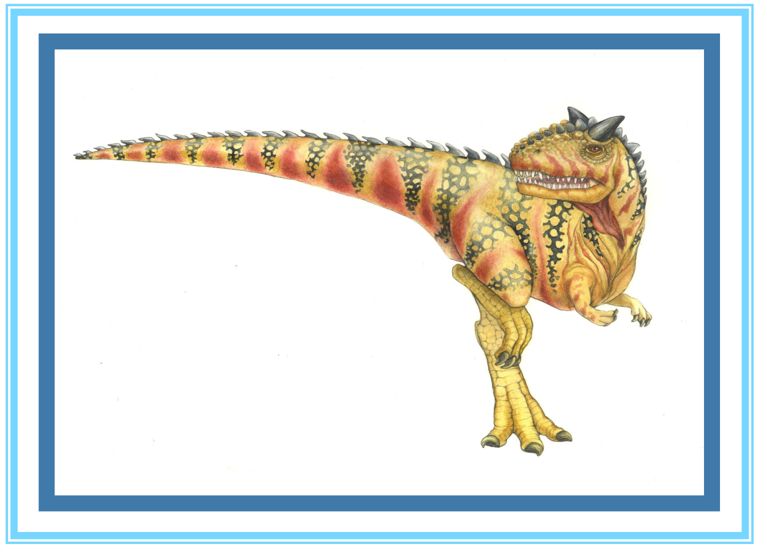
Chapter 4: Multithreaded Programming



Chapter 4: Multithreaded Programming

- Overview
- Multicore Programming
- Multithreading Models
- Threading Issues
- Operating System Examples

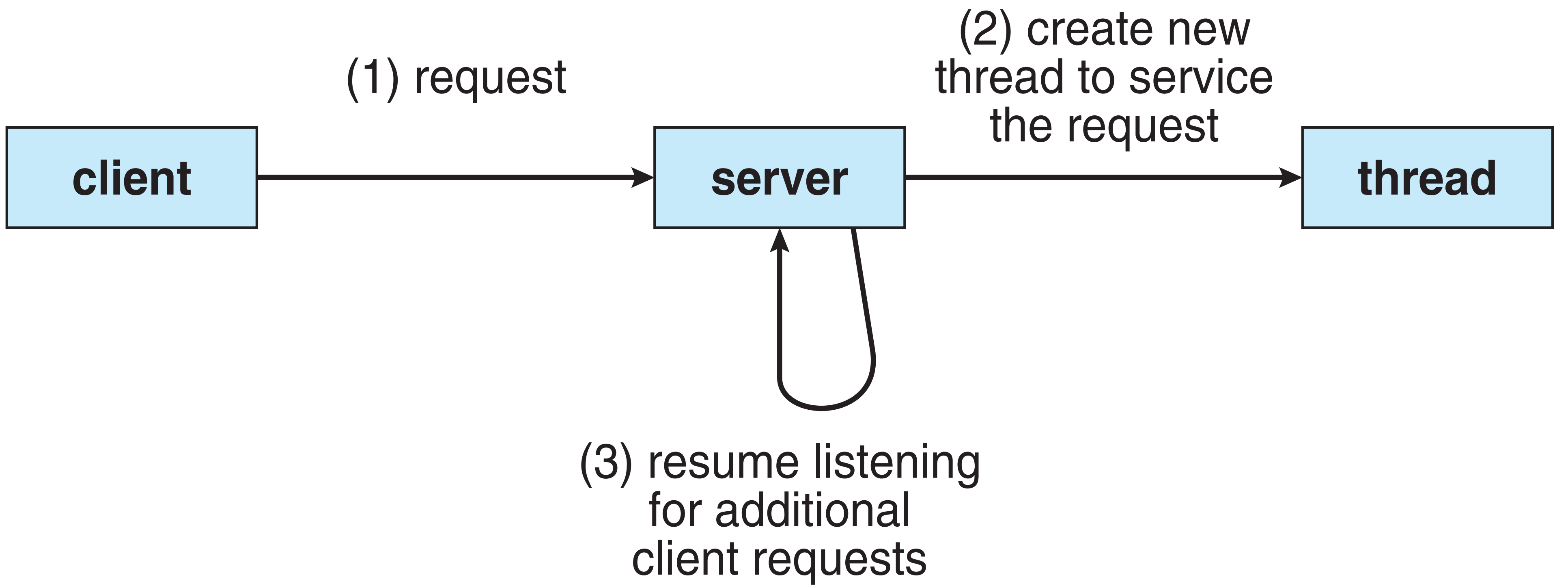
Objectives

- To introduce the notion of a thread—a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To examine issues related to multithreaded programming

Motivation

- Most modern applications or/and programs are multithreaded
- Threads run within an application or a process
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Multithreaded Server Architecture



- A single application may be required to perform several similar tasks. For example a busy web server may process thousands of web requests concurrently. Creating one process for each client request is cumbersome (resource-intensive) and time-consuming

- A single application may need to do multiple tasks. For example, a web browser (client) need to display images or text (one thread) while another thread retrieves data from the network

Benefits

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces

- **Resource Sharing –** threads with a process share resources of the process by default, easier than shared memory or message passing that must be explicitly arranged by the programmer

- **Economy –** thread creation is much cheaper than process creation, thread switching also has much lower overhead than context switching (switching to a different process)

