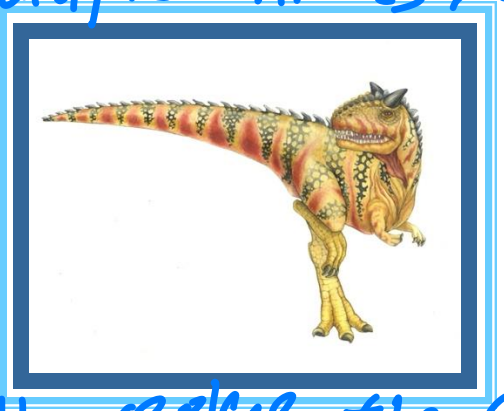
? servel threads → fork() will do?

worker! - 1 thread → UI

Chapter 4: Threads & Concurrency

- process, isolated
- have multiple threads, will share some resources

- fork()
  ↓
  exec()
  ↓
  run a new process, replace the address space



wait() → parent wait for child process finish

- blocking → x design explicitly buffer!

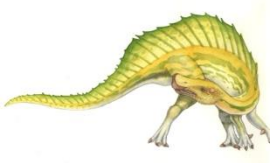
Chapter 4: Threads

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

how to run program much more fast

one thread → fork()

other thread → ?



Objectives

- Identify the basic components of a thread, and contrast thread and processes.
- Describe the major benefits and challenges in designing multithreaded processes.
- Describe how Windows and Linux operating systems represent threads

move fast

particular!

kernel → 4 threads (run parallel)

Motivation

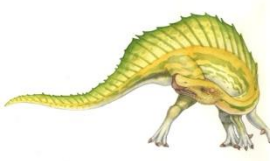
- Many applications or programs are multithreaded
  - A web browser might have one thread displaying images or text while another thread retrieving data from the network
  - A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from user, and a third thread for performing spelling and grammar checking in the background
- Most operating system kernels are multithreaded
  - For example, during system boot time on Linux, several kernel threads are created. Each thread performs a specific task, such as managing devices, memory management, or interrupt handling
- It can take advantage of processing capabilities on multicore systems
  - Parallel programming widely used in applications such as data mining, graphics, and artificial intelligence
- Process creation is time consuming and resource intensive
  - Process creation is heavy-weight while thread creation is light-weight, in which different threads belonging to the same process share code, data and others

one process ← DATA → 不同 heroes → 不同 threads (parallel execution)

run together!

involve copying the address space

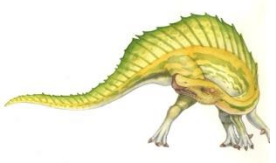
share the code, local variable, heap



Examples of Multithreaded Programs

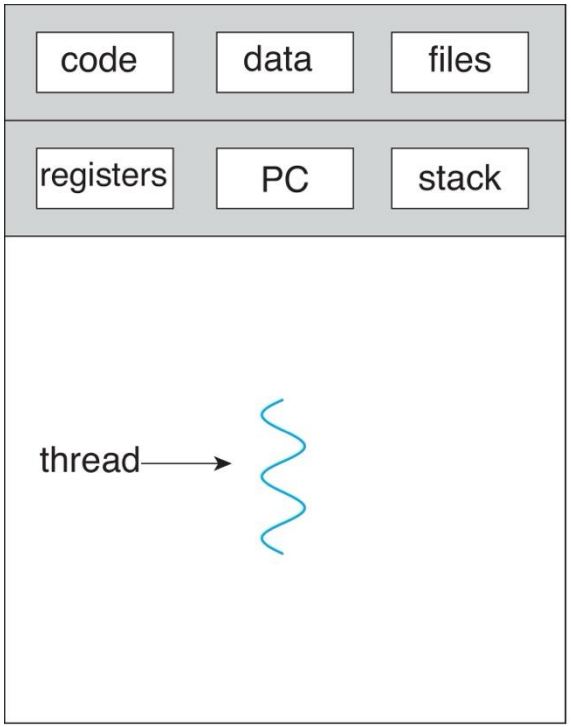
- Embedded systems 不重要!
  - Elevators, planes, medical systems, wristwatches
  - Single program, concurrent operations
- Most modern OS kernels
  - Internally concurrent to deal with concurrent requests by multiple users
  - But no protection needed within kernel
- Database Servers
  - Access to shared data by many concurrent users
  - Also background utility processing must be done
- Network Servers ← Middleware!
  - Concurrent requests from network
  - Again, single program, multiple concurrent operations ✓
  - File server, Web server, and airline reservation systems
- Parallel Programming / Run faster!
  - Split a program and data into multiple threads for parallelism

