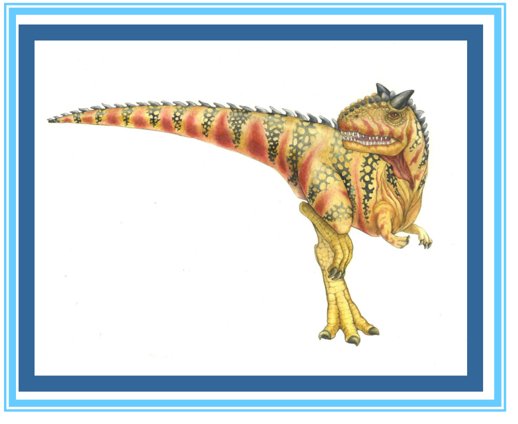
Chapter 4: Threads



Chapter 4: Threads

- Overview
- Implementation of Thread
- Multithreading Models
- Threading issues
- Operating System Examples

Overview

- To build parallel programs, such as:
  - Parallel execution on a multiprocessor
  - Web server to handle multiple simultaneous web requests
- We will need to:
  - Create several processes that can execute in parallel
  - The OS will then schedule these processes in parallel

Processes Overheads

- A full process includes numerous things:
  - an address space (defining all the code and data pages)
  - OS resources and accounting information
  - a "thread of control",
    - defines where the process is currently executing
    - That is the PC and registers
- Creating a new process is costly
  - all of the structures (e.g., page tables) that must be allocated
- Communicating between processes is costly
  - most communication goes through the OS

Need “Lightweight” Processes

- What’s similar in these processes?
  - They all share the same code and data (address space)
  - They all share the same privileges
  - They share almost everything in the process
- What don’t they share?
  - Each has its own PC, registers, and stack pointer
- Idea: why don’t we separate the idea of process (address space, accounting, etc.) from that of the minimal “thread of control” (PC, SP, registers)?

Threads and Processes

- Most operating systems therefore support two entities:
  - the process,
    - which defines the address space and general process attributes
  - the thread,
    - which defines a sequential execution stream within a process
- Threads are the unit of scheduling
- Processes are *containers* in which threads execute

Overview

- A traditional process is basic unit of program concurrent execution
  - It is a basic unit to own resources
  - It is a basic unit of CPU scheduling
- So, a traditional process is called heavyweight process
- However, most modern OS separate the two attributes of process
  - Process is a unit to own resources
  - Thread is a basic unit of CPU utilization
- So , a thread is called a lightweight process

Overview

- A *thread* (or *lightweight process*) is a basic unit of CPU utilization
  - program counter
  - register set
  - stack space
- A thread shares with its peer threads its:
  - code section
  - data section
  - operating-system resources, such as open files and signals
  - collectively known as a task.

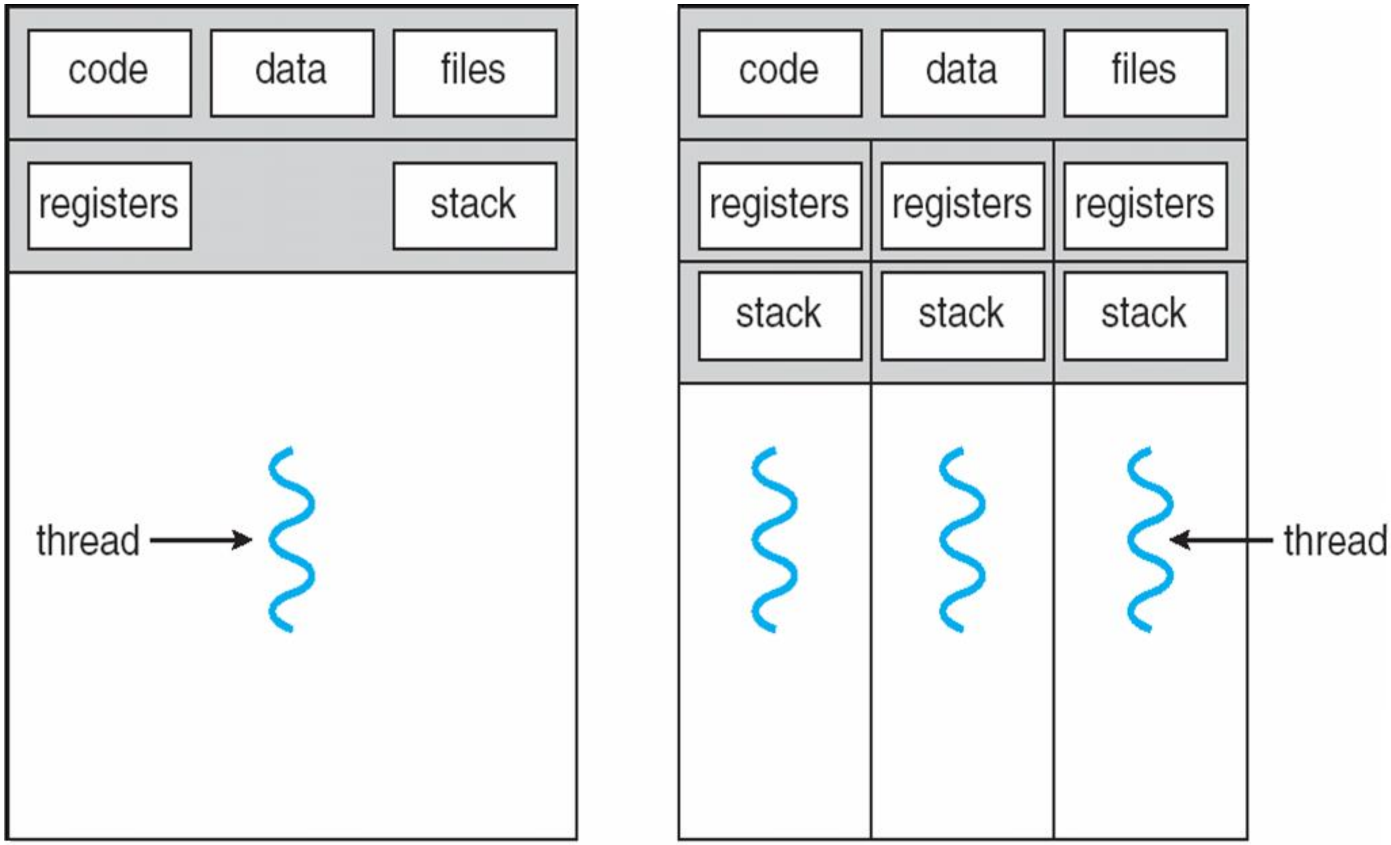
Overview

- Many software packages running on modern PCs are multithreaded
- An application is typically implemented as a separate process with several threads of control.
  - A web server might have one thread display images or text while another thread retrieves data from the network
  - A word processor may have one thread for displaying graphics, another thread for reading keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

Overview

- There are also situations where a single application may be required to perform several similar tasks.
  - E.g. a web server accepts client requests for web pages, images, sound, and so forth.
- If the web server ran as a traditional process, it would be able to service only one client at a time with its single process.
  - A client might wait a long time to be serviced
  - It may be more efficient for one process that contains several threads to serve the same purpose
- Multithread the web server
  - One thread listening for client requests
  - Create another thread to service the request when request coming.