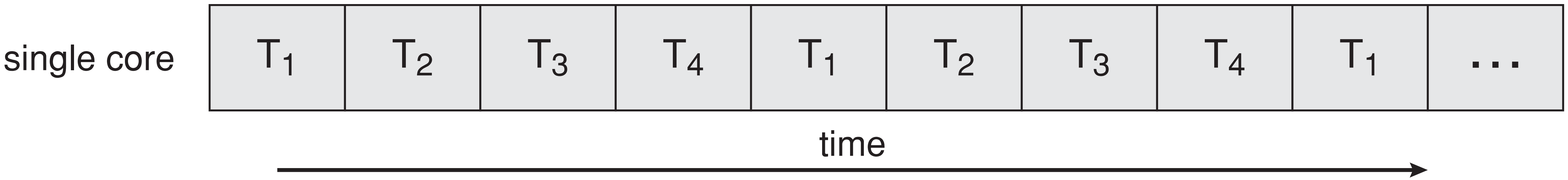
- **Scalability –** A process can take advantage of multiprocessor architectures by running multiple threads of the process simultaneously on different processors (CPUs).

Multicore Programming

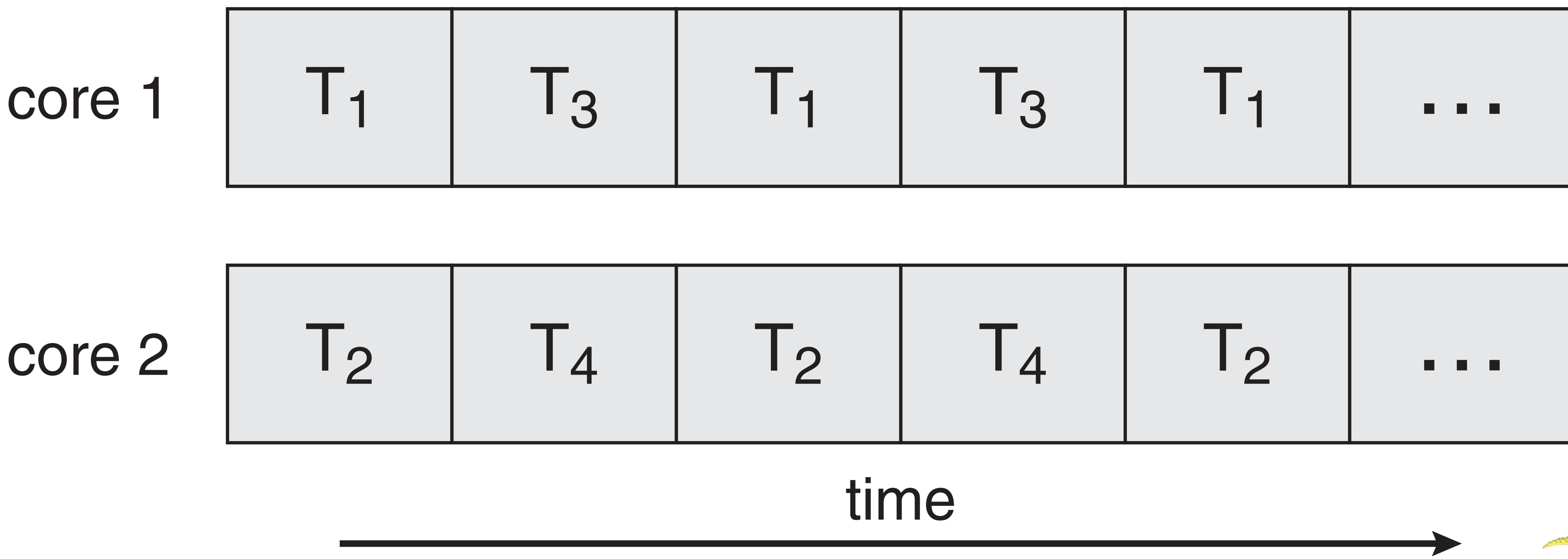
- **Multicore** or **multiprocessor** systems putting pressure on programmers to make better use of the multiple computing cores. Programming challenges in multicore systems include:
  - **Identifying tasks:** to divide applications into separate, concurrent tasks
  - **Balance:** tasks perform equal work of equal value
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- **Parallelism** implies a system can perform more than one task simultaneously
- **Concurrency** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency
- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation

Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**



- **Parallelism on a multi-core system:**



Amdahl's Law

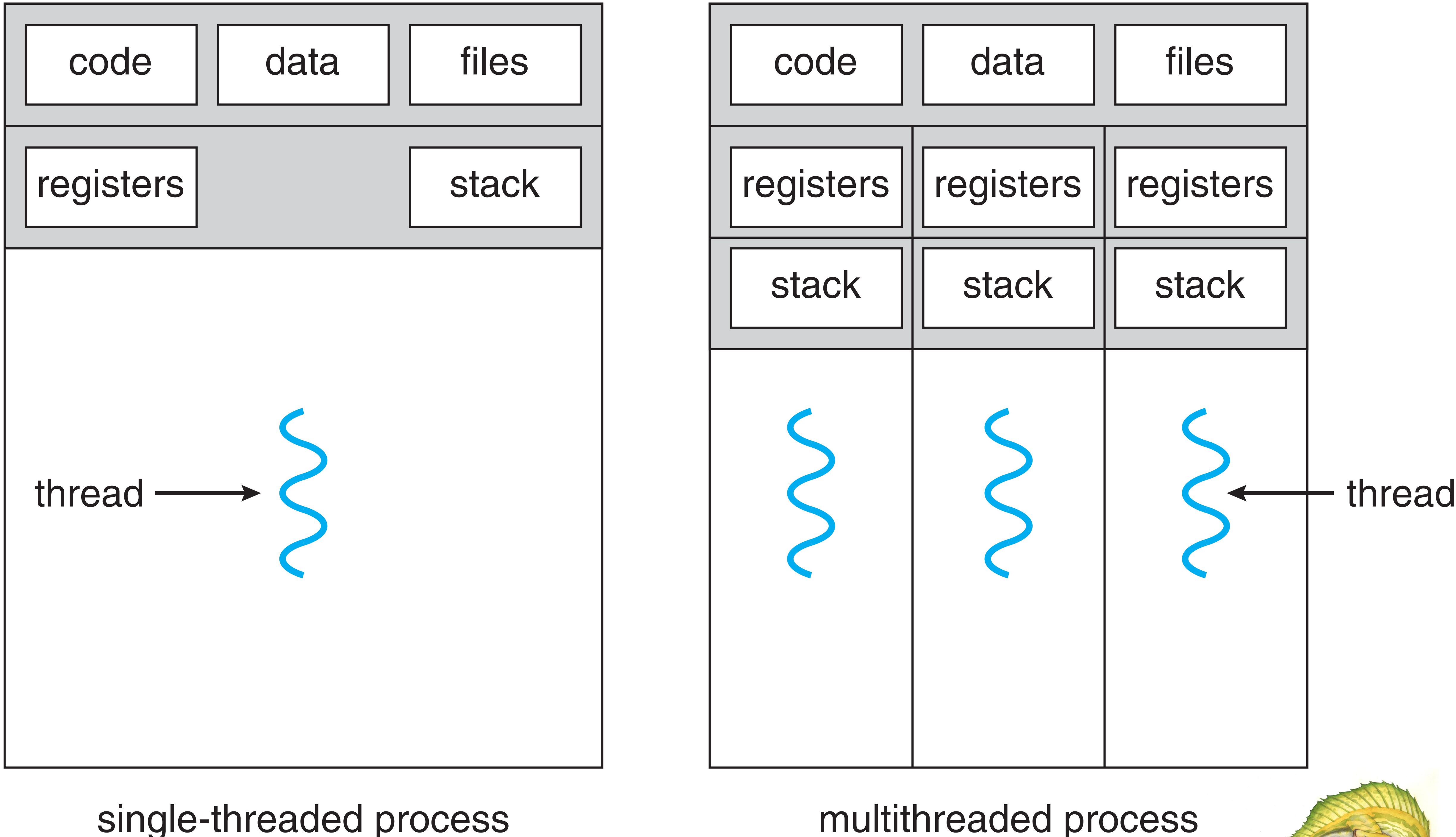
- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- *S* is serial portion
- *N* processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- If application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As *N* approaches infinity, speedup approaches 1 / *S*

Serial portion of an application has disproportionate effect on performance gained by adding additional cores

Single and Multithreaded Processes



Thread State

- Each Thread has a **Thread Control Block** (TCB)
  - Execution State: CPU registers, program counter, pointer to stack
  - Scheduling info: State (more later), priority, CPU time
  - Accounting Info:
  - Various Pointers (for implementing scheduling queues)
  - Pointer to enclosing process: PCB
- In Nachos: “thread” is a class that includes the TCB
- OS keeps track of TCBs in protected memory
  - Array, or Linked List, or …

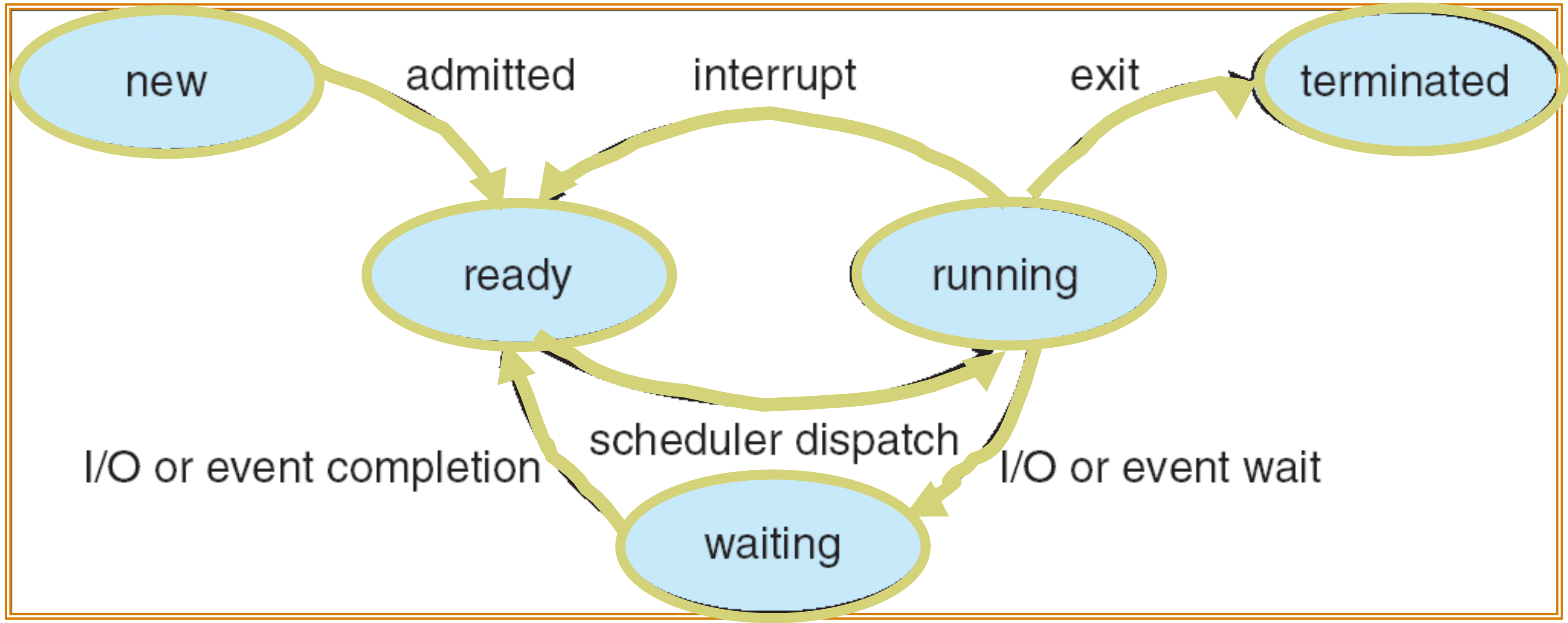
Thread State (Cont.)