Every program is multi-threaded nowadays

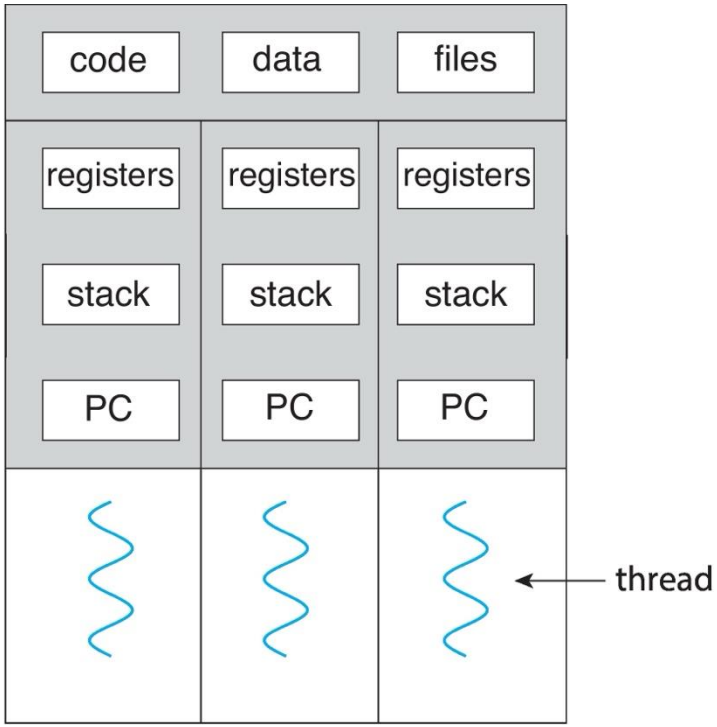


Single and Multithreaded Processes

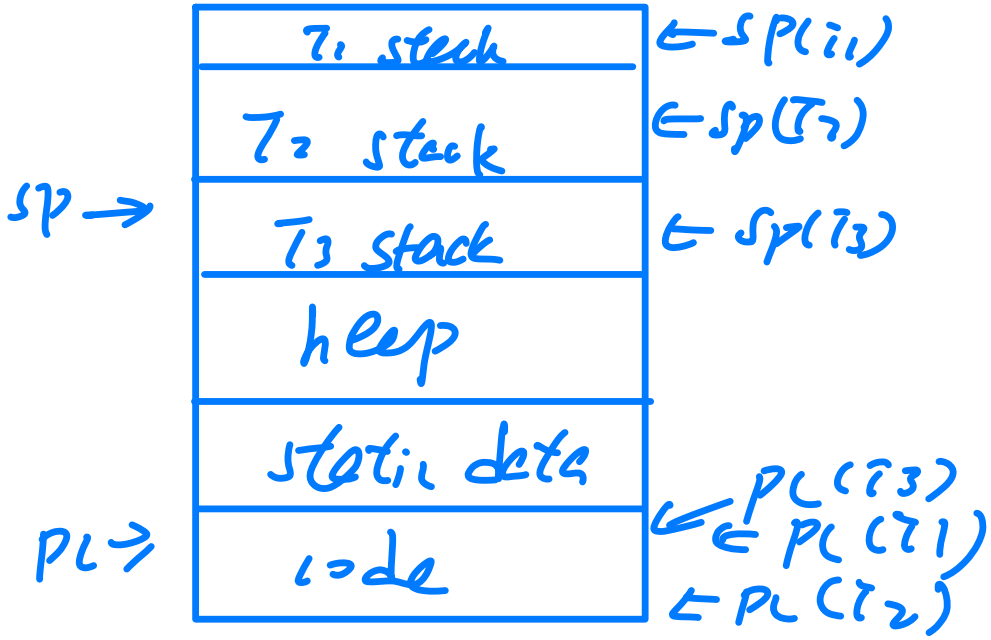
- Recall that a thread is a basic unit of CPU utilization – independently scheduled and run (referred as an instance of execution) represented by a thread ID, a program counter (PC), a register set, and a stack. It shares with other threads of the same process its code, data, and other OS resources, such as open files and signals



single-threaded process



multithreaded process



- Different PC in different threads
- Share the same code (address space)
- have own thread!
- 不同 Sp

Benefits

one process → more responsible

① Responsiveness – may allow continued execution if part of process (e.g., one thread of a process) is blocked, this could be especially important for interactive applications having user interfaces

② Resource Sharing – threads share resources of a process by default, easier than shared memory or message passing between processes – as they essentially run within the same address space of the process

2/3 threads → share

③ Economy – thread creation is much "cheaper" than process creation (consuming less time and memory) , context switching is typically faster between threads of a process than between processes

fork() → copy everything → very time consuming

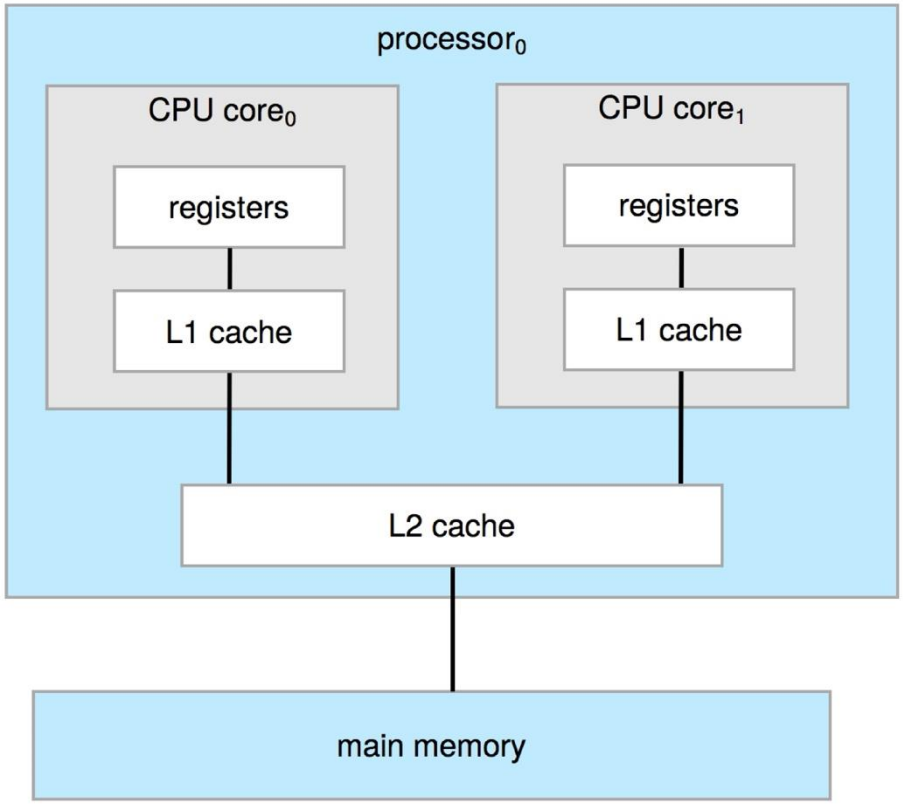
④ Scalability – processes can take advantage of multicore architecture ✓

Multiple core → run together → fits

A Multi-Core Design

- The multithreaded programming provides a mechanism for more efficient use of multicores and improves concurrency in **multicore** systems.

distribute 2 threads! Perfect fits!