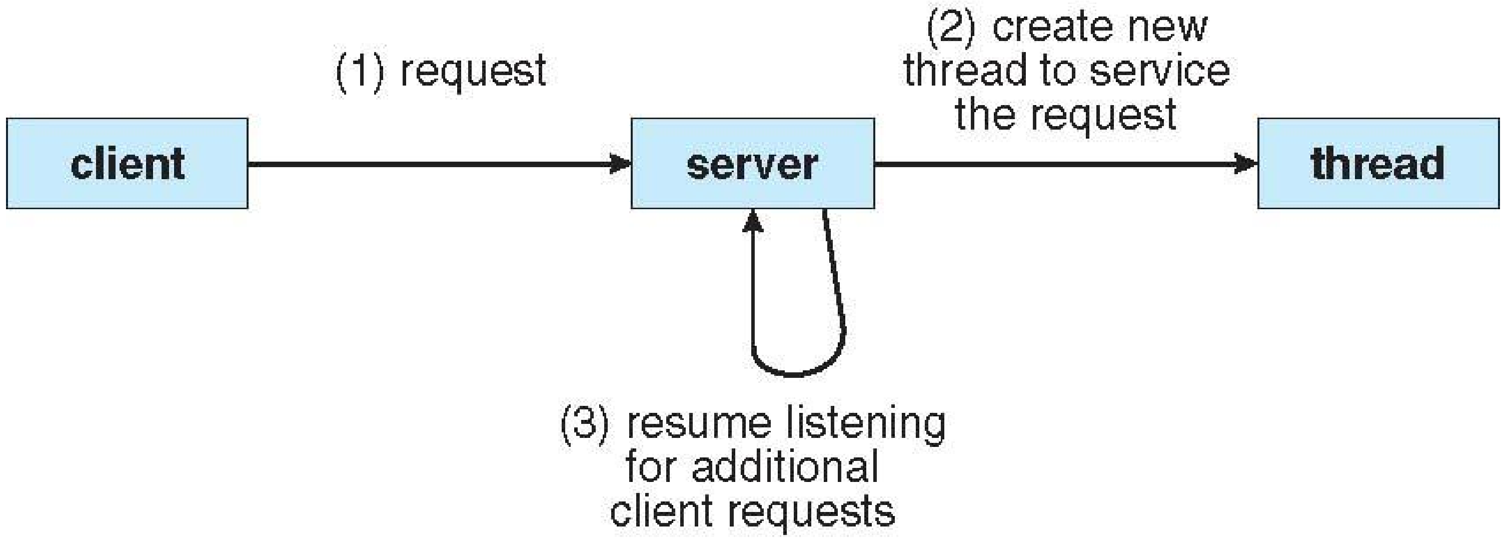
Single and Multithreaded Processes



single-threaded process

multithreaded process

Multithreaded Server Architecture



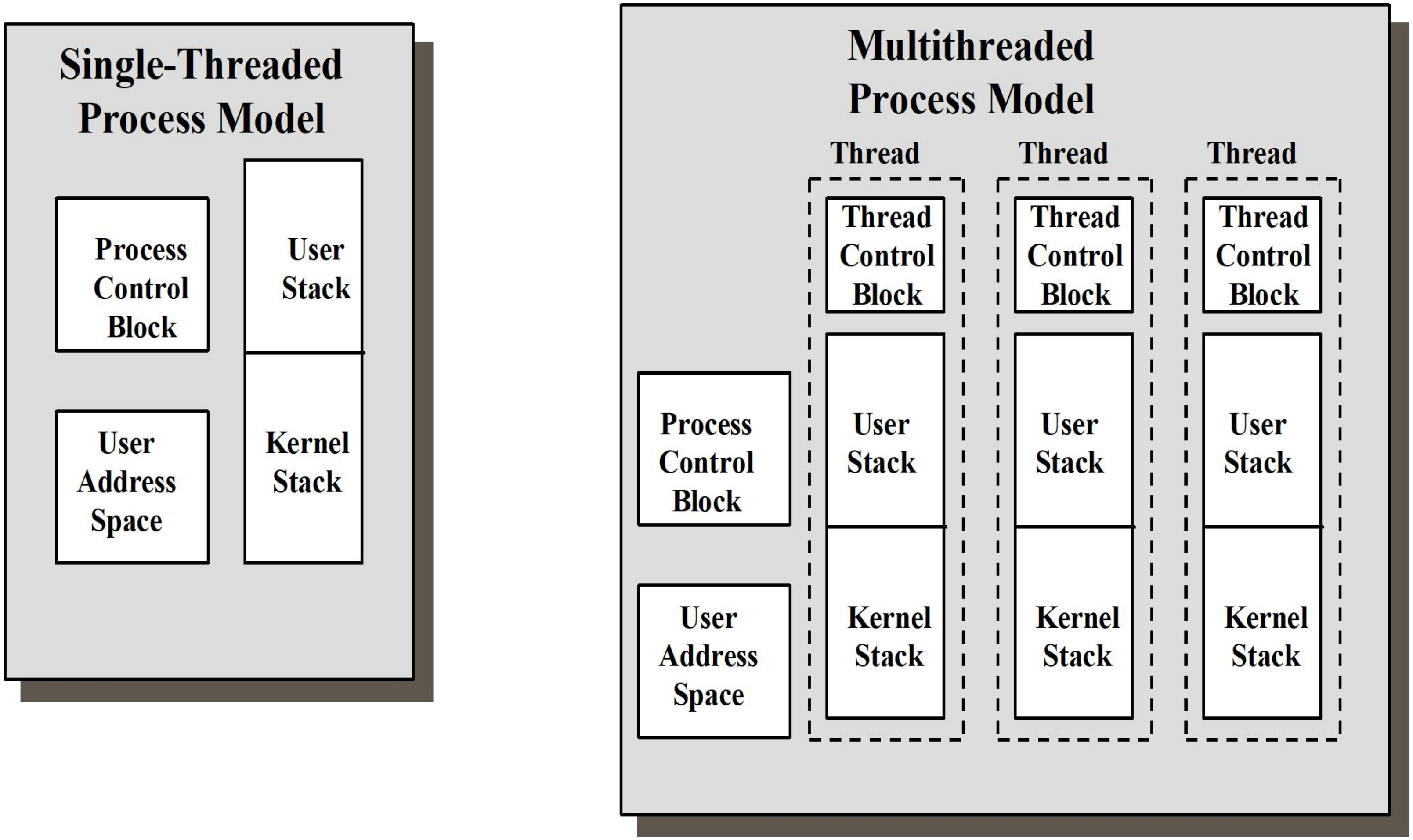
Threads vs. Processes

- A thread has no data segment or heap
- A thread cannot live on its own, it must live within a process
- *There can be more than one thread in a process*

- Inexpensive creation
- Inexpensive context switching
- If a thread dies, its stack is reclaimed

- A process has code/data/heap & other segments
- There must be at least one thread in a process
- Threads within a process share code/data/heap, share I/O, but each has its own stack & registers

- Expensive creation
- Expensive context switching
- If a process dies, its resources are reclaimed & all threads die


Single-Threaded
Process Model
Process
Control
Block
User
Stack
User
Address
Space
Kernel
Stack
Multithreaded
Process Model
Thread
Thread
Thread
Thread
Control
Block
Thread
Control
Block
Thread
Control
Block
Process
Control
Block
User
Stack
User
Stack
User
Stack
User
Address
Space
Kernel
Stack
Kernel
Stack
Kernel
Stack

Separating Threads and Processes

- Concurrency (multithreading) is useful for:
  - improving program structure
  - handling concurrent events (e.g., web requests)
  - building parallel programs
  - Resource sharing
  - Multiprocessor utilization
- Is multithreading useful even on a single processor?

Benefits

- Responsiveness
- Resource Sharing
- Economy(overhead)
- Scalability

Implementation of thread

- Kernel-supported threads
  - Supported directly by the operating system
- User-level threads
  - supported above the kernel, via a set of library calls at the user level

User-Level Threads

- Thread management done by user-level threads library
  - The library provides support for thread creation, scheduling, and management with no support from the kernel
- Each thread is represented simply by:
  - PC
  - Registers
  - Stack
  - Small control block
- All thread operations are at the user-level:
  - Creating a new thread
  - switching between threads
  - synchronizing between threads

User-Level Threads

- the thread scheduler is part of a library, outside the kernel
- thread context switching and scheduling is done by the library
- Can either use cooperative or preemptive threads
  - cooperative threads are implemented by:
    - CreateThread(), DestroyThread(), Yield(), Suspend(), etc.
  - preemptive threads are implemented with a timer (signal)
    - where the timer handler decides which thread to run next

User-Level Threads

- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads

Example User Thread Interface

- t = thread_fork(initial context):

  create a new thread of control;

- thread_stop():

  stop the calling thread, sometimes called thread_block;

- thread_start(t): start the named thread;
- thread_yield(): voluntarily give up the processor;
- thread_exit():

  terminate the calling thread, sometimes called
thread_destroy;

Kernel Threads

- Supported by the Kernel
  - Thread is the unit of CPU scheduling

- Examples
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

User-Level vs. Kernel Threads

User-Level

- Managed by application
- Kernel not aware of thread
- Context switching cheap
- Create as many as needed
- Must be used with care

Kernel-Level

- Managed by kernel
- Consumes kernel resources
- Context switching expensive
- Number limited by kernel resources
- Simpler to use

Key issue: kernel threads provide virtual processors to user-level threads, but if all of kthreads block, then all user-level threads will block *even* if the program logic allows them to proceed