- State shared by all threads in process/address space
  - Contents of memory (global variables, heap)
  - I/O state (file system, network connections, etc.)

- State “private” to each thread
  - Kept in TCB Thread Control Block
  - CPU registers (including, program counter)
  - Execution stack – what is this?

- Execution Stack
  - Parameters, temporary variables
  - Keep program counters while called procedures are executing

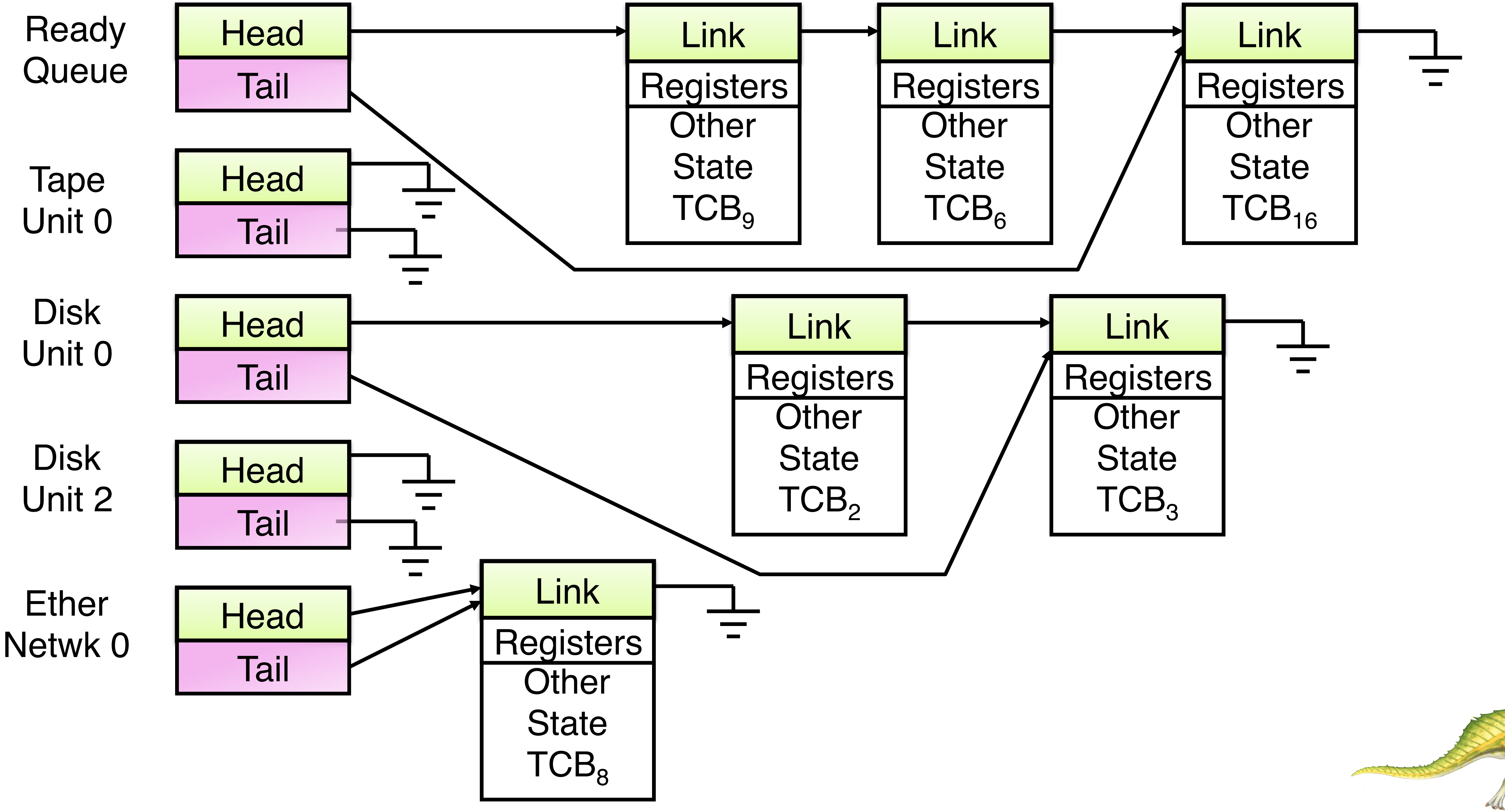
Lifecycle of a Thread



- As a thread executes, it changes state:
  - new: The thread is being created
  - ready: The thread is waiting to run
  - running: Instructions are being executed
  - waiting: Thread waiting for some event to occur
  - terminated: The thread has finished execution
- "Active" threads are represented by their TCBs
  - TCBs organized into queues based on their states

Ready Queue And Various I/O Device Queues

- Thread not running ⇒ TCB is in some scheduler queue
  - Separate queue for each device/signal/condition
  - Each queue can have a different scheduler policy



