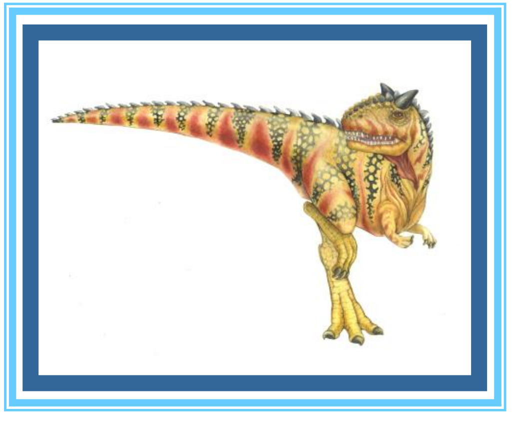
Chapter 4: Threads & Concurrency



Chapter 4: Threads

- 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

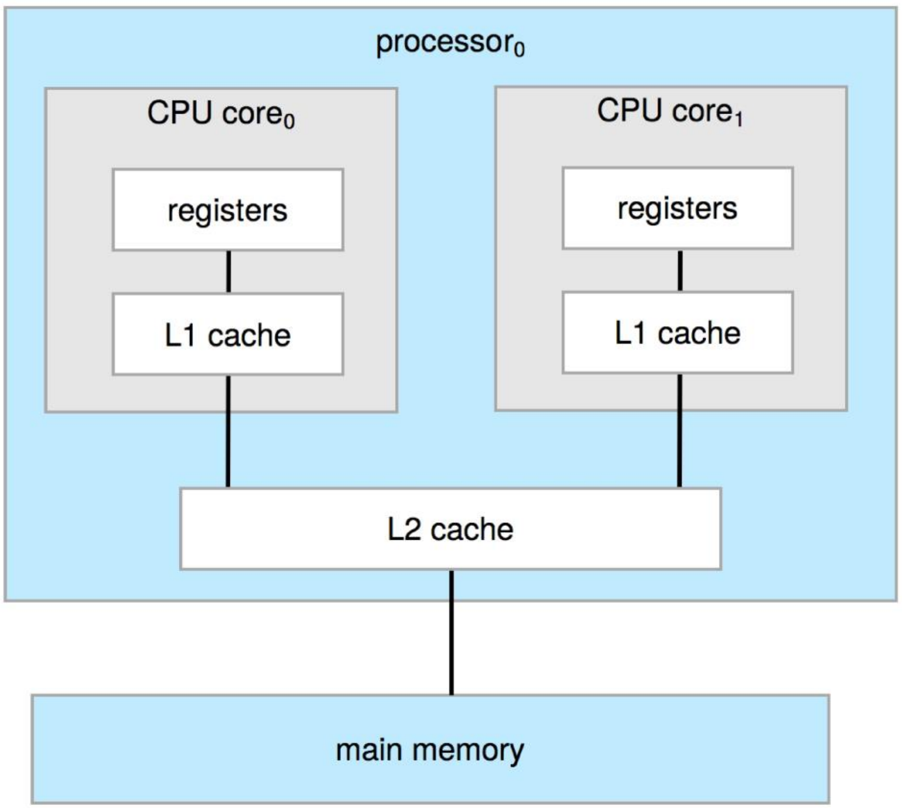
- Many modern applications are multithreaded
- Threads run within application
- Multiple tasks within an application (process) can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation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