Multithreading Models

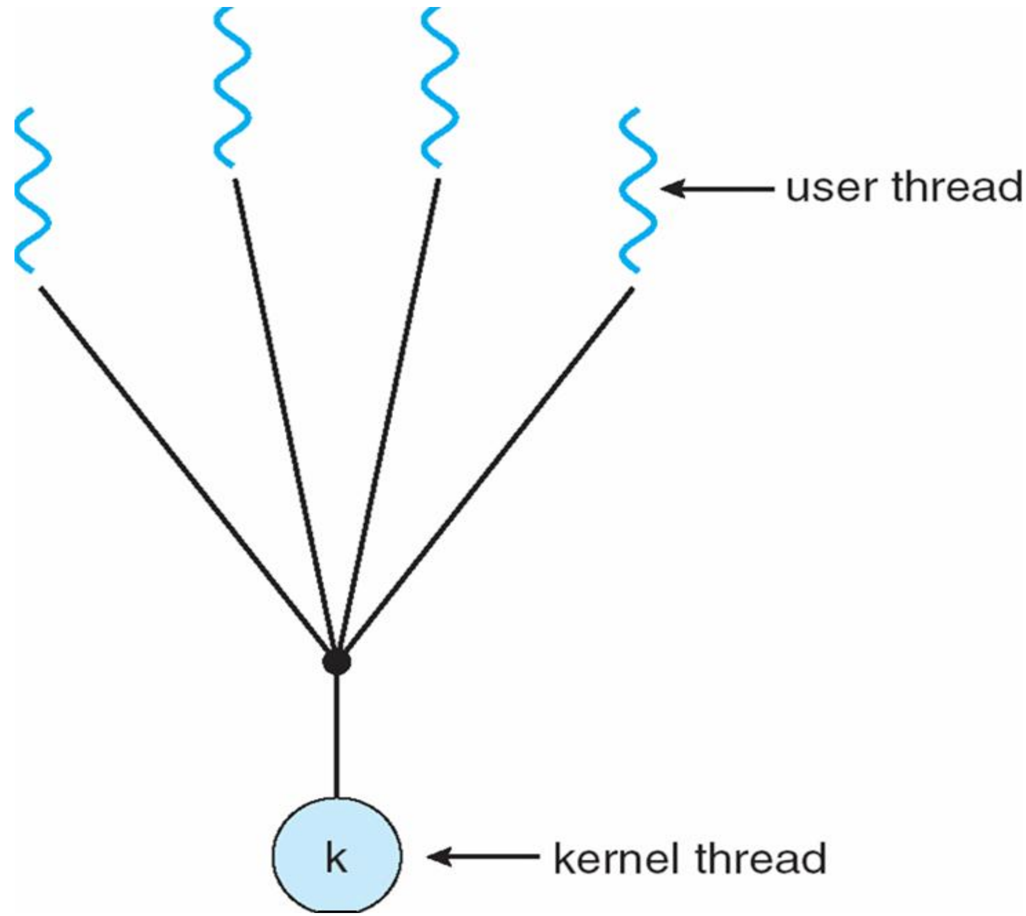
- Many-to-One
- One-to-One
- Many-to-Many
- Two level model

Many-to-One

- Many user-level threads mapped to single kernel thread
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads

Many-to-One Model

Many user-level threads mapped to single kernel thread

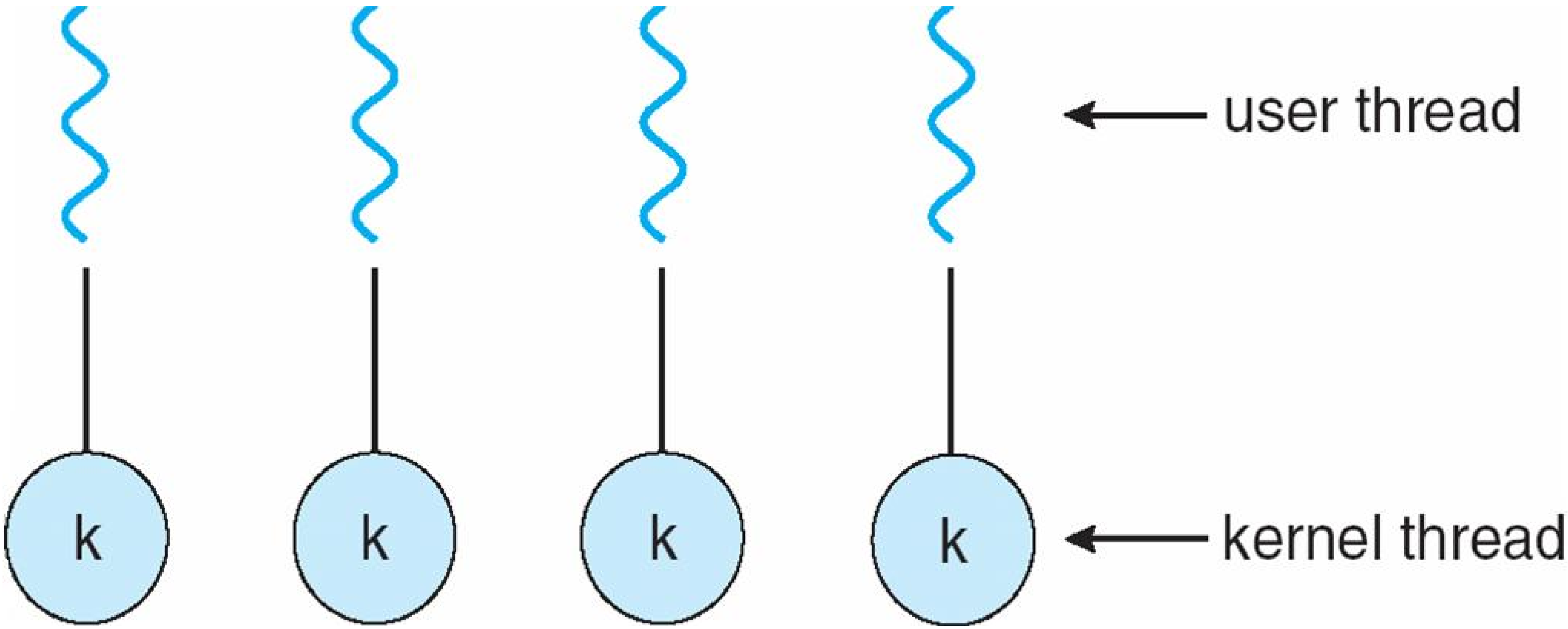


- No parallel execution of threads - can't exploit multiple CPUs
- All threads block when one uses synchronous I/O

One-to-One

- Each user-level thread maps to kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

One-to-one Model

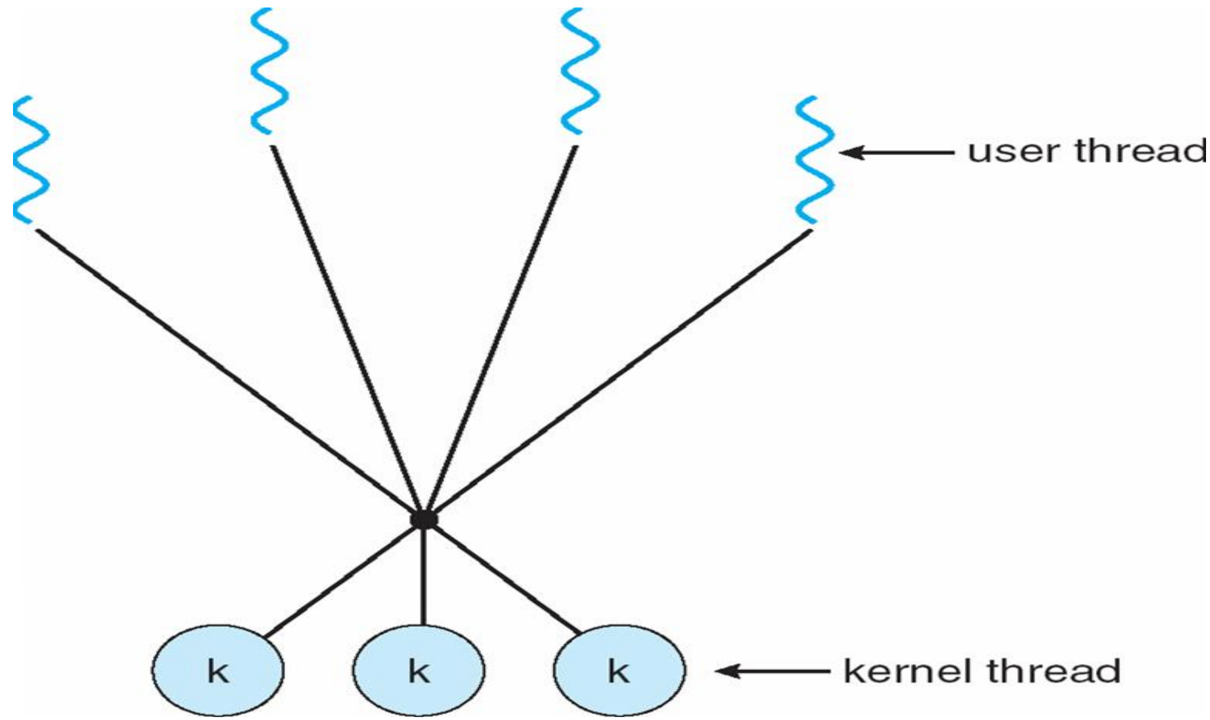


- More concurrency
- Better multiprocessor performance
- Each user thread requires creation of kernel thread
- Each thread requires kernel resources; limits number of total threads

Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Examples
  - Solaris prior to version 9
  - Windows NT/2000 with the *ThreadFiber* package

Many-to-Many Model



If U<k? No benefits of multithreading

If U>k, some threads may have to wait for an Kthread to run

- Active thread - executing on an Kthread
- Runnable thread - waiting for an Kthread