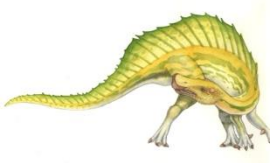
Multicore Programming

follow principles:

- In multicore or multiprocessor systems, there are significant and new challenges in programming design include:
  - Dividing tasks – how to divide into separate, concurrent tasks
  - Balance – each task perform "equal" amount of work
  - Data splitting - data used by tasks must be divided to run on separate cores
  - Data dependency – if there is a data dependency, synchronization is required
  - Testing and debugging – different path of executions makes debugging difficult

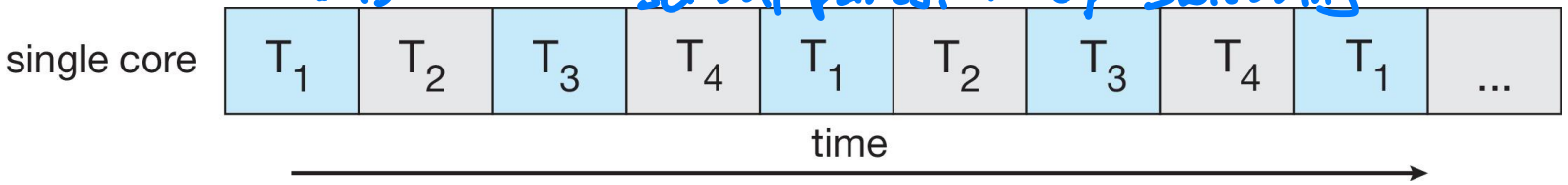
How to speed up:

- There is a clear distinction between parallelism and concurrency
- Parallelism - a system can perform more than one task simultaneously → run multiple part of yr program simultaneously → same time
- Concurrency supports more than one task for making progress Refer to the state → proceed all progress
  - Multiplexed over time, single processor core, scheduler provides concurrency
- The advent of multicore systems require an entirely new approach in designing software systems with particular emphasis on parallel programming.

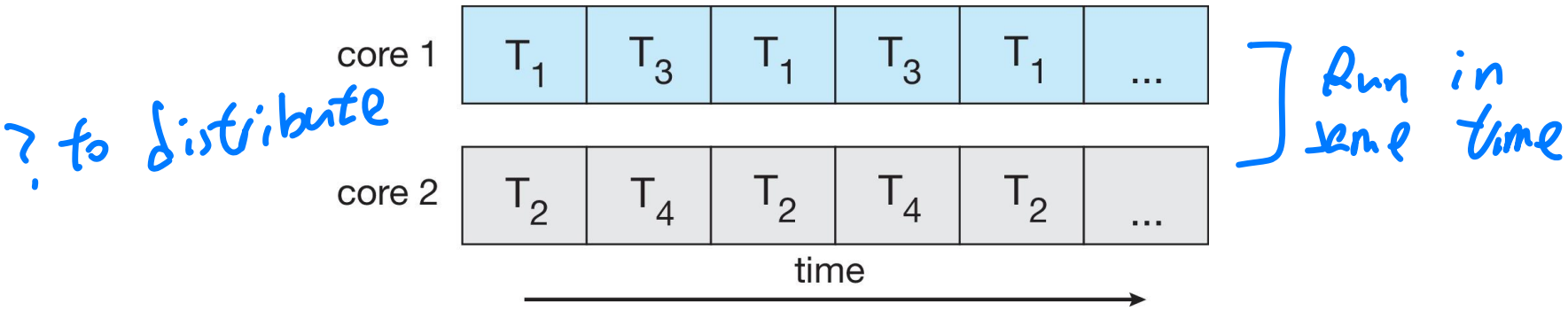


Concurrency vs. Parallelism

- Concurrent execution on single-core system – multiplexing over time

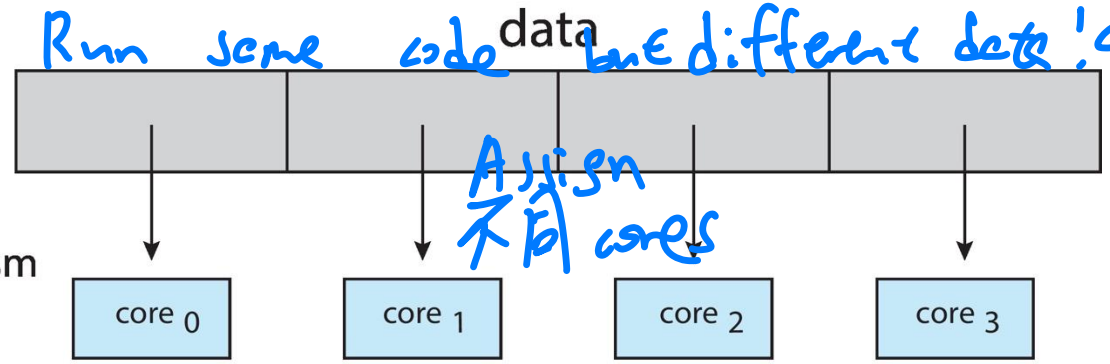


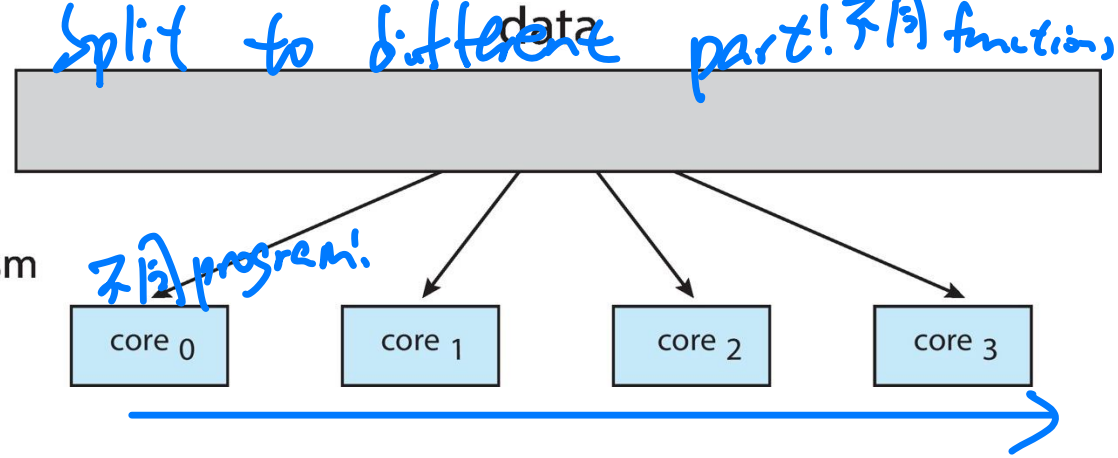
- Parallel execution on a multi-core system:

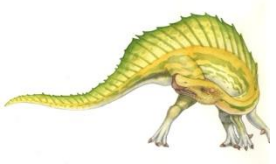


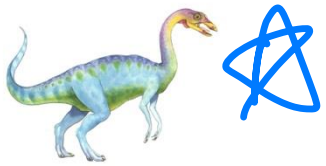
Data and Task Parallelism

- **Data parallelism** – distributes subsets of the data across multiple cores, same operation on each core (common in distributed machine learning tasks)
- **Task parallelism** – distributing threads across cores, each thread performing unique operation
- Data and task parallelism are not mutually exclusive, an application may use both - hybrid?