Benefits

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multicore architectures

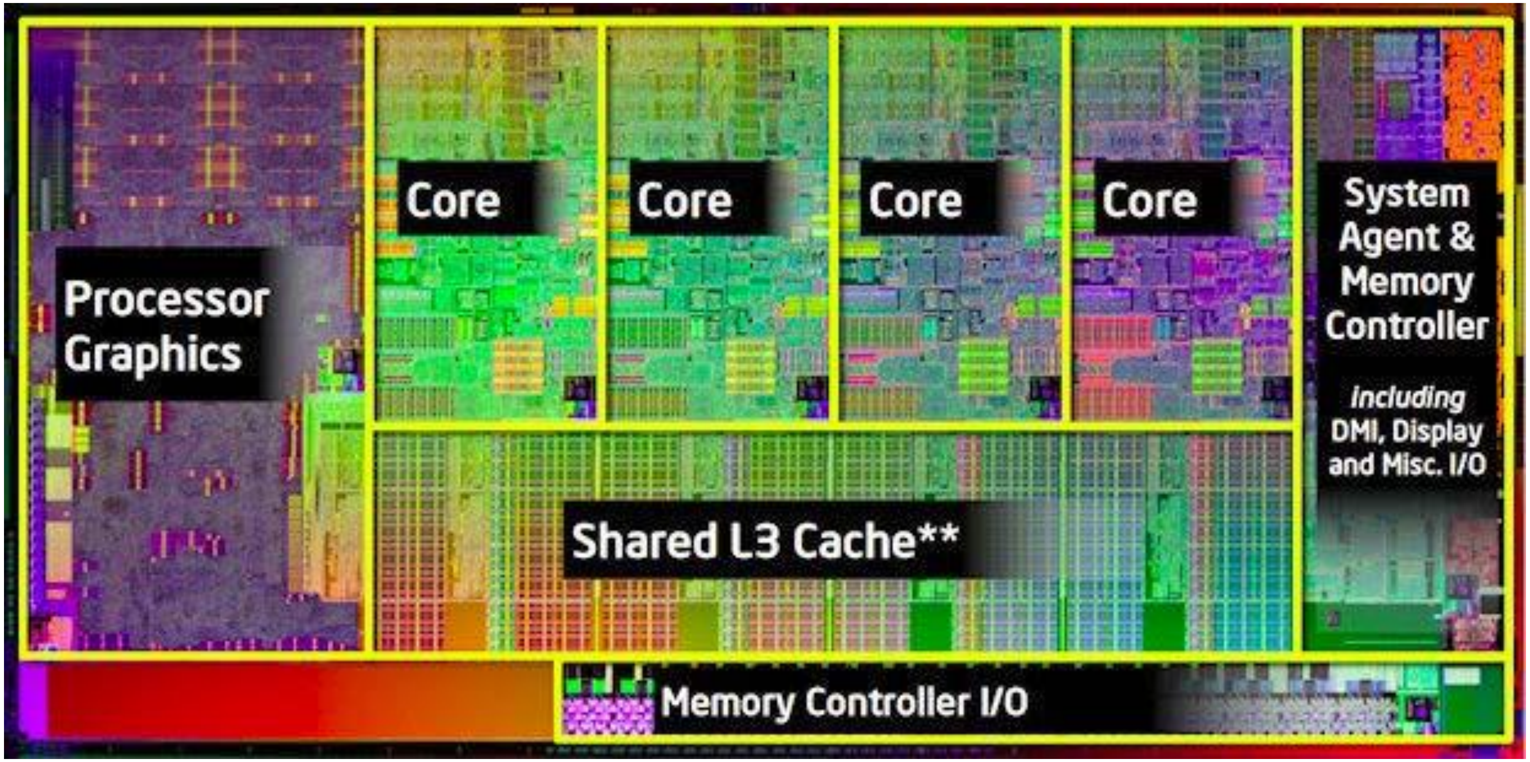
A Multi-Core Design

- Multi-chip and **multicore**
  - On-chip communications faster
  - Power consumption can be reduced



A Modern Processor

- Intel Sandy Bridge



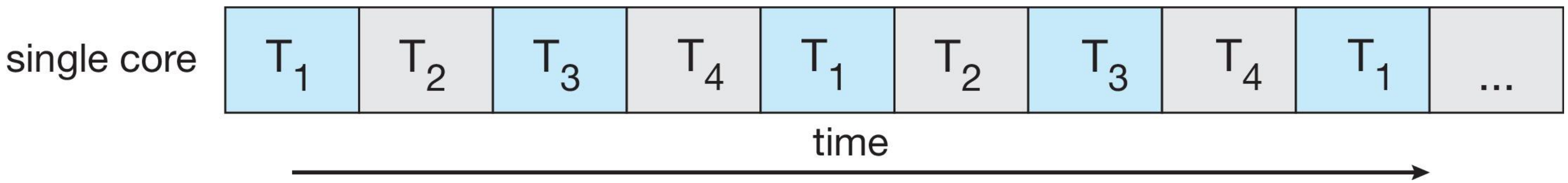
How to program this?

Multicore Programming

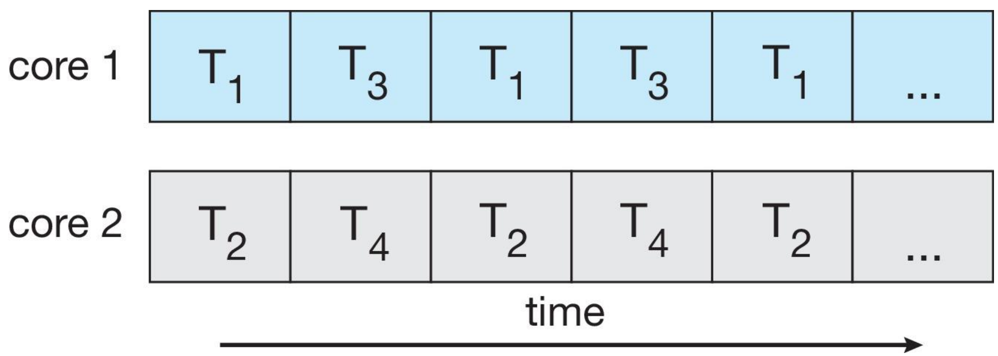
- Multicore or multiprocessor systems, programming challenges include:
  - Dividing activities – how to divide tasks
  - Balance – each task perform “equal” amount of work
  - Data splitting
  - Data dependency – synchronization
  - Testing and debugging – different path of executions
- Parallelism a system can perform more than one task simultaneously
  - 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 supports more than one task making progress
  - Multiplex over the time
  - Single processor / core, scheduler providing concurrency

Concurrency vs. Parallelism

- Concurrent execution on single-core system:

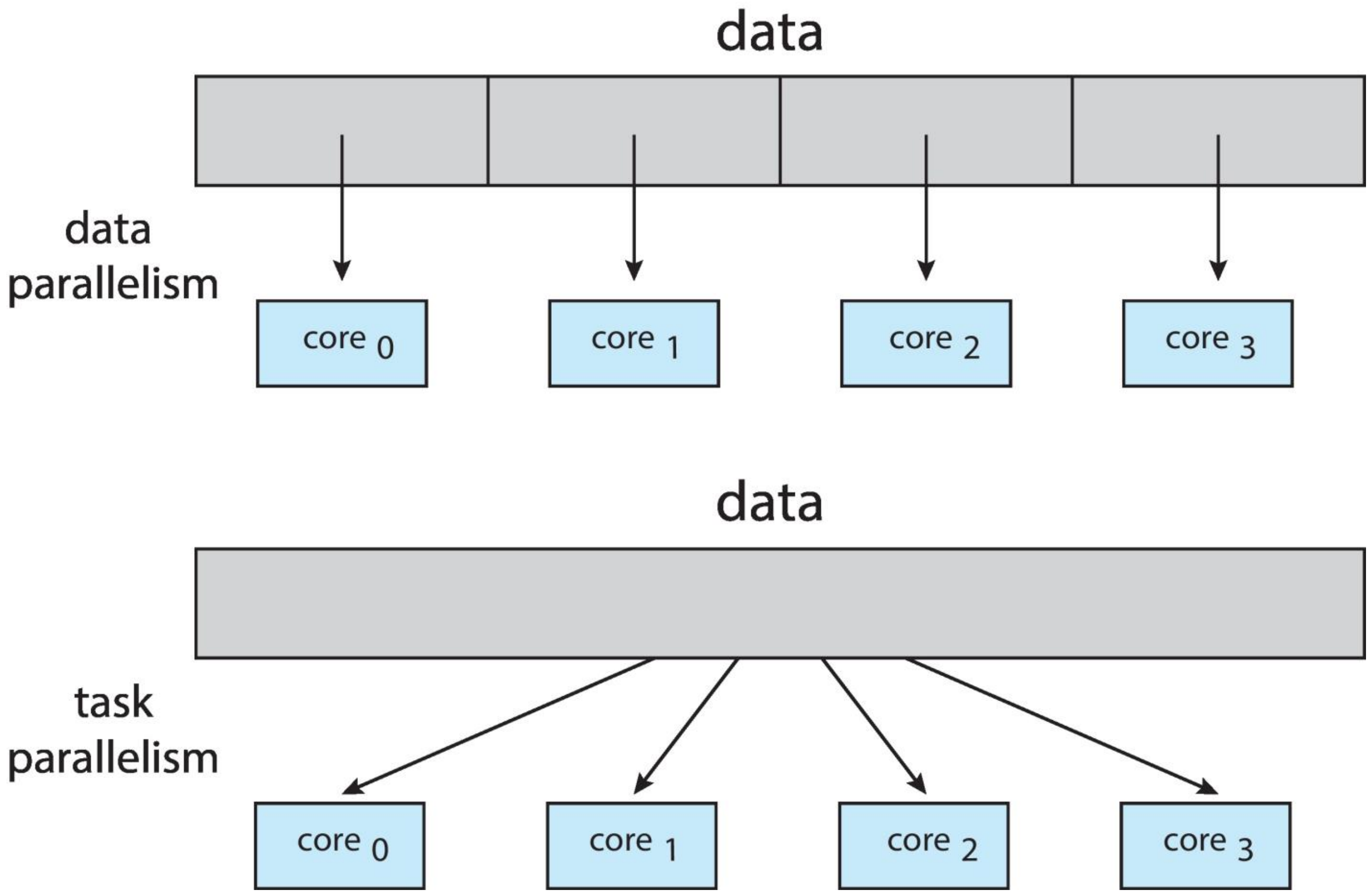


- Parallelism on a multi-core system:



Data and Task 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



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

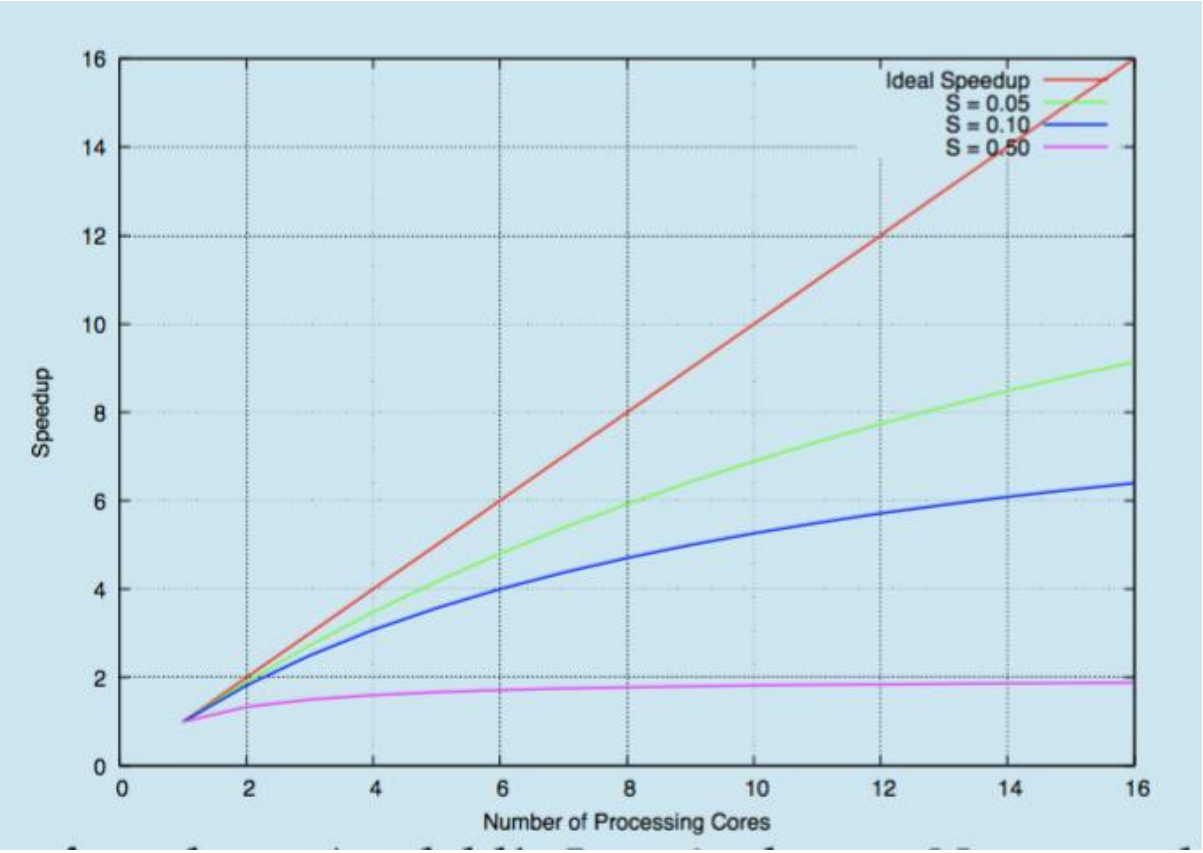
$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- That is, 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