Examples of Multithreaded Programs

- Embedded systems
  - Elevators, Planes, Medical systems, Wristwatches
  - Single Program, concurrent operations
- Most modern OS kernels
  - Internally concurrent to deal with concurrent requests by multiple users
  - But no protection needed within kernel
- Database Servers
  - Access to shared data by many concurrent users
  - Also background utility processing must be done
- Network Servers
  - Concurrent requests from network
  - Again, single program, multiple concurrent operations
  - File server, Web server, and airline reservation systems
- Parallel Programming (More than one physical CPU)
  - Split program into multiple threads for parallelism

User Threads and Kernel Threads

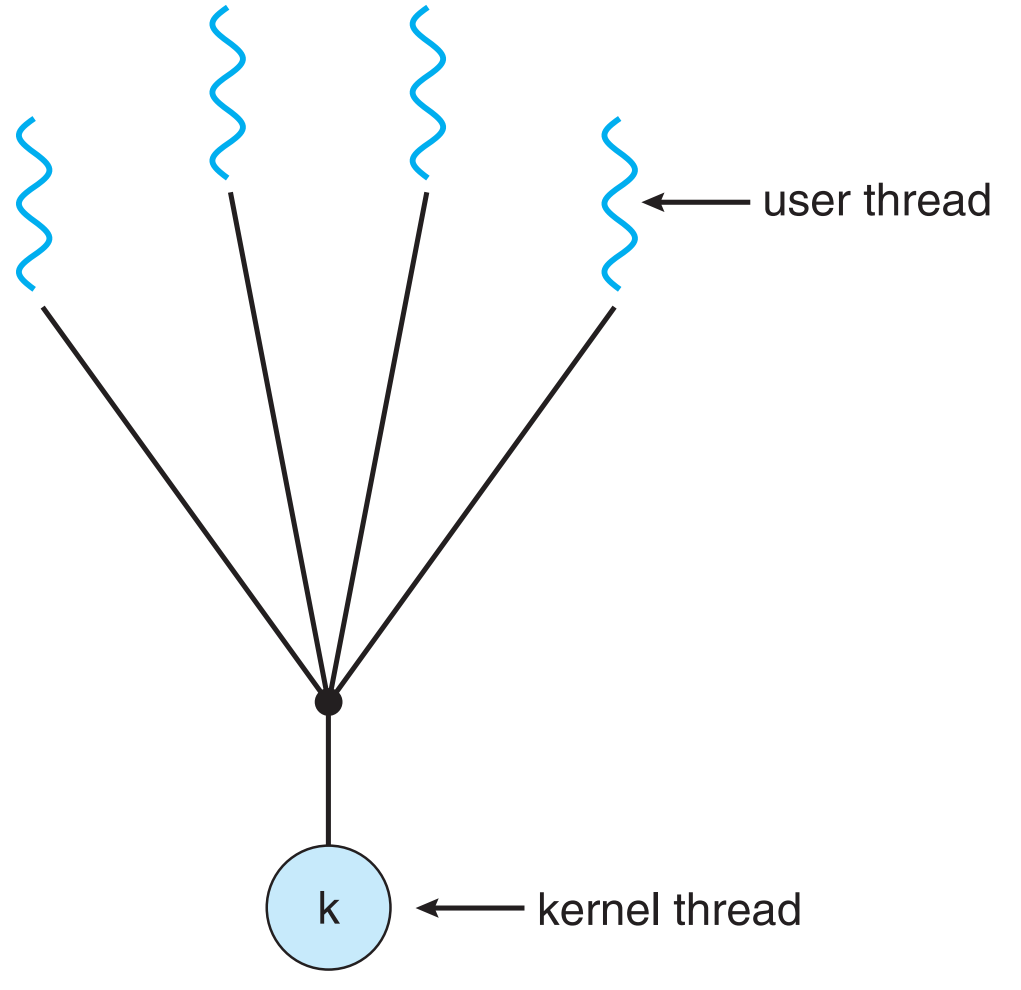
- Support for threads may be provided at either the user level, for **user threads,** or by the kernel, for **kernel threads**
- **User threads** - management done by user-level threads library without kernel support
- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java thread
- **Kernel threads** - supported by the kernel. Virtually all general-purpose operating systems support kernel threads, including:
  - Windows
  - Solaris
  - Linux
  - Mac OS X
- Ultimately, a relationship must exist between user threads and kernel threads.

Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

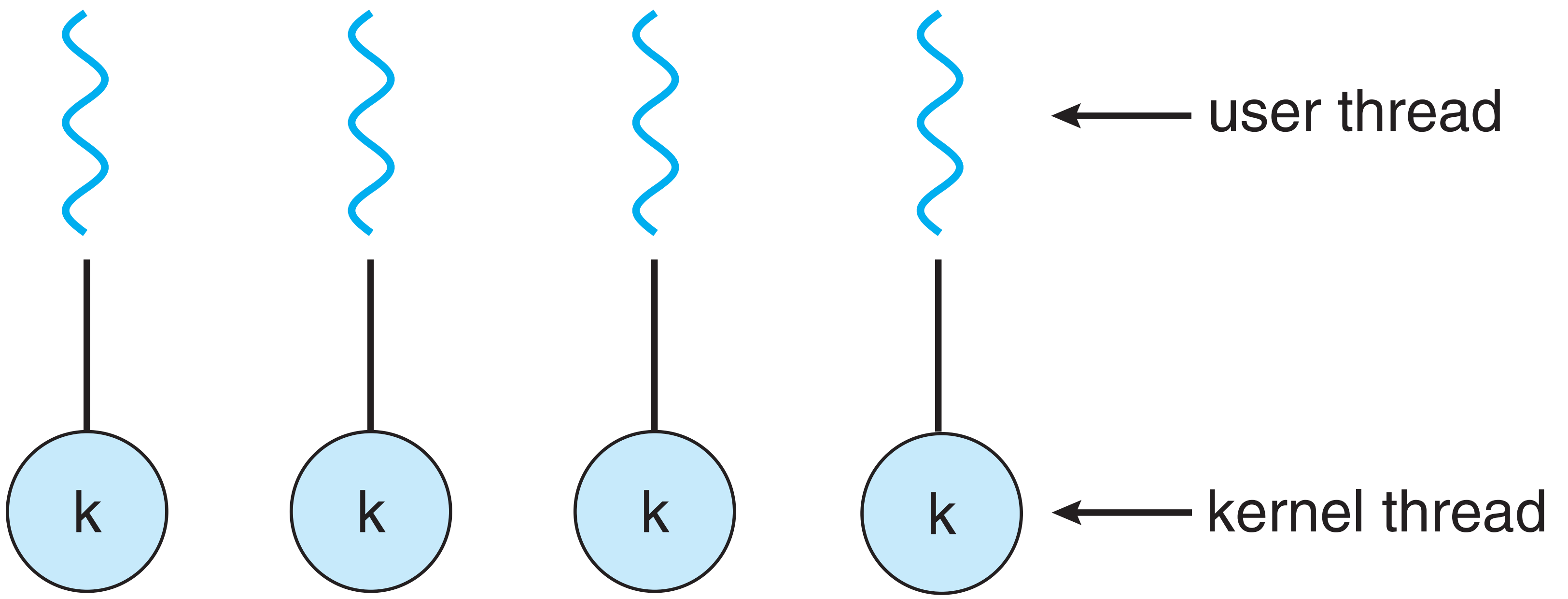
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on a multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



One-to-One

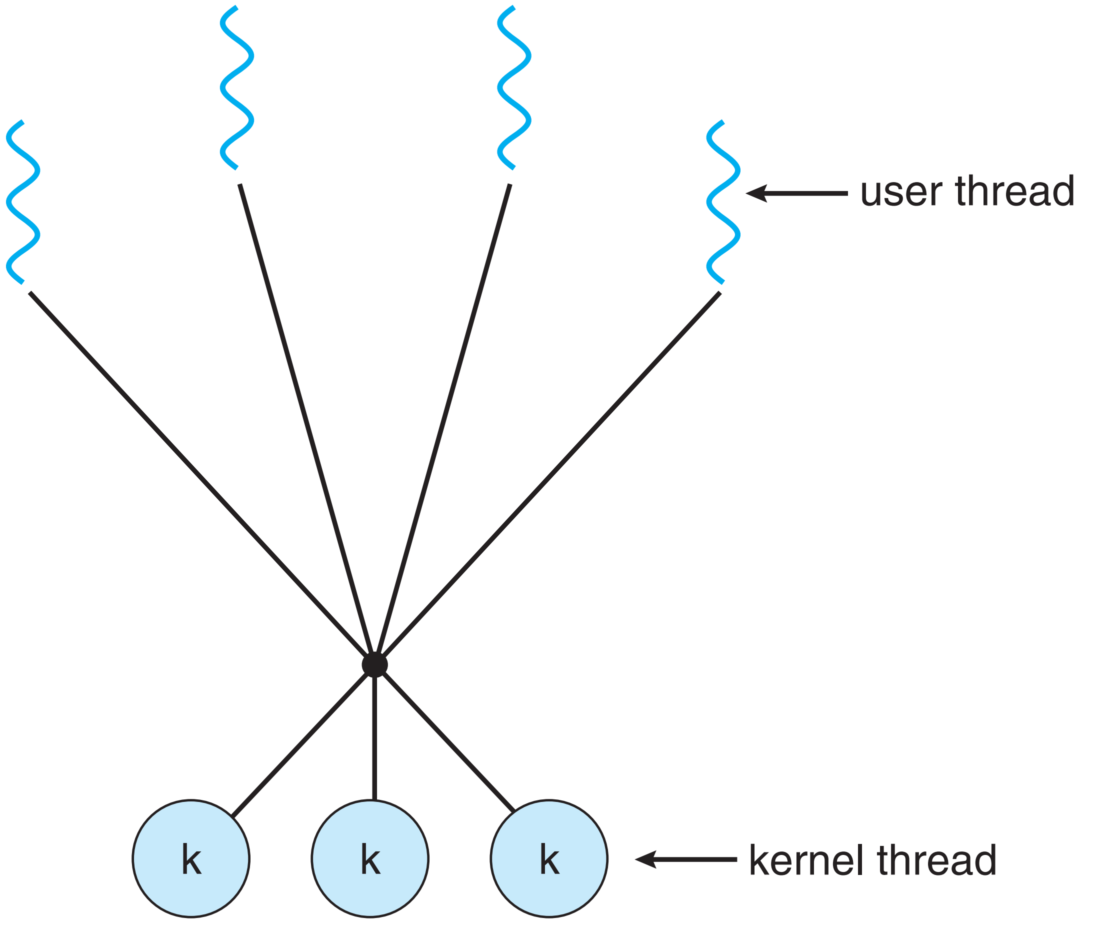
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead

- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later



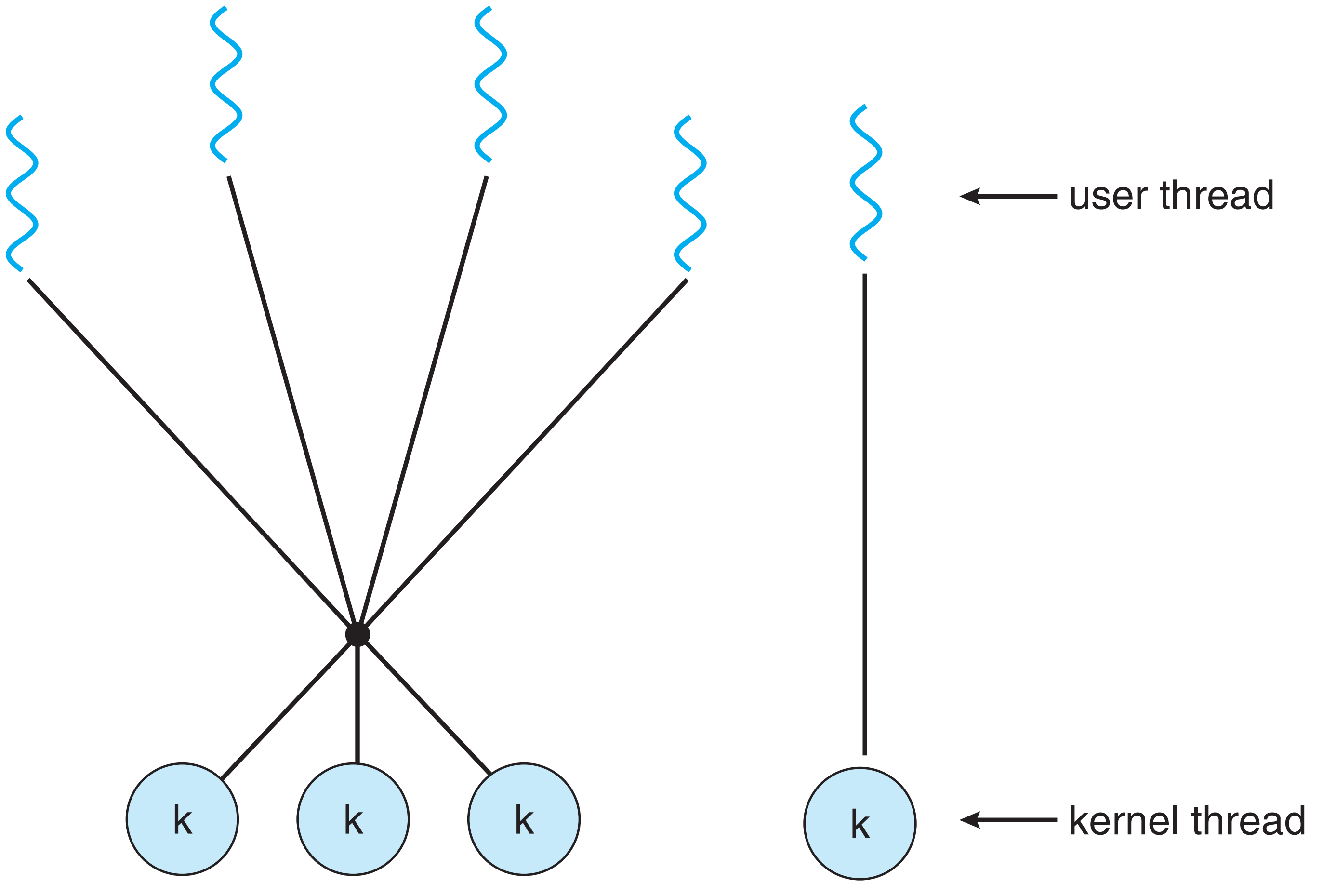
Many-to-Many Model

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



Two-level Model

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



Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space with no kernel support. This means that invoking a function in the library results in a local function call in user space, and not a system call
  - Kernel-level library supported directly by the OS
- Three main thread libraries are in use today:
  - POSIX Pthreads
  - Windows
  - Java

Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Signal handling
  - Synchronous and asynchronous
- Thread cancellation of target thread
  - Asynchronous or deferred
- Thread-local storage

Semantics of fork() and exec()

- Does **`fork()`** duplicate only the calling thread or all threads?
  - Some UNIX have two versions of fork
  - If exec() is called immediately after forking, duplicating all threads is unnecessary, as the program specified in the parameters to exec() will replace the entire process
- **`Exec()`** usually works as normal – replace the running process including all threads

Signal in UNIX

- **Signals** are in UNIX systems to notify a process that a particular event has occurred
- A signal may be received either *synchronously* or *asynchronously*, depending on the source of and the reason for the event being signalled. All signals follow the same pattern:
  - Signal is generated by the occurrence of a particular event
  - Signal is delivered to a process
  - Once delivered, the signal must be handled
- Examples of synchronous signal include illegal memory access and division of 0. Synchronous signals are delivered to the same process that performed the operation that caused the signal
- When a signal is generated by an event external to a running process, that process receives the signal asynchronously. Examples include terminating a process with specific keystrokes (such as `<control><C>`) and having a timer expire

Signal Handling

- A signal may be handled by one of the two possible handlers: has occurred
  - **A default signal hander**
  - **A user-defined signal handler**
- Every signal has default handler that kernel runs when handling signal
  - User-defined signal handler can override the default handler
  - For single-threaded, signal delivered to process
- Where should a signal be delivered a multi-threaded program?
  - 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
- The method for delivering a signal depends on the type of signal
  - Synchronous signals need to be delivered to the thread causing the signal, not other threads
  - Terminating a process signal should be sent to all threads within the process