Amdahl's Law

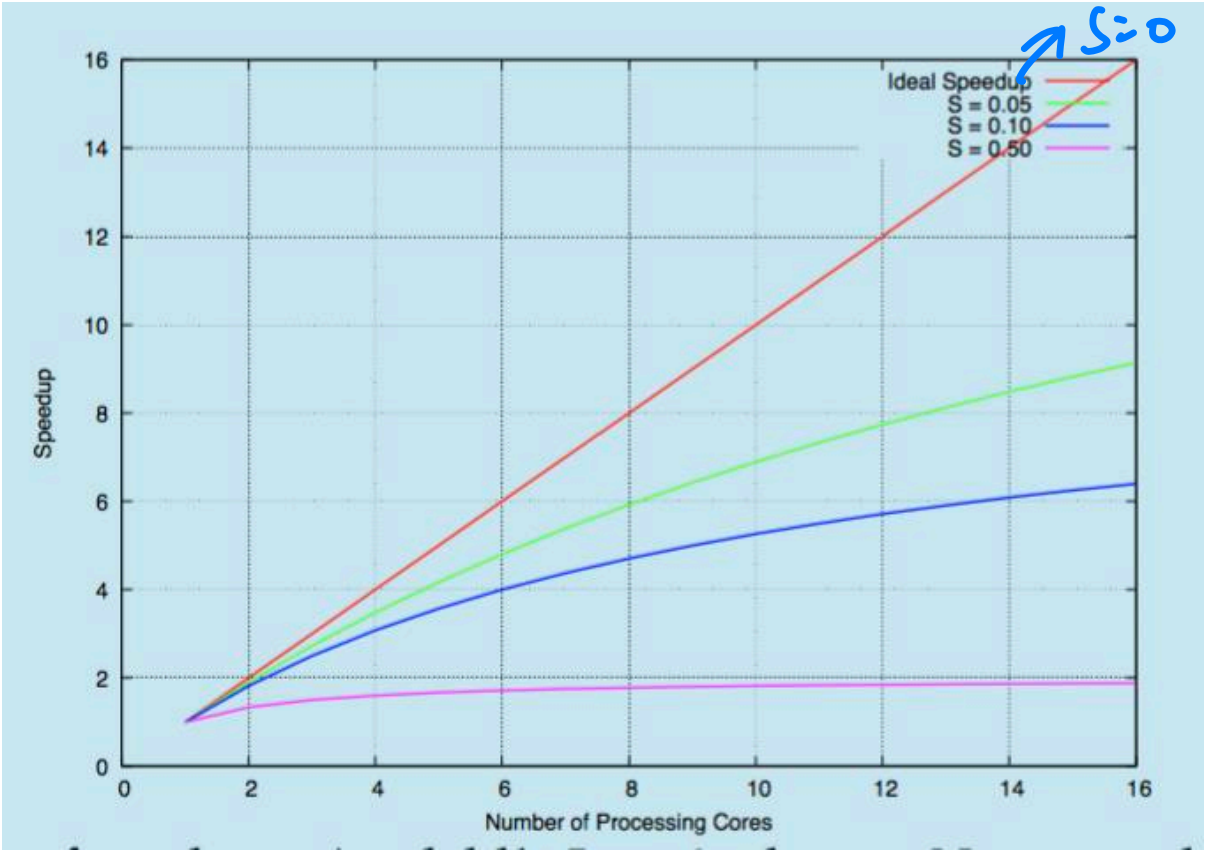
- It identifies performance gains (theoretical speedup in latency of the execution of a task at fixed workload) from adding additional cores to an application that has both serial and parallel components in programs
- *S* is serial portion and 1-S is parallel portion
- *N* processing cores

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

- That is, if application is 75% parallel and 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 a disproportionate effect on performance gained by adding additional cores

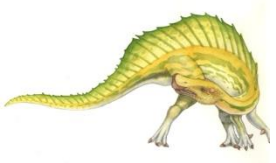
Amdahl's Law



large code → x parallel → very slow :(

? design the code!

S↓↓ as small as possible!

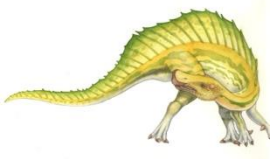


Multithreaded Process

- A multi-threaded process has more than one instance of execution - each of which with a program counter is being fetched and executed
- A thread is similar to a process, except they share the same address space and thus can access the same set of data with other thread(s) in a process
- The state of a single thread is also similar to that of a process - It has a program counter (PC) that tracks where the program is fetching instructions from. Each thread has its own private set of registers for execution
- If two threads run on a single processor, when switching from running one (T1) to running the other (T2), a context switch must also take place
  - The register state of T1 must be saved and the register state of T2 restored from T2's stack before running T2
  - The address space remains the same, the context switch overhead is much smaller
  - When switching to a thread belonging to a different process, overhead is more
- This provides parallelism in a process (multithreaded) execution and can enable overlap of I/O with other activities within a single program
  - One thread is running on CPU, and another thread of the process is doing I/O

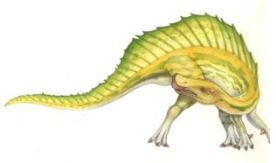
Thread

- Thread: The 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, which process it belongs to