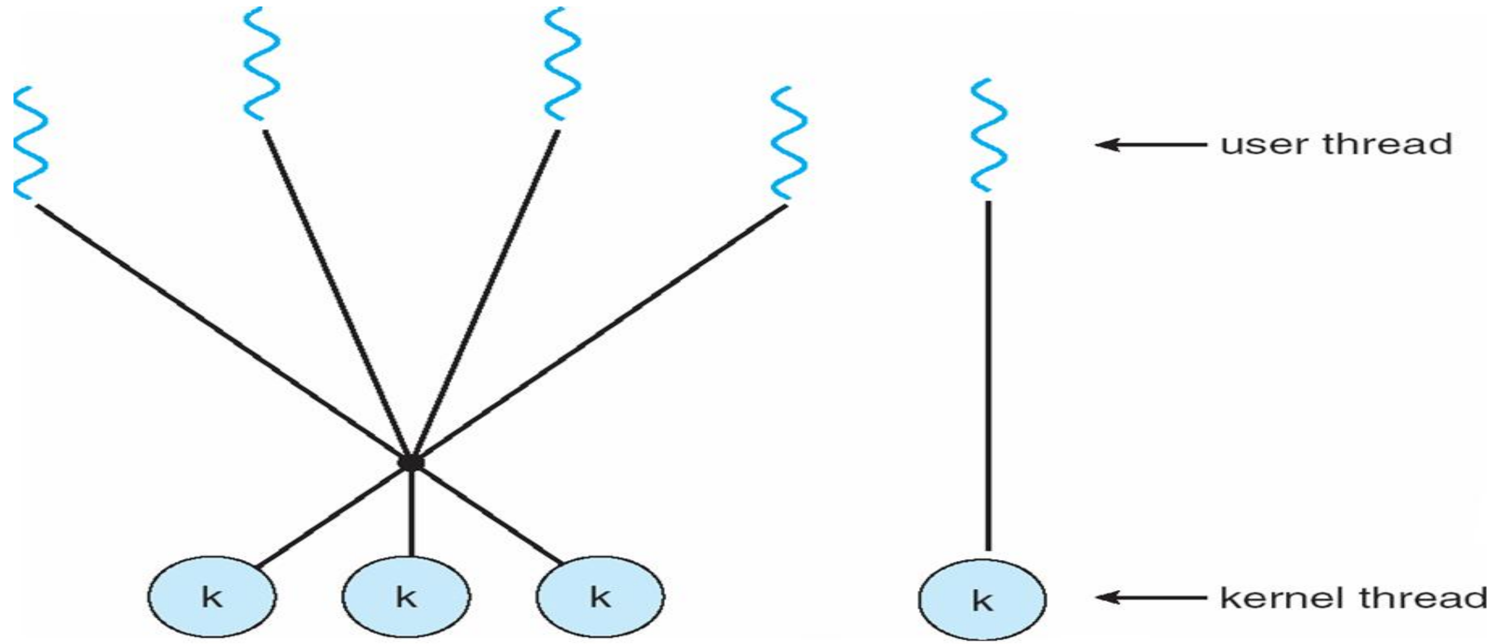
A thread gives up control of Kthread under the following:

– synchronization, lower priority, yielding, time slicing

Two-level Model

- Similar to M:N(M>=N), except that it allows a user thread to be bound to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier

Two-level Model



- Supports both bound and unbound threads
  - Bound threads - permanently mapped to a single, dedicated kthread
  - Unbound threads - may move among kthreads in set
- Thread creation, scheduling, synchronization done in user space

Thread Libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS
- Examples:
  - POSIX Pthreads
  - Win32 threads
  - Java thread

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

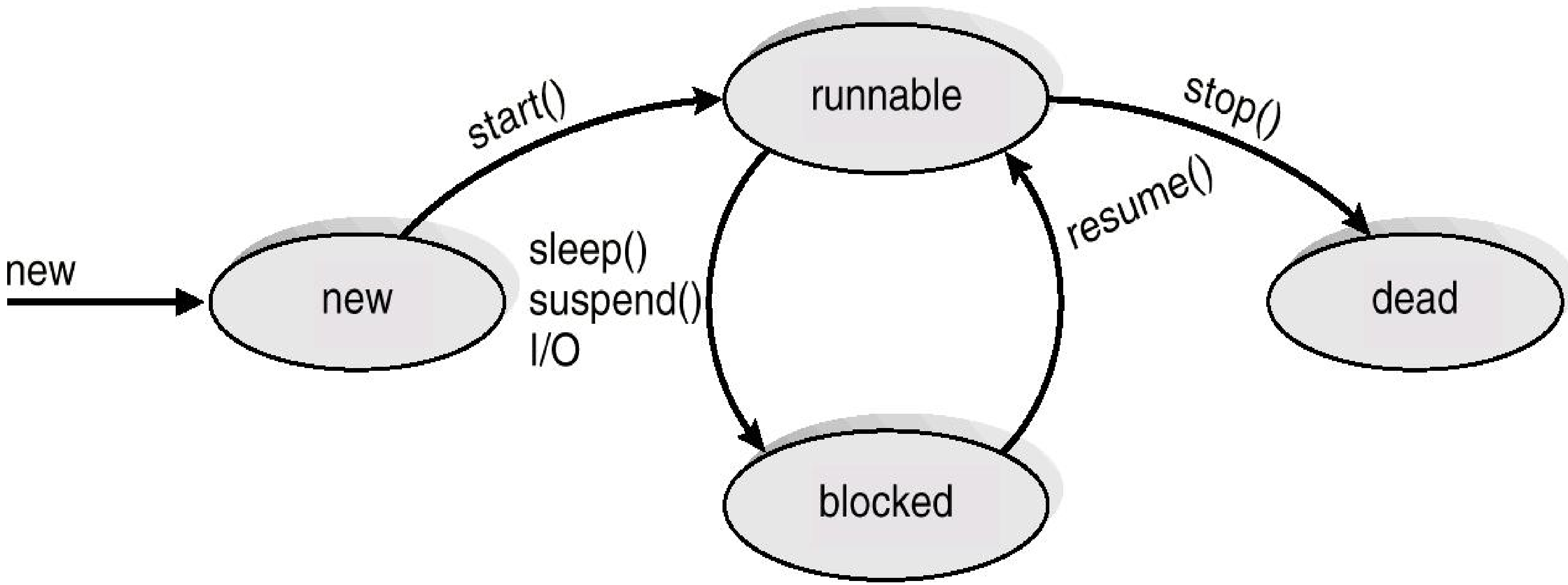
Win32 Threads

- Win32 API is the primary API for Microsoft OS (Win95,98,NT,2000,XP)
- A kernel-level library on windows systems

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

Java Thread States



Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Thread cancellation of target thread
  - Asynchronous or deferred
- Signal handling
- Thread pools
- Thread-specific data
- Scheduler activations

Semantics of fork() and exec()

- The semantics of fork() and exec() system calls change in a multithreaded program.
- If one thread calls **fork()**, does the new process duplicate only the calling thread or all threads?
- Some UNIX have chosen to have two versions of fork():
  - One duplicates all threads
  - Another duplicates only the thread calling fork()
- The **exec()** typically works in the same way as described in Chapter 3.
  - If a thread calls exec(), the program specified in the parameter to exec() will replace the entire process---- including all threads.

Thread Cancellation

- Terminating a thread before it has finished
  - E.g. a user presses a button on a web browser that stops a web page from loading any further
  - Usually, a web page is loaded using several threads----each image is loaded in a separate thread, when the browser is closed, all threads loading the page are canceled.
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately ？？
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled, allowing it an opportunity to terminate itself in an orderly fashion.

Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred.
- A signal handler is used to process signals, follow the same pattern:
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled once received

Signal Handling(Cont.)

- Handling signals in a single-threaded program is straightforward
  - Signals are always delivered to a process
- In a multithreaded program, where should a signal be delivered?
  - The following options exist:
    - Deliver the signal to the thread to which the signal applies
    - Deliver the signal to every thread in the process
    - Deliver the signal to certain threads in the process
    - Assign a specific thread to receive all signals for the process

Thread Pools

- Create a number of threads at process startup and place them into a pool where they await work.
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Limits the number of threads that exist at any one point

Thread Specific Data

- Threads belonging to a process share the data of the process.
- Sometimes, each thread might need its own copy of certain data----thread specific data
  - E.g. in a transaction processing system, we might service each transaction in a separate thread, so we need thread specific data
- Most thread libraries (Win32, Pthread, Java) provide some form of support for thread specific data

Scheduler Activations

- Many OS implementing M:M or Two-level models place an intermediate data structure between the user and kernel threads-----lightweight process(LWP)
- To the user-thread library, the LWP appears to be a virtual processor on which the application can schedule a user thread to run
- Each LWP is attached to a kernel thread, and it is kernel threads that OS schedules to run on physical processors.
- If a kernel thread blocks, the LWP blocks as well, and the user-level thread attached to the LWP also blocks.

Scheduler Activations

- Scheduler activation is one scheme for communication between the user-thread library and the kernel.
- Scheduler activations provide upcalls（回调）- a communication mechanism from the kernel to the thread library
- Upcalls are handled by the thread library with an upcall handler(回调处理程序), and an upcall handler must run on a virtual processor(LWP).