Thread Cancellation

- **Thread cancellation** involves terminating a thread before it has completed. Example,
  - Multiple threads are concurrently searching through a database, one thread returns the result, the remaining threads might be cancelled

- Thread to be canceled is **target thread**
- Cancellation of a target thread may occur in two different scenarios:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
  - This involves in reclaiming the resource allocated to a thread, in which asynchronous cancellation might not be able to free up resource immediately

Thread-Local Storage

- **Thread-local storage** (**TLS**) allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations
- Similar to `static` data
  - TLS is unique to each thread

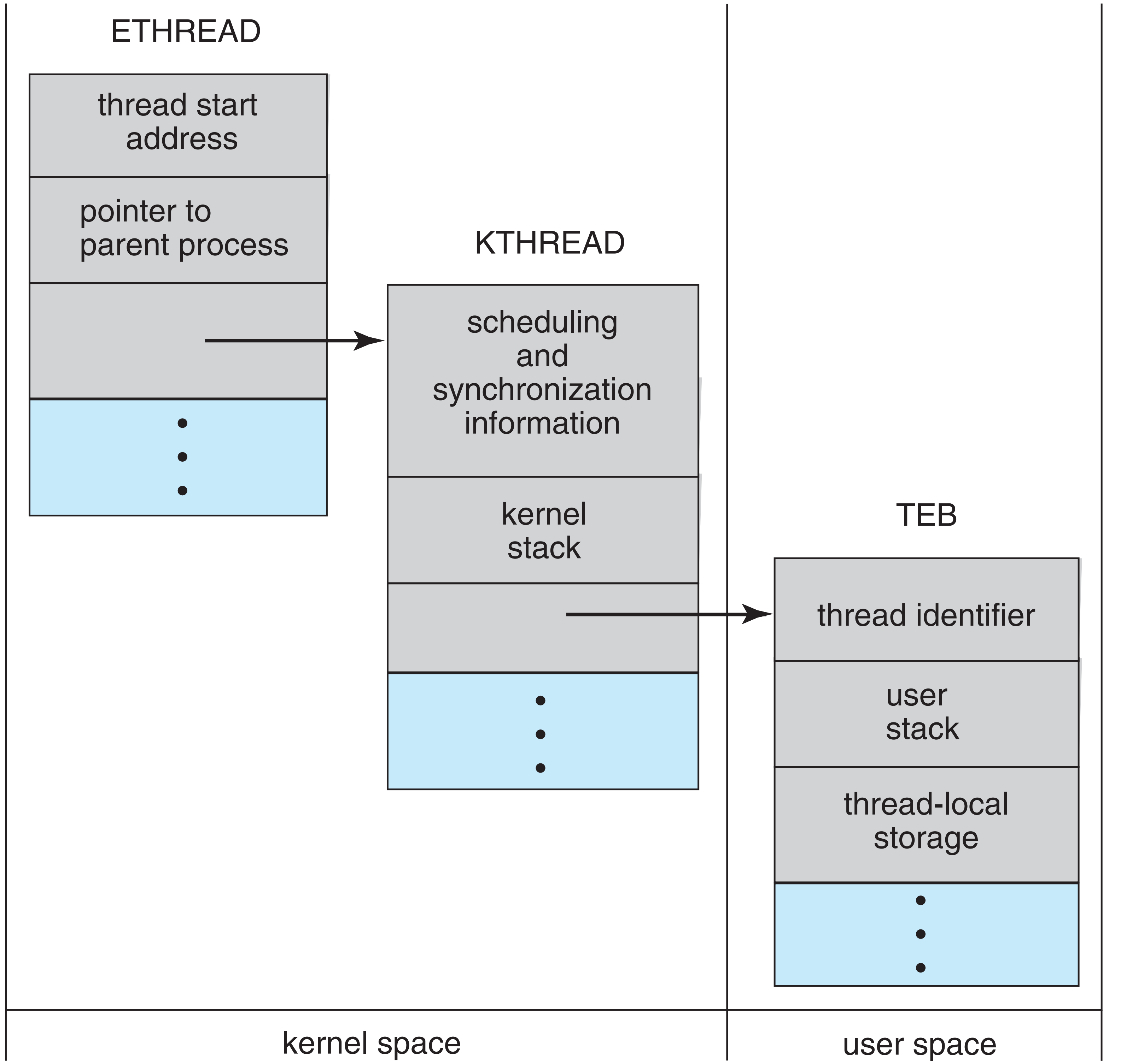
Operating System Examples

- Windows XP Threads
- Linux Thread

Windows Threads

- Windows implements the Windows API – for Win 98, NT, 2000, Win XP, and Window 7
- Implements the one-to-one mapping, kernel-level
- Each thread contains
  - A thread ID uniquely identifying the thread
  - Register set representing the status of the processor
  - Separate user and kernel stacks for when thread runs in user mode or kernel mode
  - Private data storage area used by run-time libraries and dynamic link libraries (DLLs)
- The register set, stacks, and private storage area are known as the **context** of the thread
- The primary data structures of a thread include:
  - ETHREAD (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space
  - KTHREAD (kernel thread block) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space
  - TEB (thread environment block) – thread ID, user-mode stack, thread-local storage, in user space

Windows XP Threads Data Structures



Linux Threads

- Linux refers to processes and threads as ***tasks*** rather than ***threads***
- Thread creation is done through **`clone()`** system call
- **`clone()`** allows a child task to determine how to share the address space of the parent task (process)
  - Flags control behavior

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.