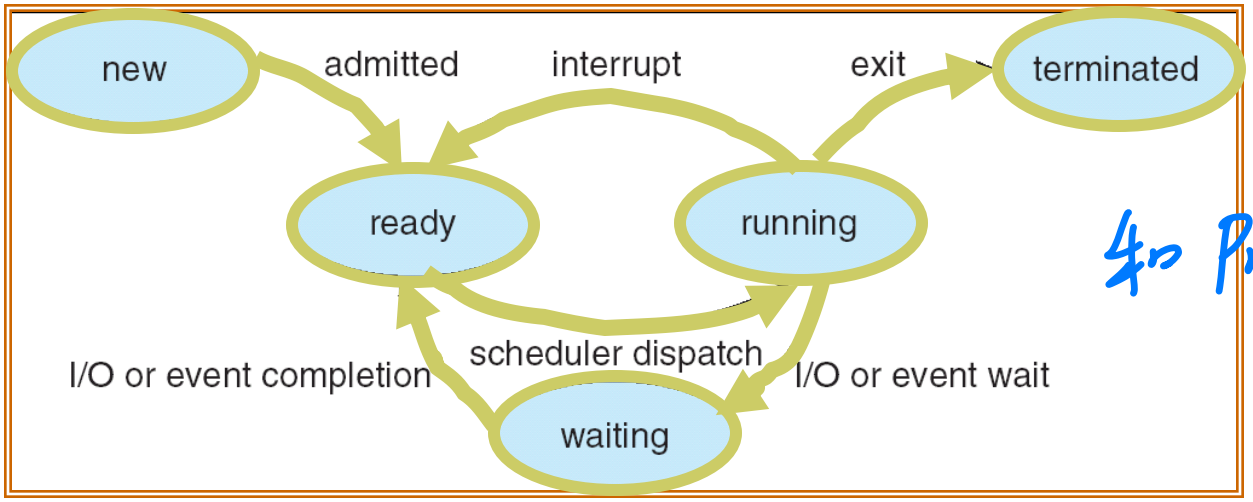


Thread State

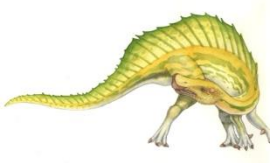
- Threads in a way encapsulate concurrency, which is the "active" component of a process
- Address space encapsulates protection: which can be considered as the "passive" part of a process
  - One program address space is different from that of another program so to keep buggy program from thrashing the entire 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)



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



Some threads is waiting
→ How to know whether process is running or not?

Process is running
- ≥ 1 thread is running → process is running

Process in ready
- ≥ 1 thread in ready → process is ready
- X threads are running

Process in waiting state
- All threads in process is waiting state

Process in terminated state
All of its threads have finished and are in terminated state

context switch of process produces much more overhead!

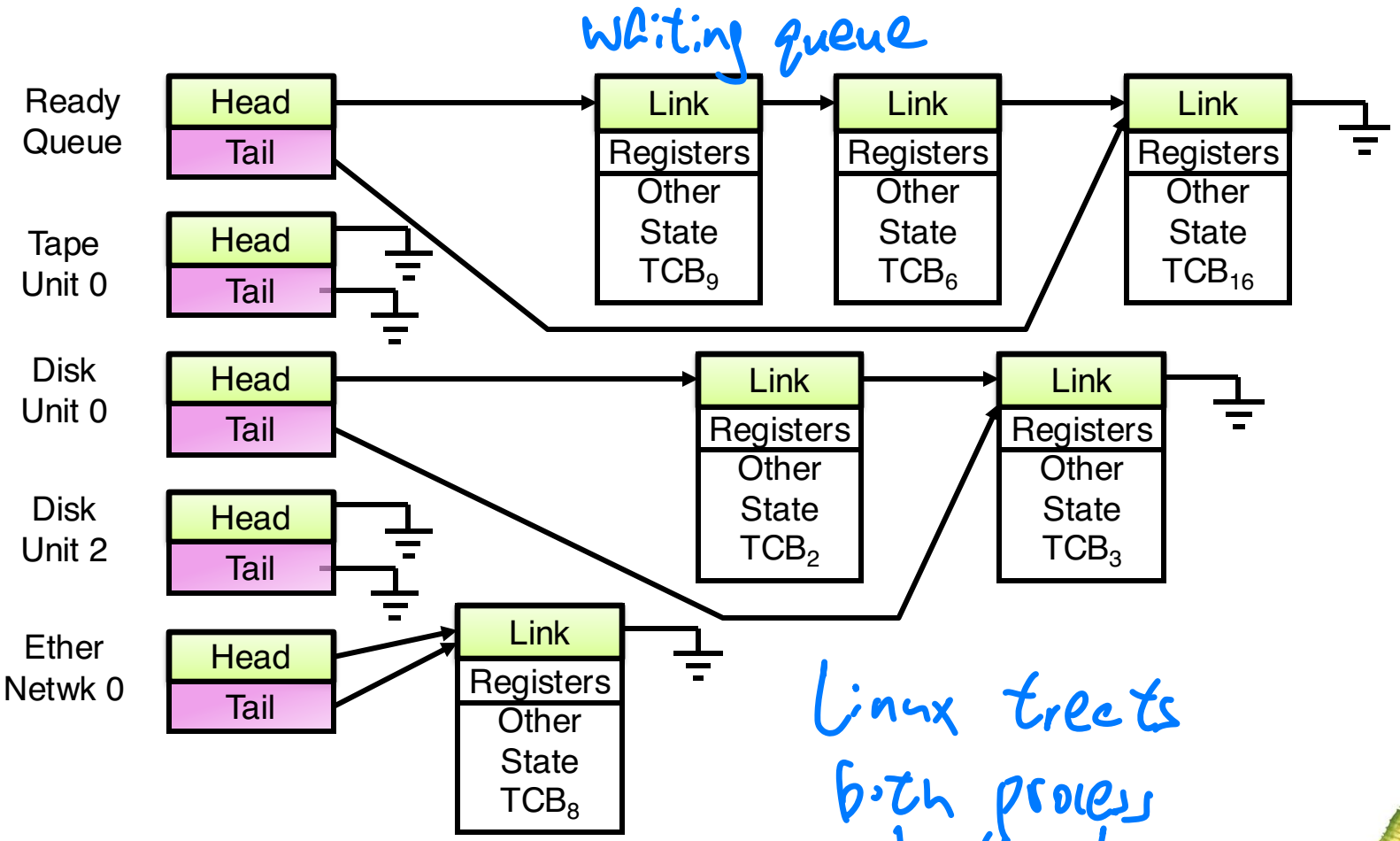
Single thread, only one-to-one mapping CPU do Scheduling

Similar to PCB!