Operating System Examples

- Windows XP Threads
- Linux Threads
- Solaris threads

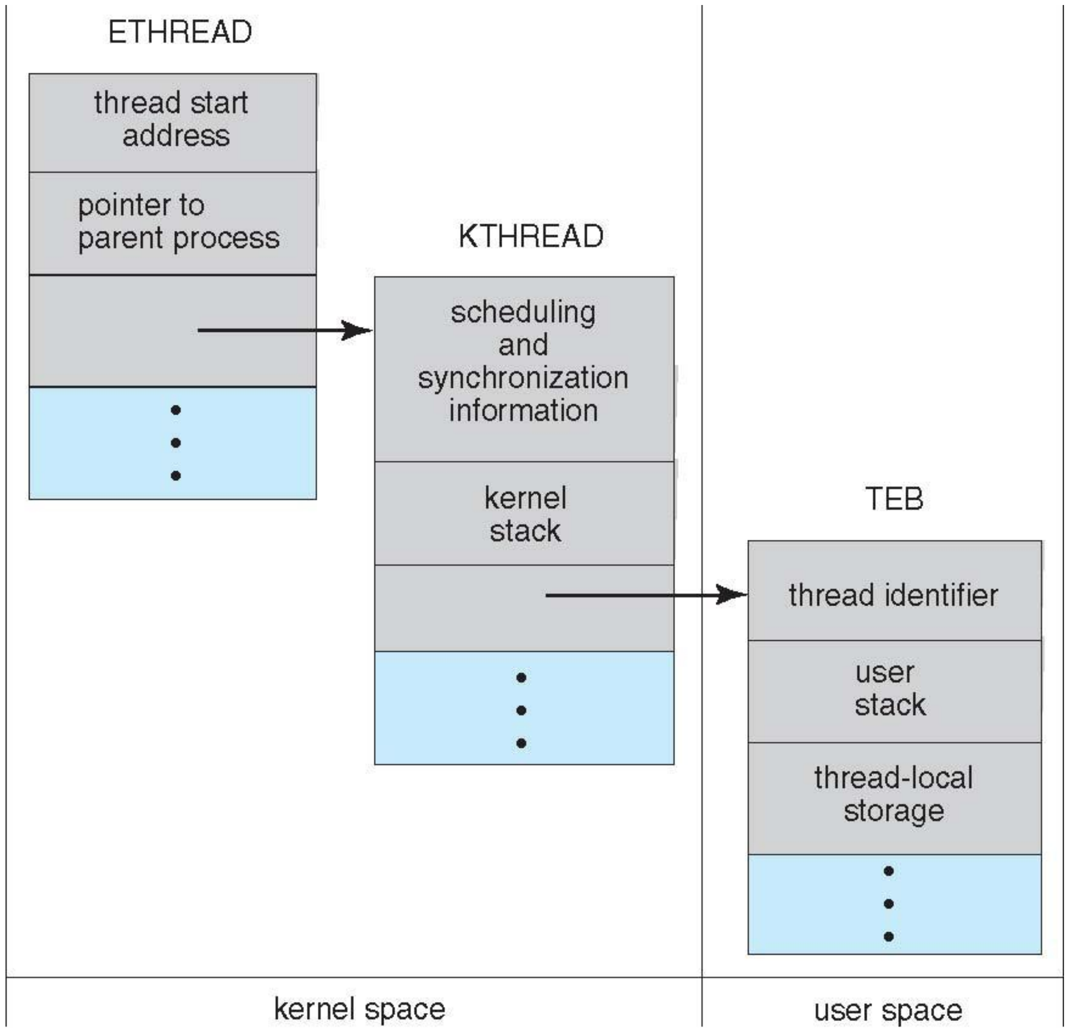
Windows XP Threads

- Implements the one-to-one mapping, kernel-level
- Each thread contains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area
- The register set, stacks, and private storage area are known as the context of the threads

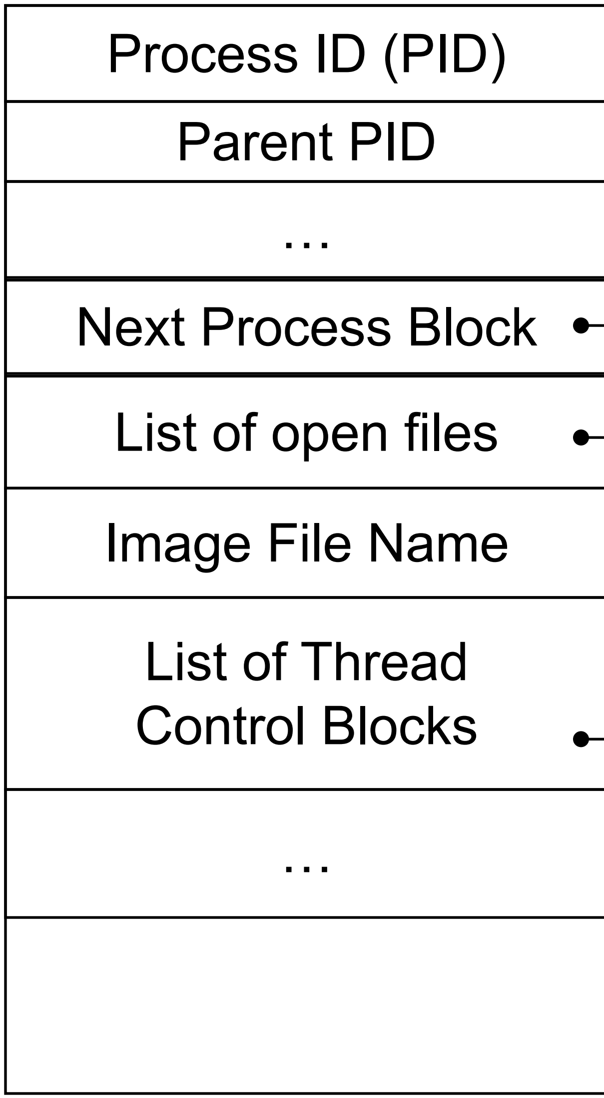
Windows XP Threads

- The primary data structures of a thread include:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)