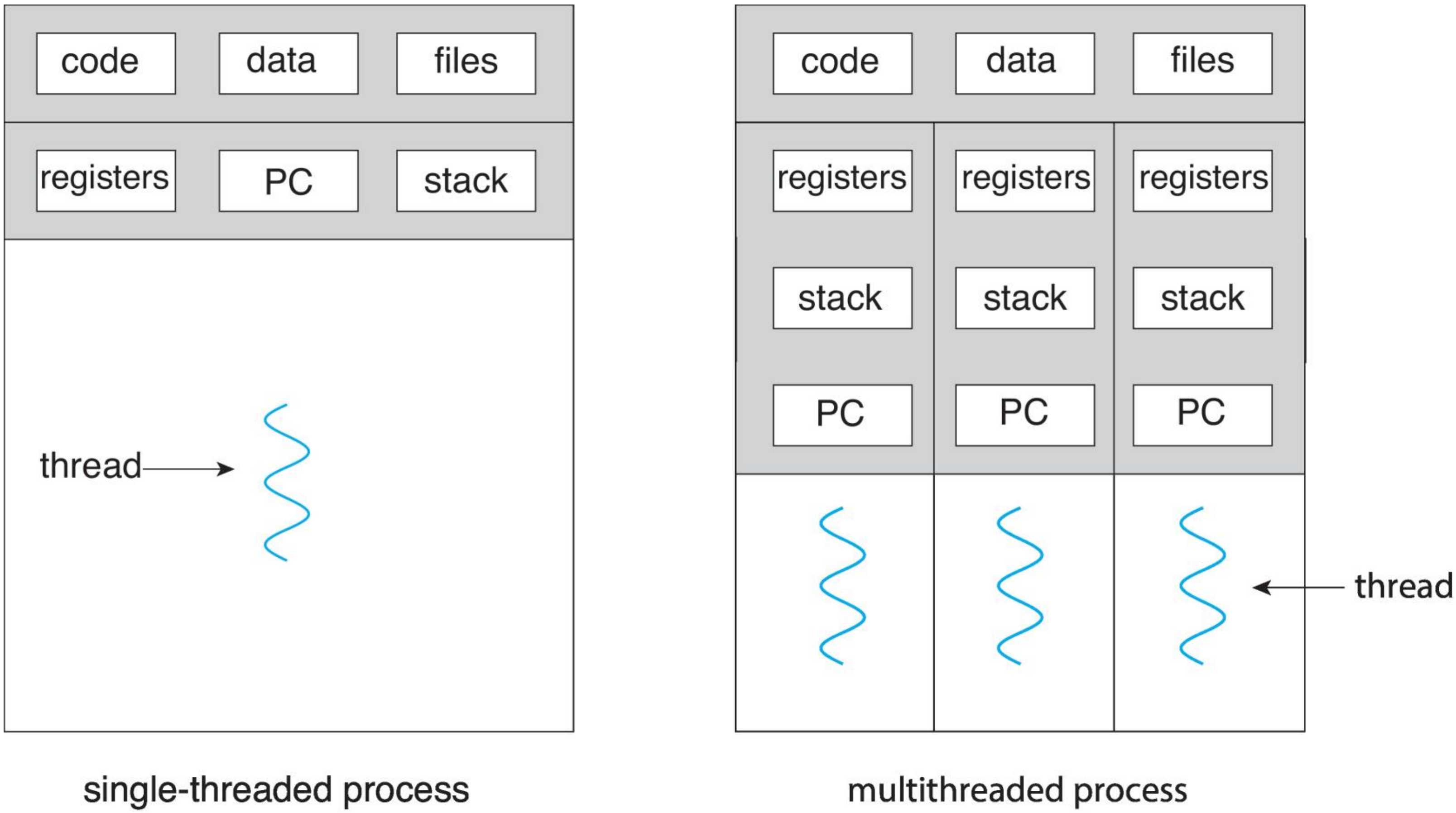
- But does the law take into account contemporary multicore systems?

Amdahl's Law



Ideal Speedup
S = 0.05
S = 0.10
S = 0.50
Speedup
Number of Processing Cores
0
2
4
6
8
10
12
14
16

Single and Multithreaded Processes



Thread

- Thread: Single unique execution context ("lightweight process")
  - Program Counter, Registers, Execution Flags, Stack
  - A thread is executing on a processor when it is resident in the registers.
  - PC register holds the address of executing instruction in the thread
  - Registers hold the root state of the thread (other state in memory)
- 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 region
  - Array, or Linked List, or …