Ready Queue And Various I/O Device Queues

Thread not running ⇒ TCB is in some other queue

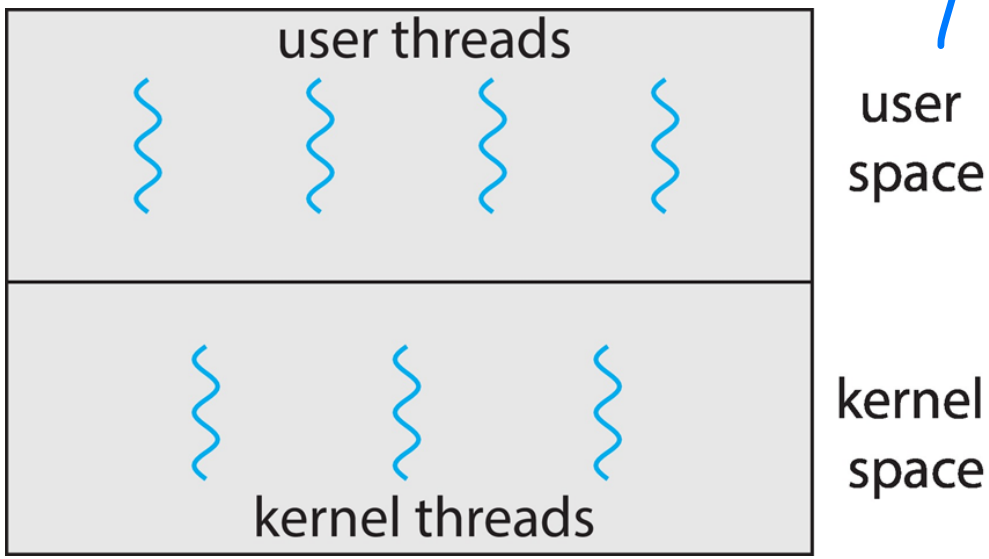
- Queue for each device/signal/condition, each with its own scheduling policy



Linux treats both process and threads as same thing, called tasks

User Threads and Kernel Threads

- User threads – independently executable entity within a program, created and managed by user-level threads library
- Kernel threads – can be scheduled and execute on a CPU, supported and managed by the operating system
- Examples – virtually all general-purpose operating systems, including:
  - Windows, Linux, Mac OS X iOS, Android



Multithreading Models

- User-level threads are visible only to programmers and unknown to the kernel. Consequently, they cannot be scheduled to run on a CPU. In another word, OS only manages and schedules kernel threads
- Threads libraries provide APIs for creating and managing user threads in programs. There are three primary thread libraries - POSIX Pthreads, Windows threads and Java threads, corresponding to the three most common APIs, e.g., Win32 API, POSIX API and Java API
- In a way, user-level threads provide the concurrent and modular entities in a program that the OS can potentially take advantage of. Or if there is no user threads defined or specified in a program, there will be no kernel thread(s) that can be scheduled for the program execution