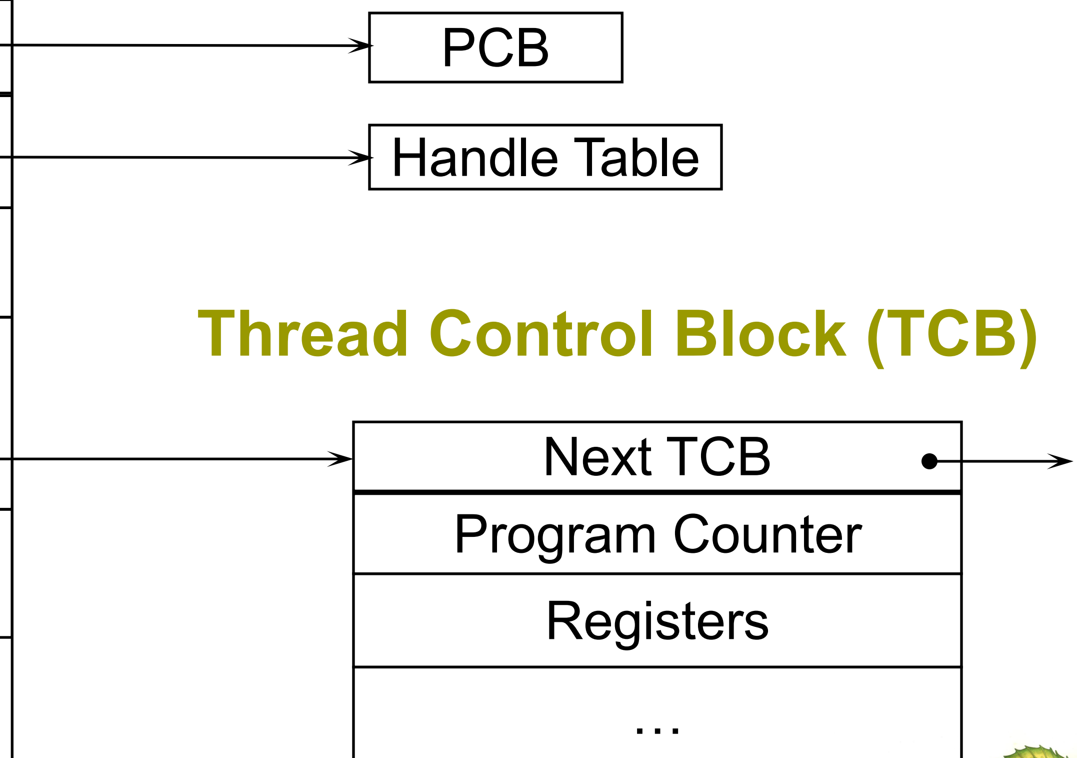
Windows XP Threads



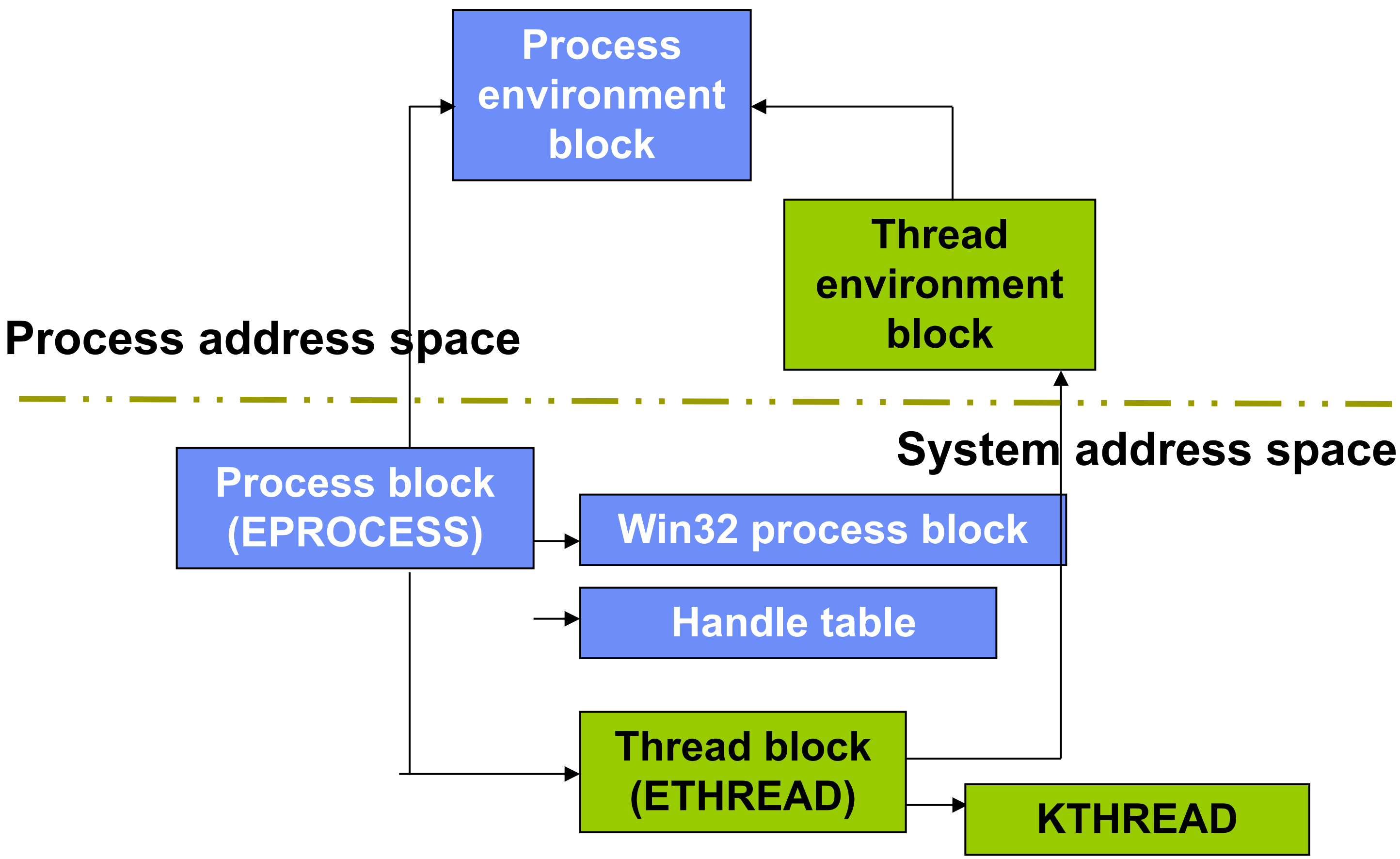
Process Control Block (PCB)



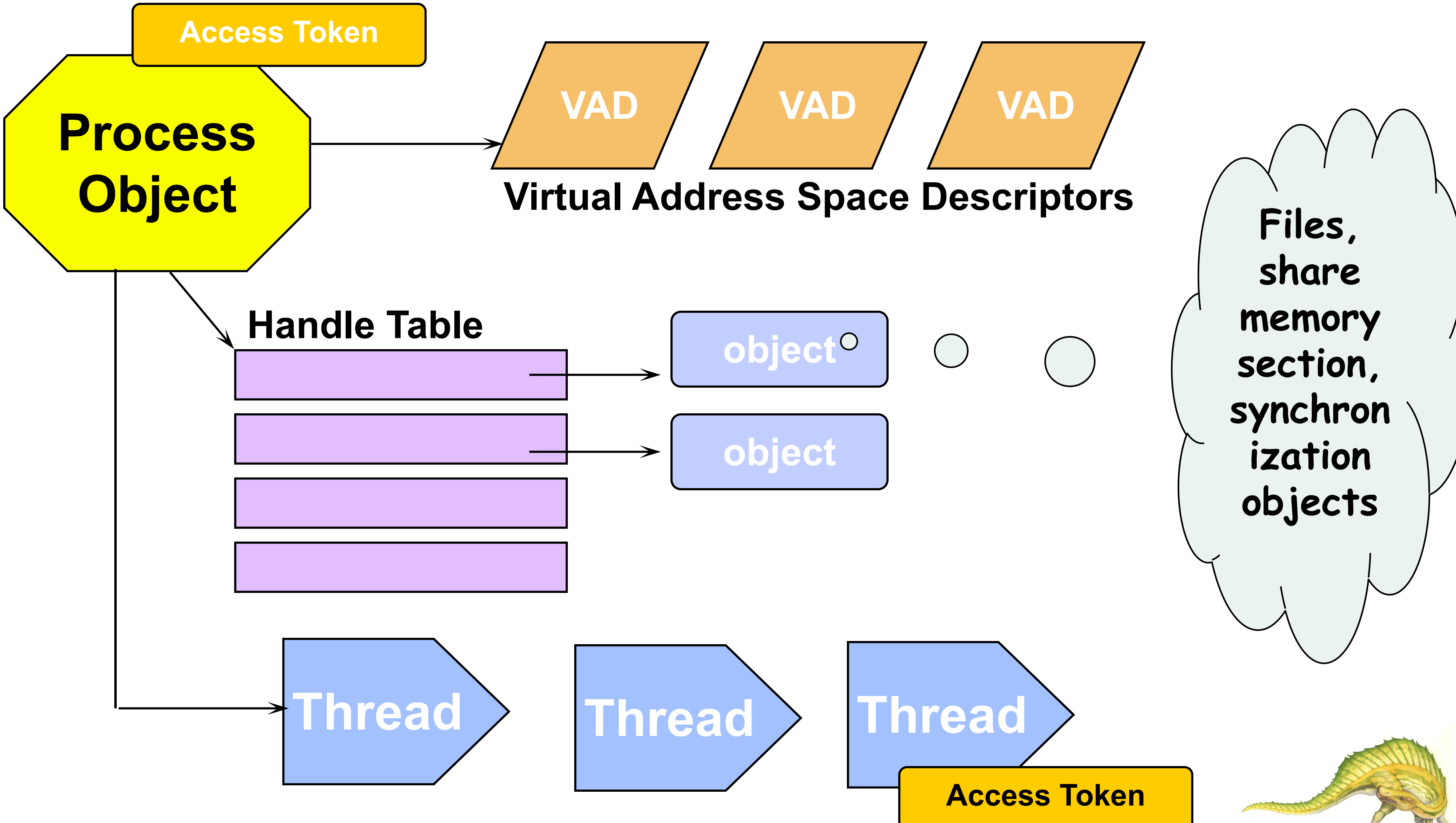
- This is an abstract view
- Windows implementation of PCB is split in multiple data structures