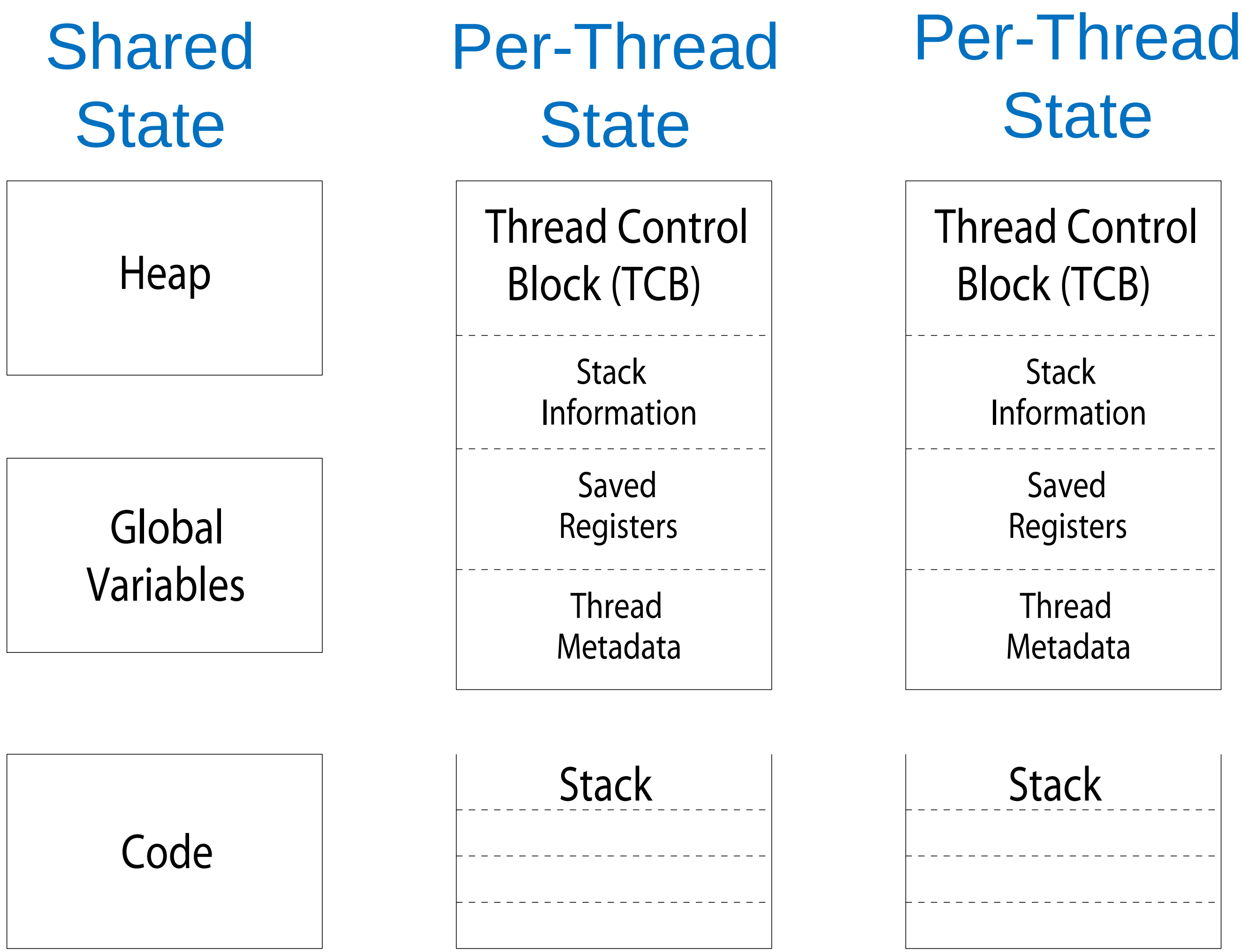
Thread State

- Threads encapsulate concurrency: "Active" component
- Address spaces encapsulate protection: "Passive" part
  - Keeps buggy program from thrashing the system

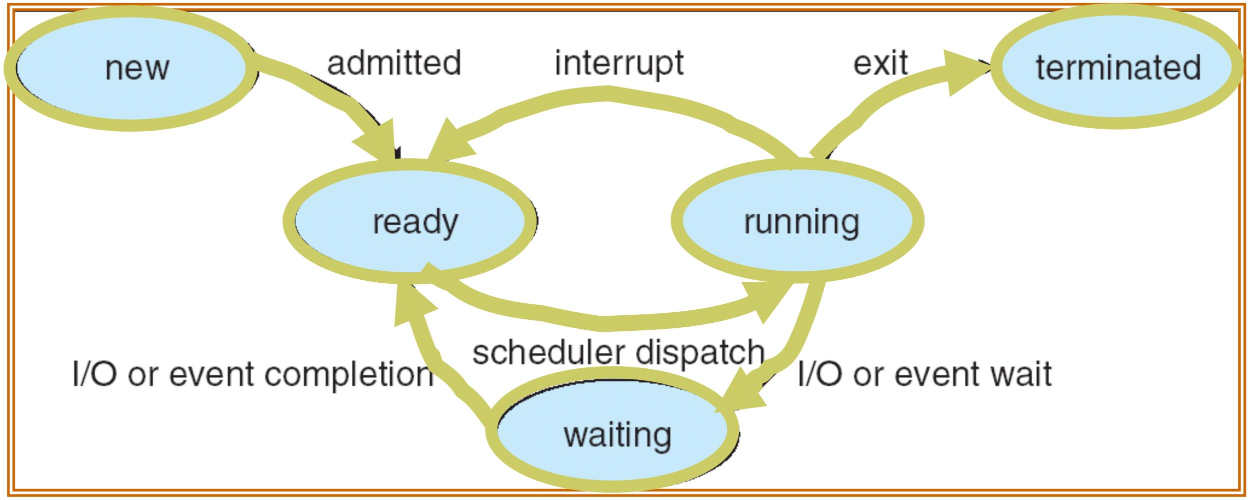
- State shared by all threads in process/address space
  - Contents of memory (global variables, heap)
  - I/O state (file descriptors, network connections, etc.)

- State "private" to each thread
  - Kept in TCB ≡ Thread Control Block
  - CPU registers (including, program counter)
  - Execution stack (parameters, temporary variables, PC saved)

