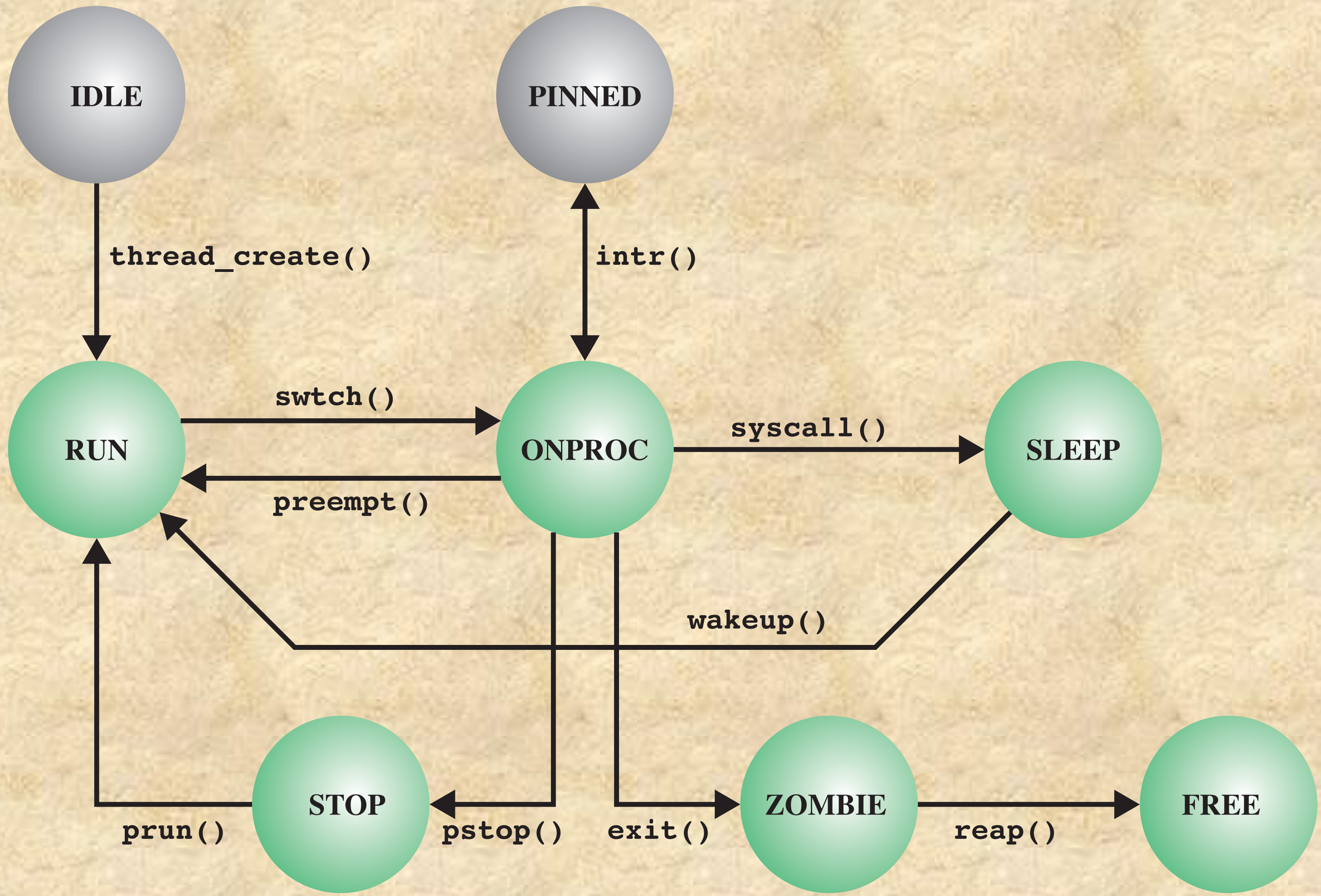


Figure 4.14 Solaris Thread States

Interrupts as Threads

- Most operating systems contain two fundamental forms of concurrent activity:

Processes (threads)	Cooperate with each other and manage the use of shared data structures by primitives that enforce mutual exclusion and synchronize their execution
Interrupts	Synchronized by preventing their handling for a period of time

- Solaris unifies these two concepts into a single model, namely kernel threads, and the mechanisms for scheduling and executing kernel threads
  - To do this, interrupts are converted to kernel threads

Solaris Solution

- Solaris employs a set of kernel threads to handle interrupts
  - An interrupt thread has its own identifier, priority, context, and stack
  - The kernel controls access to data structures and synchronizes among interrupt threads using mutual exclusion primitives
  - Interrupt threads are assigned higher priorities than all other types of kernel threads