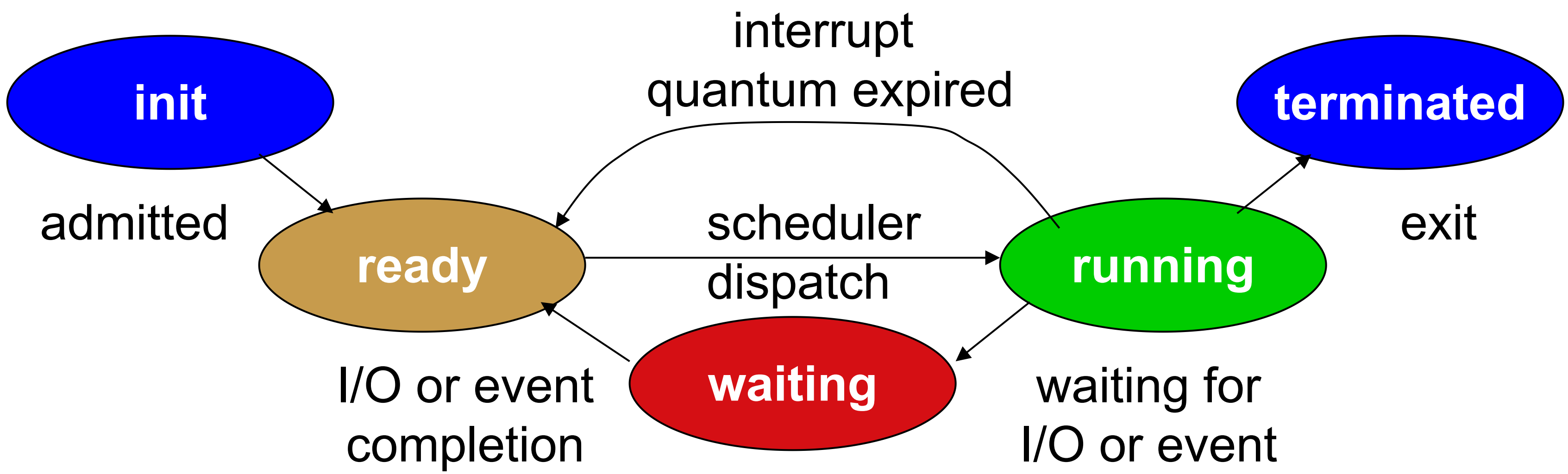
Windows Process and Thread Internals



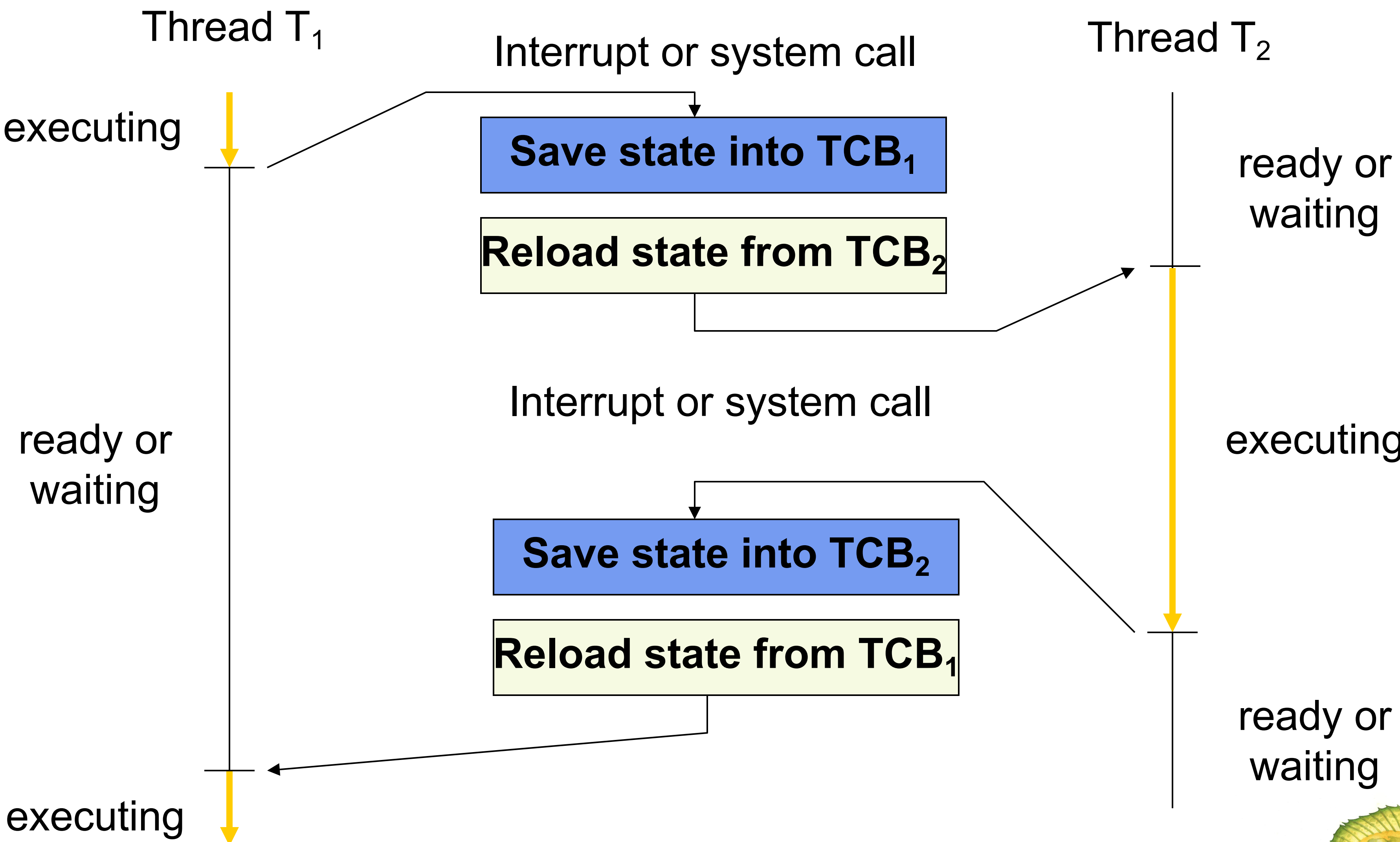
Windows Processes & Threads Internal Data Structures



Windows XP Threads



CPU Switch from Thread to Thread



Linux Threads

- Linux does not distinguish between processes and threads
- Linux refers to them as *tasks* rather than *threads*
- Thread creation is done through **clone()** system call
- **clone()** allows a child task to share the address space of the parent task (process)

Linux Threads

- When fork(), a task is created, along with a copy of all the associated data structures of the parent process
- When clone(), a task is created, no copying of data structures, the task points to the data structures of the parent

Linux Threads

- clone() allows a child task to share the address space of the parent task (process)
- Flags determine how much sharing is to take place between the parent and child
- If no flag is set when clone(), no sharing take place, resulting in functionality similar to fork()

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

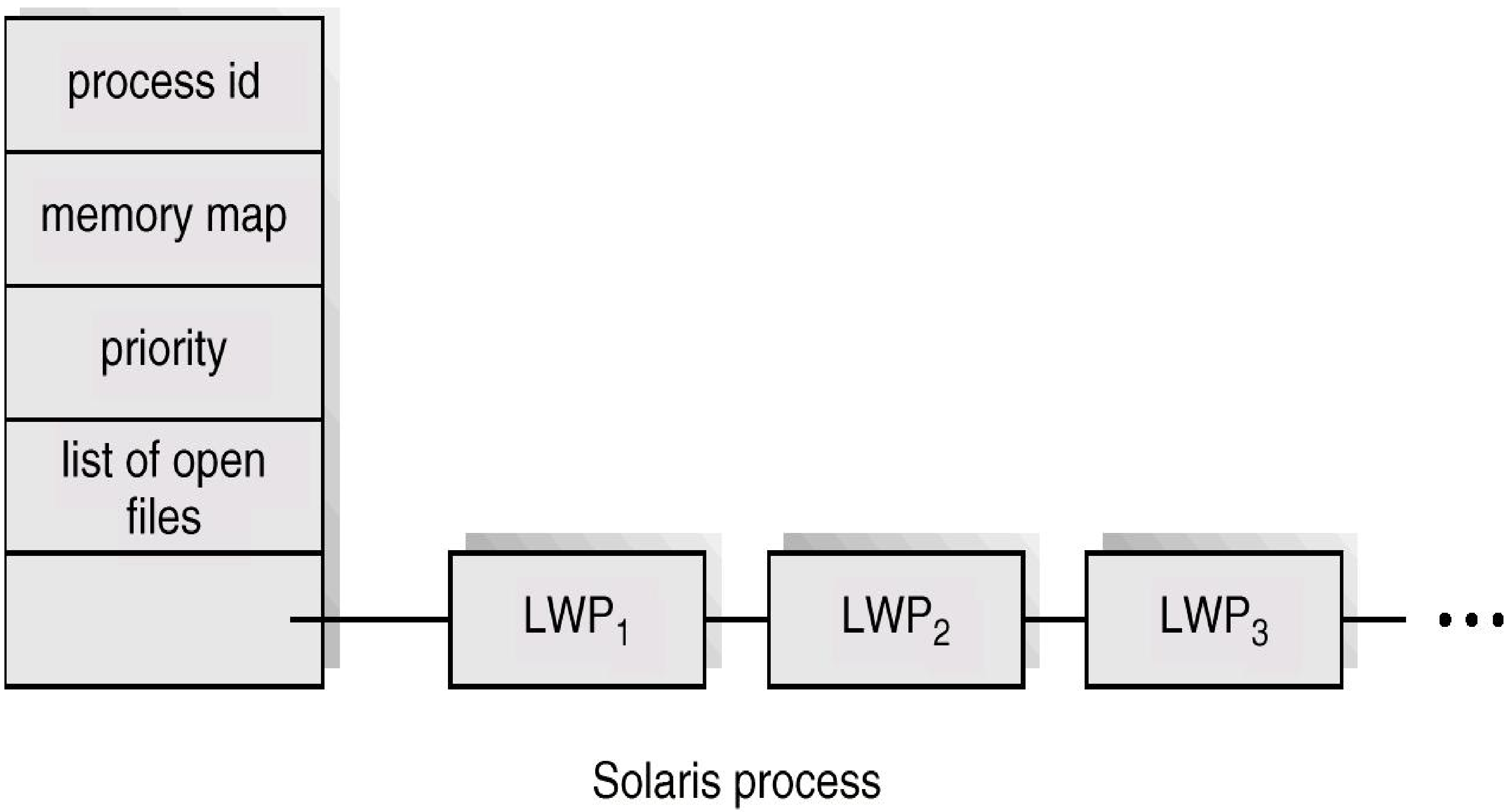
Solaris threads

- Solaris is a version of UNIX with support for threads at the kernel and user levels, symmetric multiprocessing, and real-time scheduling.