Shared vs. Per-Thread State



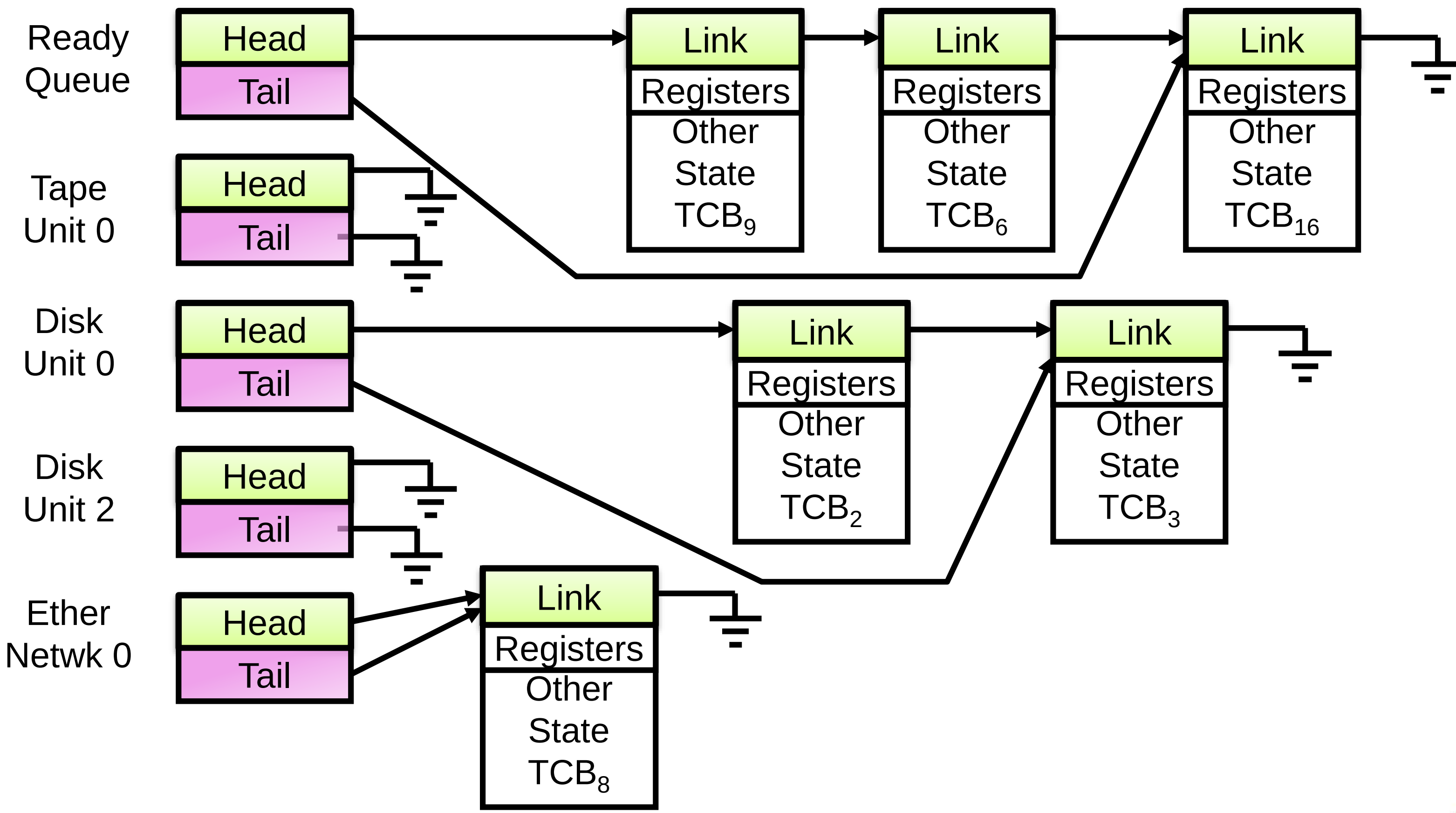
Lifecycle of a Thread



- 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

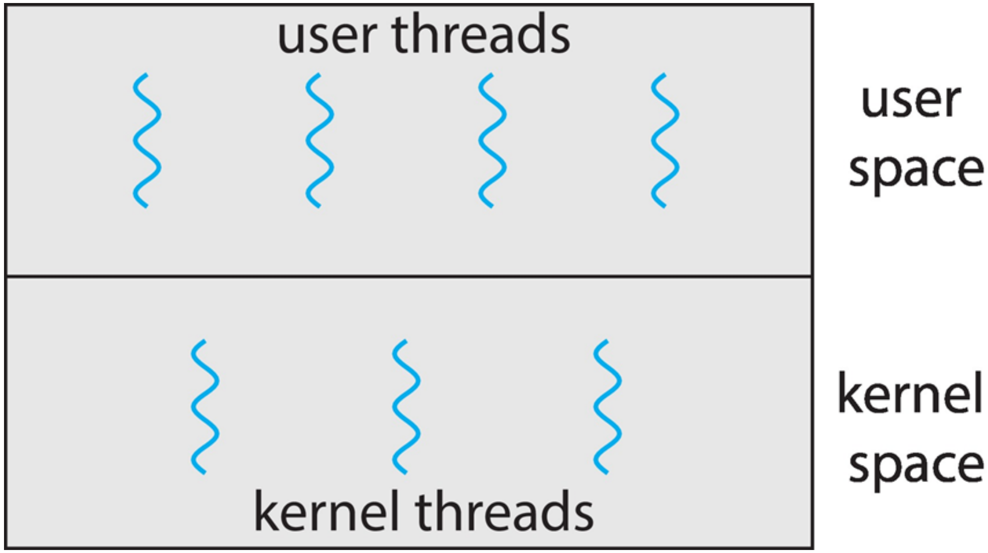


Examples of Multithreaded Programs

- 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 a program into multiple threads for parallelism

User Threads and Kernel Threads

- User threads - management done by user-level threads library
- Three primary thread libraries:
  - POSIX Pthreads, Windows threads and Java threads
- Kernel threads - supported by the operating system
- Examples – virtually all general purpose operating systems, including:
  - Windows, Linux, Mac OS X iOS, Android