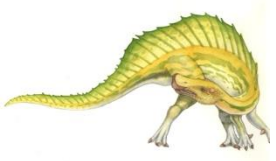
write in program

logically, cannot directly execute in CPU

```
pthread_t tid;

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

→ in labs!

OS only maps users threads!



Multithreading Models (Cont.)

- Ultimately, a mapping must exist between user threads and kernel threads
  There are three common ways of establishing such a relationship:
  - Many-to-One: many user-level threads mapped to one kernel thread
  - One-to-One: each user-level thread mapped to a kernel thread – most common in modern operating system
  - Many-to-Many: many user-level threads mapped to many kernel (usually smaller number of) threads

Hw / Exam!.

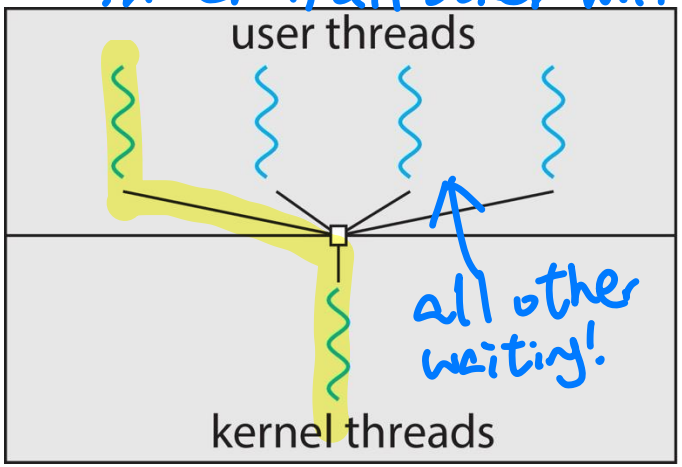
many core, choose one Block all others except only one!.

Many-to-One

- Many user-level threads mapped to single kernel thread
- Which user thread is currently mapped to the kernel thread is a scheduling problem, known as process-contention scope or PCS (to be discussed in Chapter 5)
- One thread blocking (i.e., the kernel thread) causes all threads mapped to this thread to block, i.e., the entire process is blocked
- Multiple user threads can not run in parallel on multicore system because only one kernel thread can be active at a time
- Few systems currently use this model
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads

=) X speed up!

only one thread can scheduled in CPU, all other will be blocked!



all other waiting!

No meaningful!

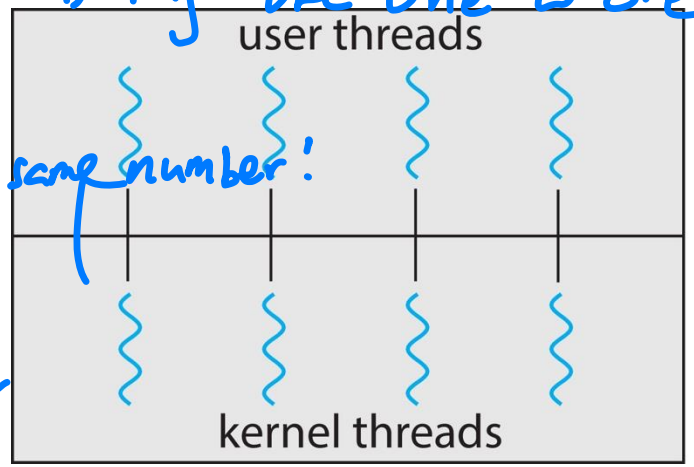
Fake multi-threading!

Separate to other core.

All other in waiting state!

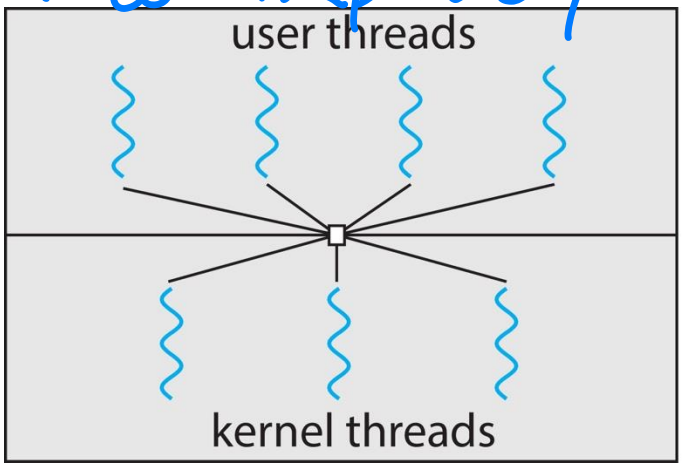
One-to-One

- Each user-level thread maps to one kernel thread
- It provides the maximum concurrency, and it also allows multiple threads to run in parallel on multiprocessor or multicore systems
- This implies that creating a user-level thread mandates creating a corresponding kernel thread – the number of threads per process sometimes can be restricted due to overhead (kernel threads consume resources such as memory, I/O, etc.)
- Examples
  - Windows
  - Linux



Many-to-Many Model

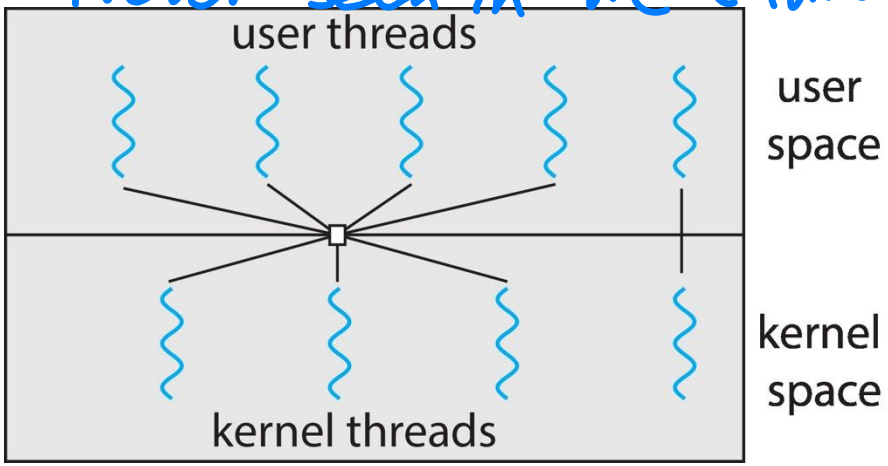
- Multiplexe many user-level threads to a smaller or equal number of kernel threads, specific to a particular application or machine
- This provides certain level of concurrency for a process execution – also process-contention scope or PCS is involved for scheduling
- This allows the operating system to create a sufficient number of kernel threads in advance – thread pool
- Windows with the ThreadFiber package
- Otherwise not very common

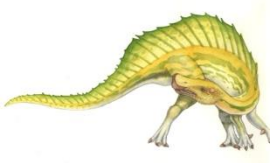


Two-level Model

- Similar to the Many-to-Many model, except that it allows a user thread to be bound to a kernel thread

combine one to one to many to many!
Never seen in the environment



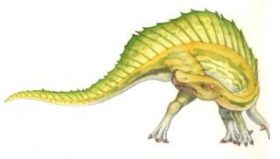


Multithreading Mapping Models

- The many-to-many model is most flexible, but can be difficult to implement in practice
  - For instance, an application may be allocated more kernel threads on a system with eight processing cores than on a system with four cores
- With an increasing number of processing cores in modern computer systems, limiting the number of kernel threads has become less important.
- Most operating systems now use the one-to-one model

Few OS uses this!

↓
resource waste
↓
can be ignored



Threading Issues

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

Semantics of fork() and exec()

- Does **`fork()`** called by a thread duplicate only the calling thread or all threads of a process?
- Some UNIX systems have chosen to have two versions of **`fork()`** - If **`exec()`** is called immediately after forking, then duplicating all threads is unnecessary, as the program specified in **`exec()`** will replace the entire process. In this instance, duplicating only the calling thread is appropriate.
- **`exec()`** usually works as normal – replace the entire process including all threads

Duplicate all the threads

do not copy everything!

Only copy the calling thread!

Save some overhead

boost the performance

Signal Handling

- **Signals** are used 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 and the reason
  - Synchronous signals include illegal memory access and division by 0 - delivered to the same process that performed the operation that caused the signal (that is why they are considered synchronous)
  - When a signal is generated by an event external to a running process – asynchronous signals. Examples include terminating a process with specific keystrokes (such as **<control><C>**) and a timer expires
- Signals are handled in different ways
  - Some signals may be ignored, while others (for example, an illegal memory access) are handled by terminating the program

Signal Handling (Cont.)

multithread may be different

- A signal handler is used to process signals, following the pattern below
  - Signal is generated by a particular event (e.g., process termination)
  - Signal is delivered to a process
  - Signal is handled by one of two signal handlers, default or user-defined
- Every signal has default handler that kernel uses to handle that signal
  - User-defined signal handler can override default handler
- For a single-threaded process, signals are delivered to that process or thread
- Where should signal be delivered for a multi-threaded process?
  - 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

Question?

Signal Handling (Cont.)

- The method for delivering a signal depends on the type of signal generated
  - For example, synchronous signals need to be delivered to the thread causing the signal and not to other threads in the process
  - Some asynchronous signals — such as a signal that terminates a process (**<control><C>**, for example)—should be sent to all threads
- UNIX function for delivering signal **kill(pid t pid, int signal)**, which specifies the process (**pid**) to which a particular signal (**signal**) is to be delivered
- POSIX Pthreads function allows a signal to be delivered to a specified thread (**tid**): **pthread_kill(pthread t tid, int signal)**

Thread Cancellation

how to terminate the thread!

- Terminating a thread before it has finished, and thread to be canceled is referred to as the target thread.
- The cancellation occurs in two different scenarios
  - false cancellation
  - Asynchronous cancellation terminates the target thread **immediately** right now
  - Deferred cancellation The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion
- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state and type

  Tallocate some memory
  tell thread → terminate
  delete the unnecessary resources!
- In Pthreads, if thread has cancellation disabled, cancellation remains pending until thread enables it – the default type is deferred

write some in flag!

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

→ different Strategy

Thread Cancellation (Cont.)

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

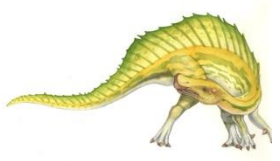
In labs/Tutorial1