Linux Tasks

A process, or task, in Linux is represented by a *task_struct* data structure

This structure contains information in a number of categories

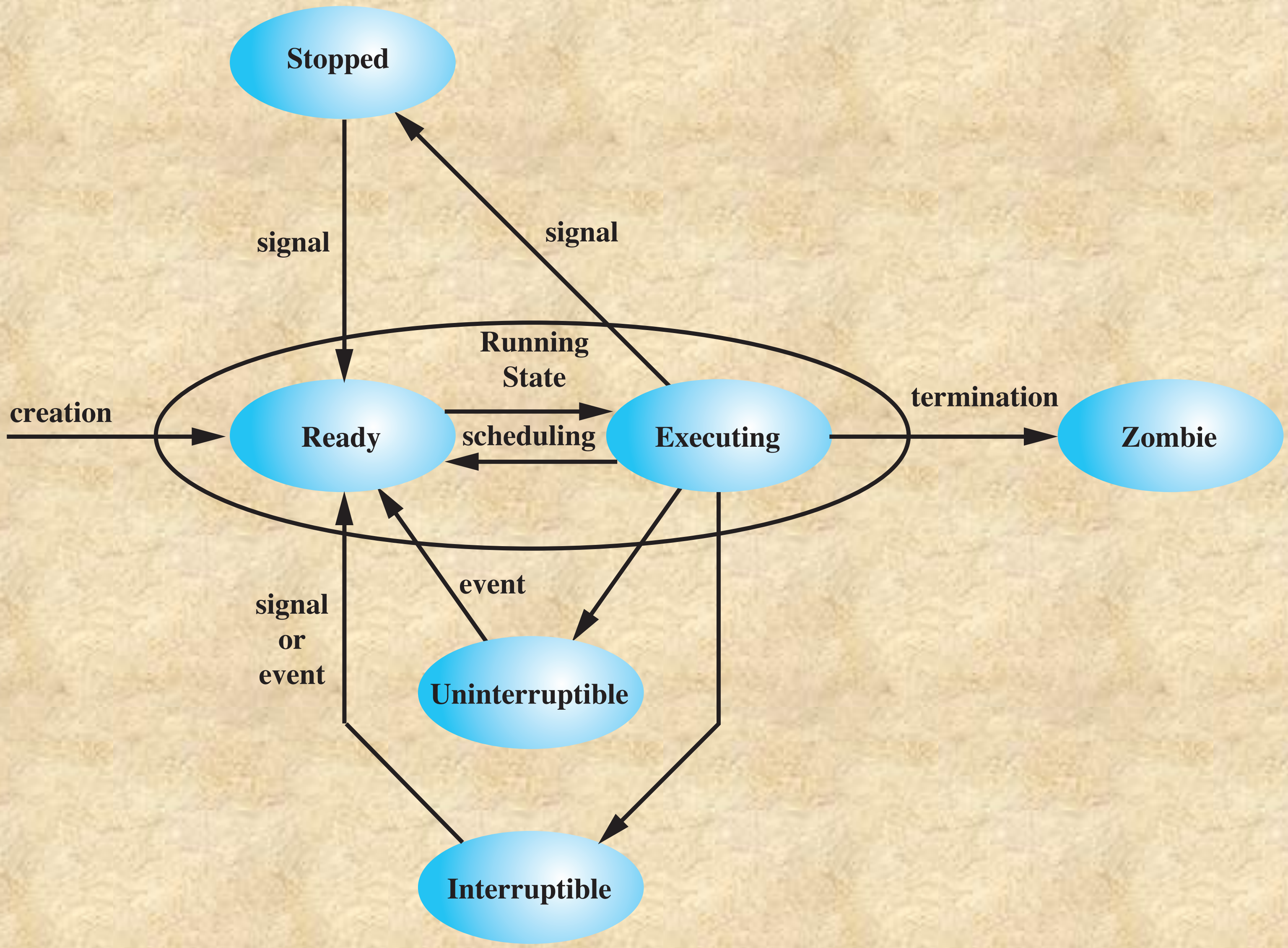


Figure 4.15 Linux Process/Thread Model

Linux Threads

- Linux does not recognize a distinction between threads and processes
- User-level threads are mapped into kernel-level processes
- A new process is created by copying the attributes of the current process
- The new process can be *cloned* so that it shares resources
- The clone() call creates separate stack spaces for each process

Linux Namespaces

- A namespace enables a process to have a different view of the system than other processes that have other associated namespaces
- There are currently six namespaces in Linux
  - mnt
  - pid
  - net
  - ipc
  - uts
  - user

Summary

- Processes and threads
  - Multithreading
  - Thread functionality
- Types of threads
  - User level and kernel level threads
- Multicore and multithreading
  - Performance of Software on Multicore
- Windows process and thread management
  - Management of background tasks and application lifecycles
  - Windows process
  - Process and thread objects
  - Multithreading
  - Thread states
  - Support for OS subsystems
- Solaris thread and SMP management
  - Multithreaded architecture
  - Motivation
  - Process structure
  - Thread execution
  - Interrupts as threads
- Linux process and thread management
  - Tasks/threads/namespaces
- Android process and thread management
  - Android applications
  - Activities
  - Processes and threads
- Mac OS X grand central dispatch