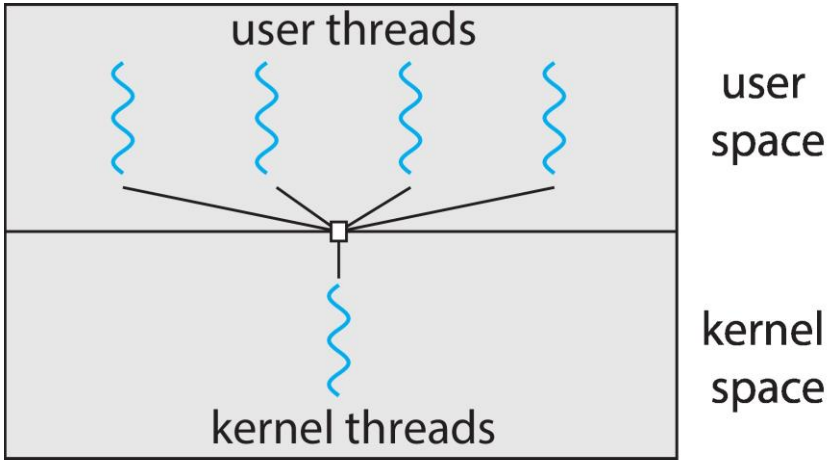


Multithreading Models

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

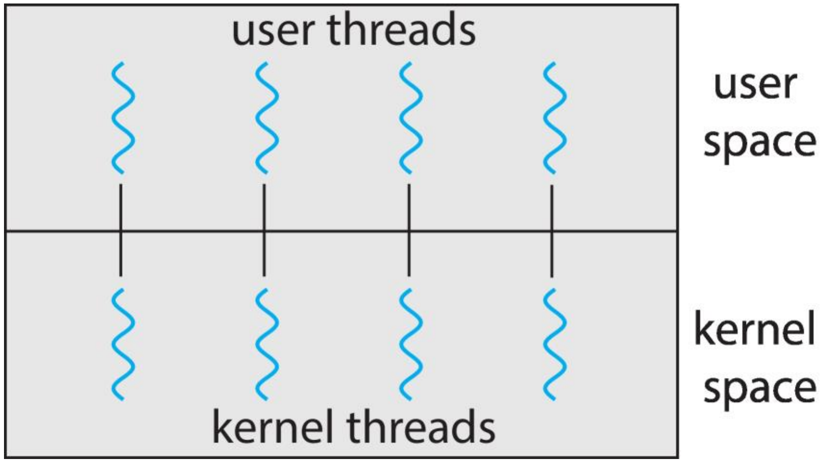
Many-to-One

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



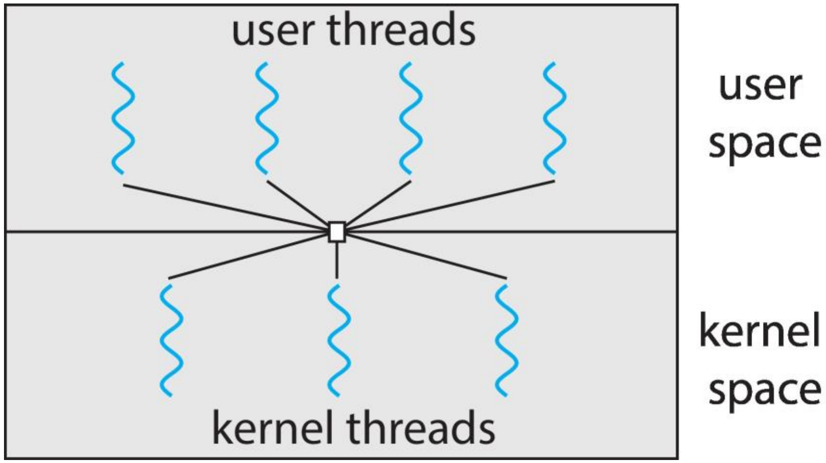
One-to-One

- 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
  - Linux



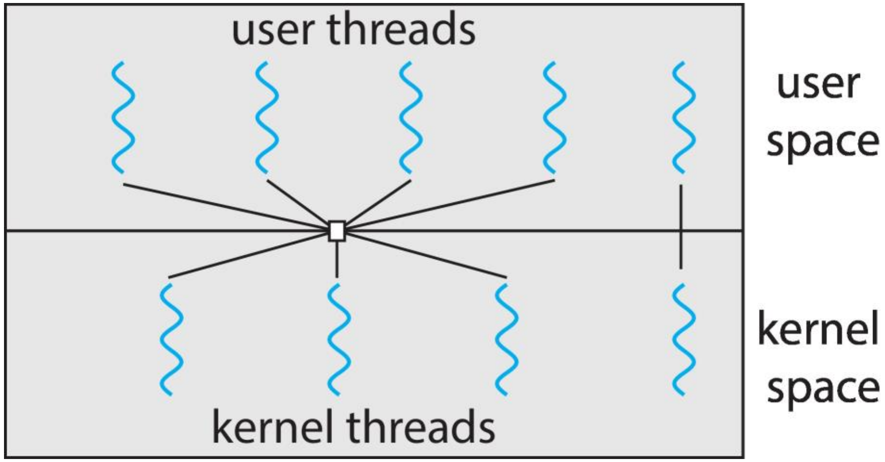
Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads M:N
- Allows the operating system to create a sufficient number of kernel threads
- Windows with the *ThreadFiber* package
- Otherwise not very common



Two-level Model

- Similar to M:N, except that it allows a user thread to be bound to kernel thread



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
- Scheduler Activations

Semantics of fork() and exec()

- Does **`fork()`** duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of **`fork()`**
- **`exec()`** usually works as normal – replace the running process including all threads

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
  1. Signal is generated by a particular event (e.g., process termination)
  2. Signal is delivered to a process
  3. Signal is handled by one of two signal handlers:
     1. default
     2. user-defined
- Every signal has default handler that kernel runs when handling signal
  - User-defined signal handler can override default
  - For single-threaded, signal delivered to process

Signal Handling (Cont.)