```
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);
```

Start routine

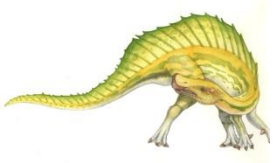
not immediately

make cancelation



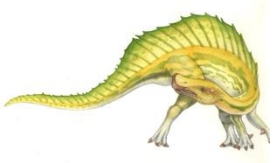
Operating System Examples

- Windows Threads
- Linux Threads



Windows Threads

- A Windows application runs as a separate process, and each process may contain one or more threads
- Windows API – primary API for Windows applications
- Windows uses the one-to-one mapping, where each user-level thread maps to an associated kernel thread
- The general components of a Window thread include:
  - A thread ID identifying the thread
  - Register set representing the state of the processor
  - A program counter (PC)
  - Separate user and kernel stacks 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

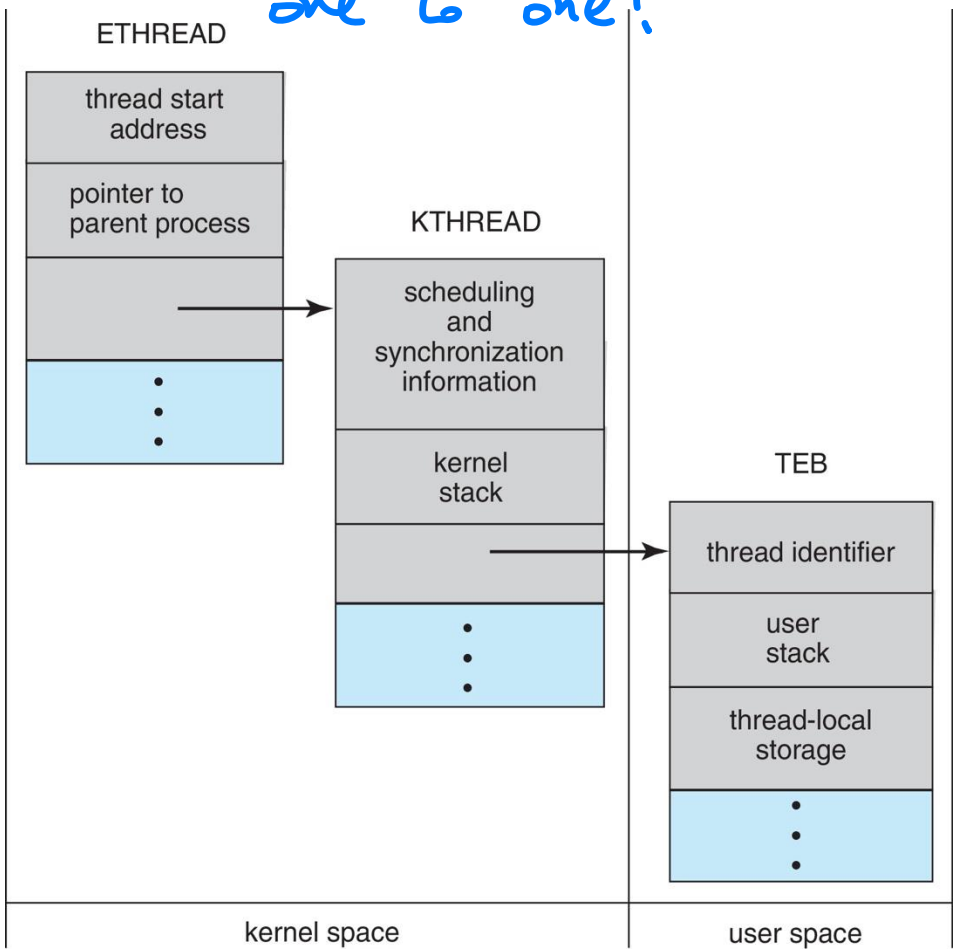


Windows Threads (Cont.)

Memory windows, threads/process are different

The primary data structures of a thread include:

- ETHREAD (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD,
- 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



Linux Threads

- Linux does not differentiate between threads and processes. To Linux, a thread is just a special kind of process, all referred to as a **task**
- A thread in Linux is a process that may share certain resources with other threads. Each thread has a unique **task_struct** (illustrated in Chapter 3) and appears as a normal process - threads in Linux just happen to share resources, such as an address space, with other processes
- This approach to threads contrasts greatly with operating systems such as Microsoft Windows or Sun Solaris (Unix), which have explicit kernel support for threads (and sometimes call threads *lightweight processes*)
- For example, if a process consists of two threads
  - On Microsoft Windows, one PCB exists describing the shared resources such as address space or open files, and that, in turn, points to the two different threads. Each thread TCB describes the resources it alone possesses
  - In Linux, there are simply two tasks with normal **task_struct** structures. The two processes are set up to share resources

Linux Threads

- Threads are created in the same way as normal tasks, with the exception that the **`clone()`** system call is used to pass flags corresponding to the specific resources to be shared:

```
clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);
```

  - **`clone()`** allows a child task to share the part of the execution context with a parent task, such as address space, file descriptors, signal handler
  - Flags control behavior

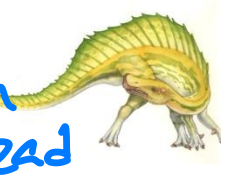
flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

fork()
↓
clone system do not share everything

- Instead of copying all data structures like **`fork()`**, new task points to data structures of parent task, depending on set of flags passed to **`clone()`**
- A normal **`fork()`** can be implemented as **`clone(SIGCHLD,0)`**;

control two execution instance!

create between process and thread



End of Chapter 4