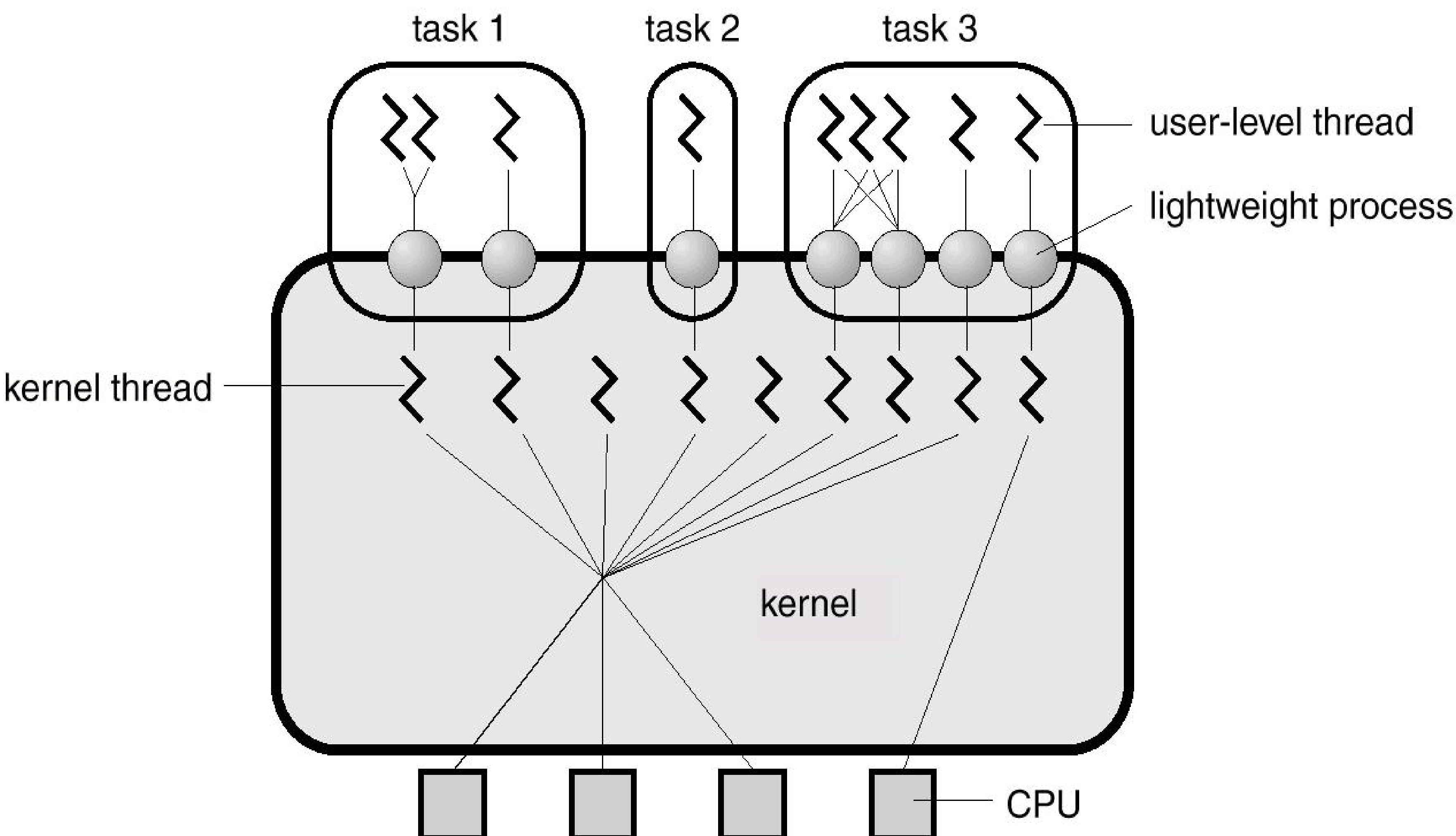
Threads Support in Solaris

- Solaris supports
  - Kernel threads
  - Lightweight Processes
  - User Level Threads
- LWP – (lightweight processes)intermediate level between user-level threads and kernel-level threads.

Solaris threads



Solaris threads



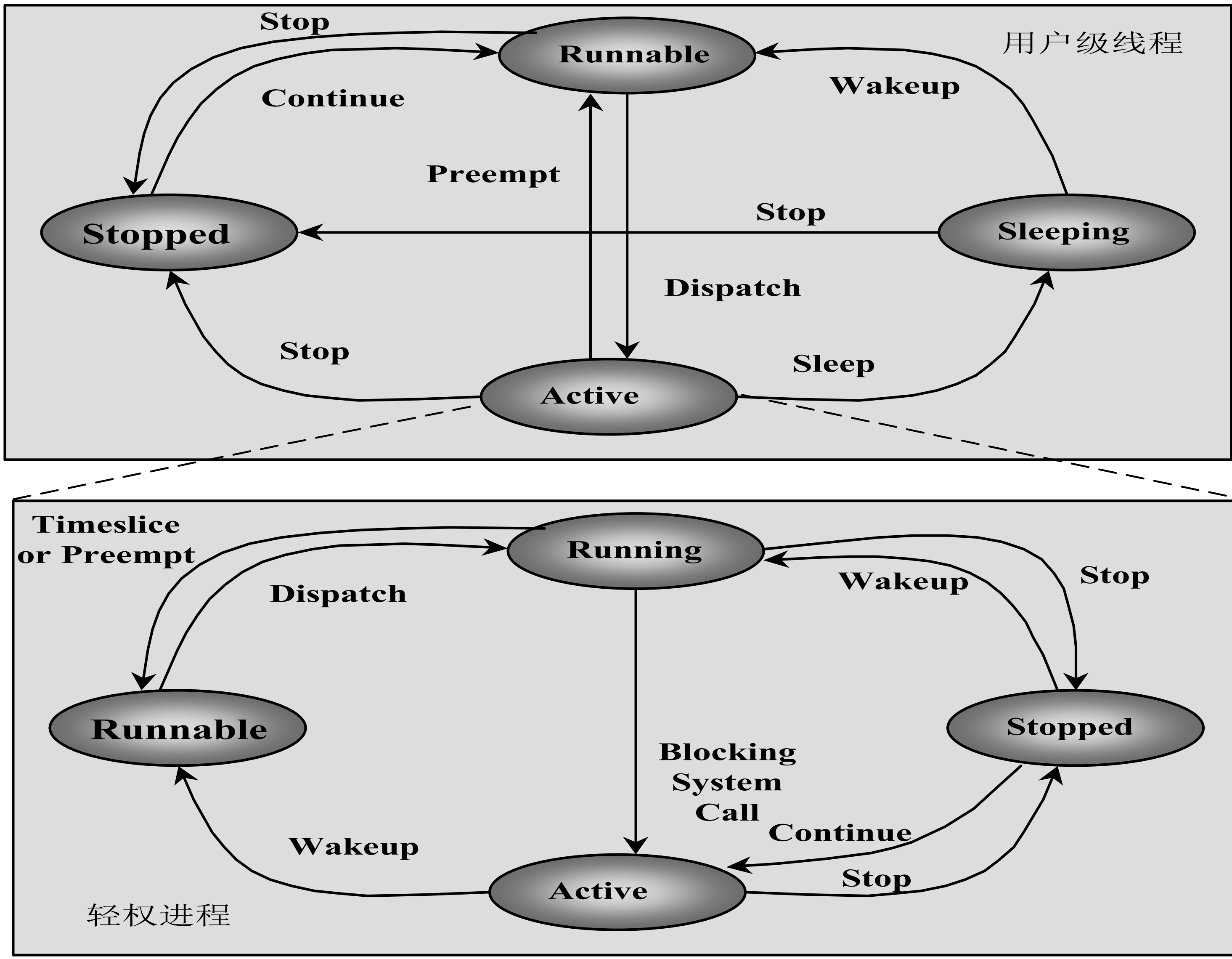
Solaris threads

- Kernel Threads
  - The only objects scheduled within the system
- LWP
  - Each process contains at least one LWP
  - The thread library multiplexes user-level threads on the pool of LWPs for the process
  - Without LWPs, user threads would contend at system call
- User Threads
  - Managed and scheduled by the thread library
  - May be either bound or unbound

LWP

- LWP is supported by kernel, it is kernel data structure
- LWP is virtual execution environment for user threads。
  - When a user thread issues a system call, its register states are put in the stack of LWP
- Each process contains at least one LWP
- Each LWP is connected to exactly one kernel-level thread.

Solaris: User Threads and LWP



Resources required for different threads in Solaris

- Kernel thread: small data structure and a stack; thread switching relatively fast.
- LWP: a register set for the user-thread it is running, accounting and memory information, switching between LWPs is relatively slow.
- User-level thread: only need stack and program counter; no kernel involvement means fast switching.
- Kernel only sees the LWPs that support user-level threads.

Solaris threads

- User level threads may be either bound or unbound
- Bound
  - A user thread is permanently attached to a LWP
- All unbound threads in an application are multiplexed onto the pool of available for the application
  - Threads are unbound by default .
- The thread library adjusts LWPs in the pool
  - The thread library ages LWPs and deletes them when they are unused for a long time, typically about 5 minutes.

End of Chapter 4