- Where should a signal be delivered for multi-threaded?
  - 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 Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is target thread
- Two general approaches:
  - Asynchronous cancellation terminates the target thread immediately
  - Deferred cancellation allows the target thread to periodically check if it should be cancelled
- Pthread code to create and cancel a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

 . . .

/* cancel the thread */
pthread_cancel(tid);

/* wait for the thread to terminate */
pthread_join(tid,NULL);
```

Thread Cancellation (Cont.)

- pThread code to create and cancel a thread:
  - pThreads: POSIX standard for thread programming
  - Need to #include <pthread.h>

Start routine

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

 . . .

/* cancel the thread */
pthread_cancel(tid);

/* wait for the thread to terminate */
pthread_join(tid,NULL);
```

Thread Cancellation (Cont.)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state and type

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

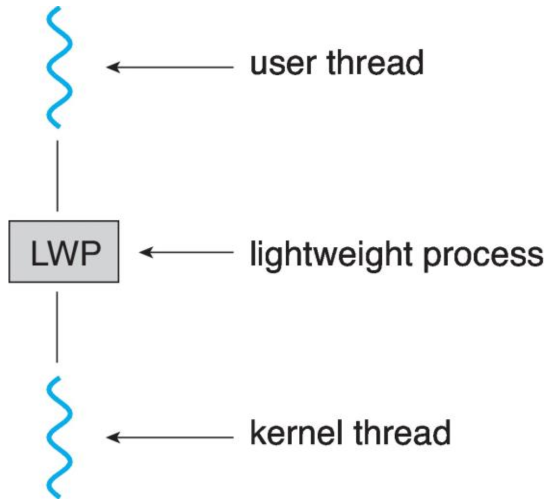
- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is *deferred*
  - Cancellation only occurs when thread reaches cancellation point
    - I.e., **`pthread_testcancel()`**
    - Then cleanup handler is invoked
- On Linux systems, thread cancellation is handled through signals

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 (i.e., when using a thread pool)
- 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

Scheduler Activations

- Both M:N and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads – lightweight process (LWP)
  - Appears to be a virtual processor on which process can schedule user thread to run
  - Each LWP attached to kernel thread
  - How many LWPs to create?
- Scheduler activations provide upcalls - a communication mechanism from the kernel to the upcall handler in the thread library
- This communication allows an application to maintain the correct number kernel threads



Operating System Examples

- Windows Threads
- Linux Threads

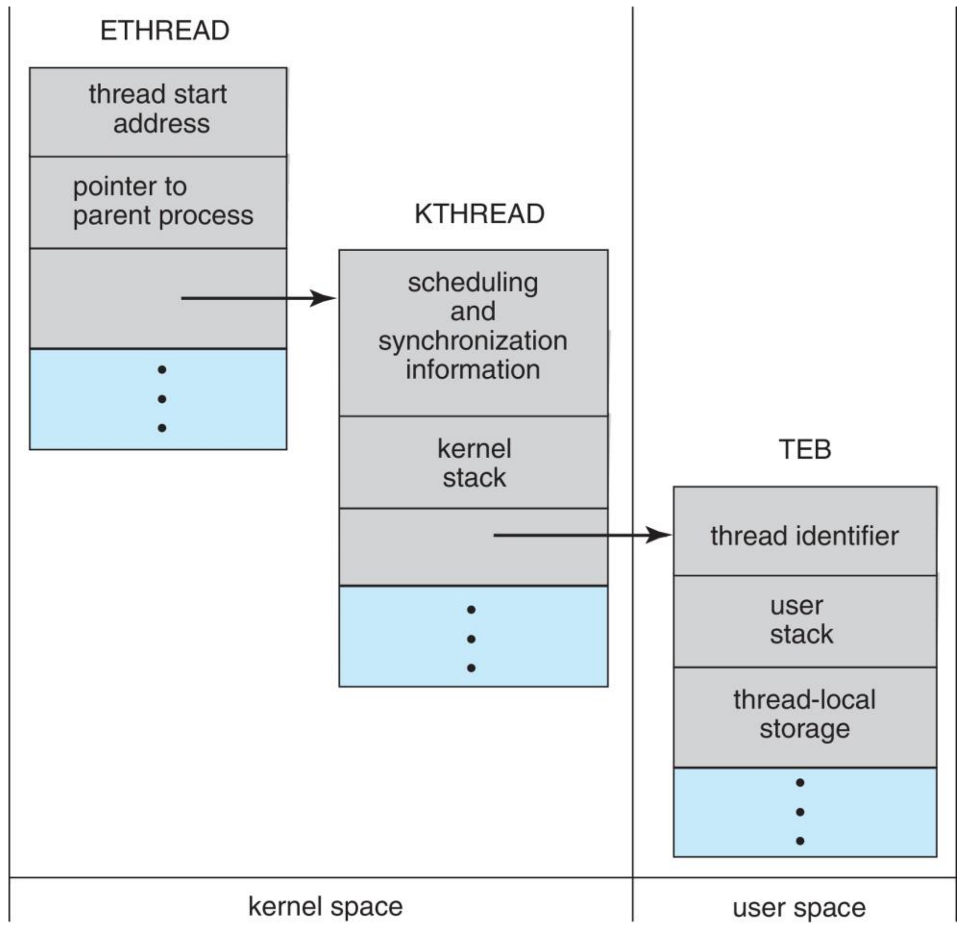
Windows Threads

- Windows API – primary API for Windows applications
- Implements the one-to-one mapping, kernel-level
- Each thread contains
  - A thread id
  - Register set representing state of 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

Windows Threads (Cont.)

- 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 Threads Data Structures



Linux 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 part of the execution context with the parent, such as address space, file descriptors, signal handler
- 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.

```
int clone(int (*fn)(void *), void *child_stack,
                int flags, void *arg, ... );
```

- The child thread executes the function **`fn`**, **`arg`** specified the arguments for the function **`fn`**.

End of Chapter 4