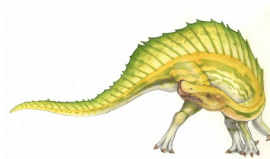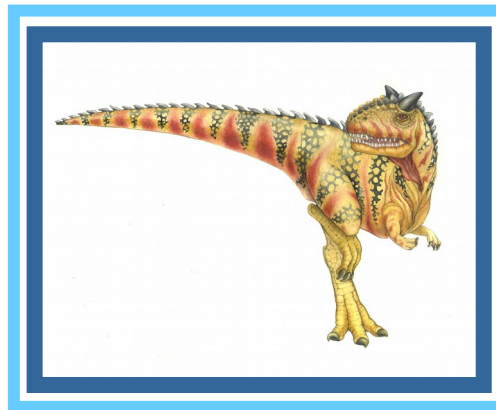# Chapter 4: Threads & Concurrency

Chapter 4: Threads

- Overview
- Multi-core Programming
- Multi-threading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples

Objectives

- Identify the basic components of a thread, and contrast threads and processes

- Describe the benefits and challenges of designing multi-threaded applications

- Illustrate different approaches to implicit threading, including thread pools and fork-join

- Describe how the Linux operating system represents threads

- Explore multi-threaded applications using the Pthreads, Java, and Windows threading APIs

Motivation

- Kernels are generally multi-threaded

- Most modern applications are multi-threaded

- Whereas cooperating processes are independent, cooperating threads run within the same process (think application)

- Multiple functions or tasks within an application can be implemented by separate threads. Example decomposition:

  - A thread to update display

  - A thread to fetch data from a database

  - A thread to run a tool such as a spell-checker

  - A thread to respond to a network request

- Process creation is costly and slow, whereas thread creation is light-weight

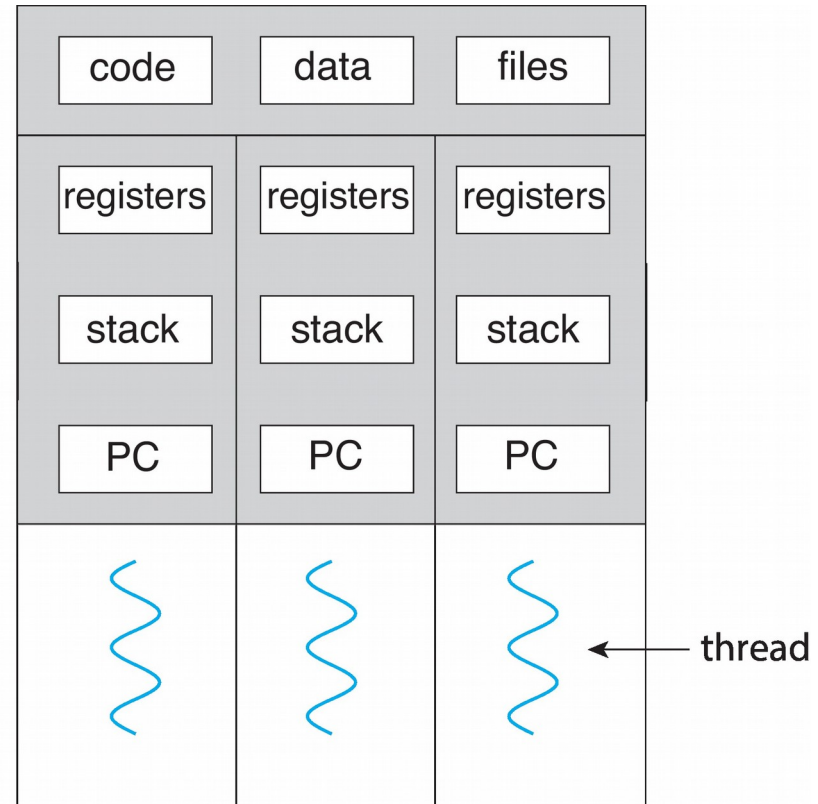- Proper threading can simplify code, increase efficiency
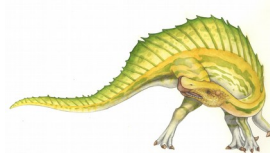
Single versus Multi-threaded Processes

code	data	files
registers	PC	stack
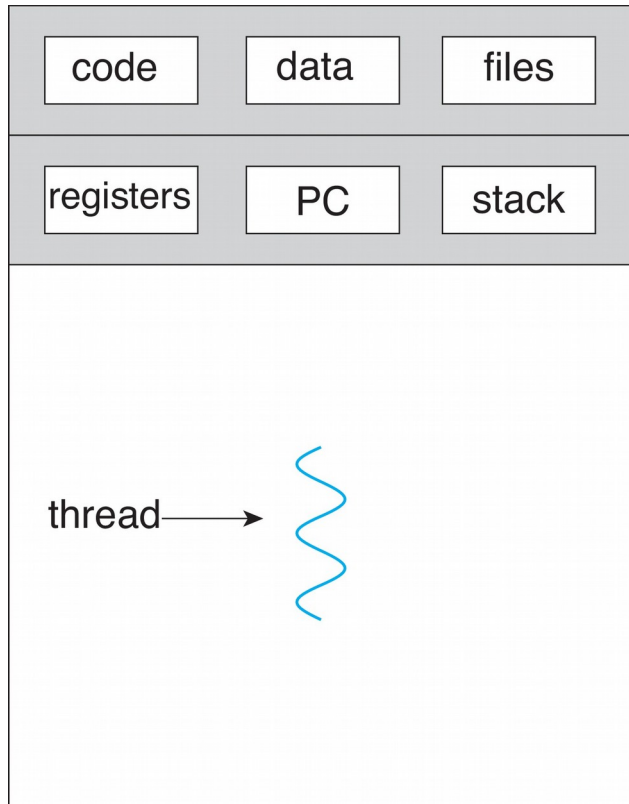
thread →

single-threaded process

| code	data	files
registers	registers	registers
stack	stack	stack
PC	PC	PC

← thread

multithreaded process

Each thread has its own register set, stack, and PC.

Multi-threaded Server Architecture

1) client sends request to server;
2) server creates a thread to process the request, and
3) immediately returns to listening for the next request
   from a client in the same main thread.
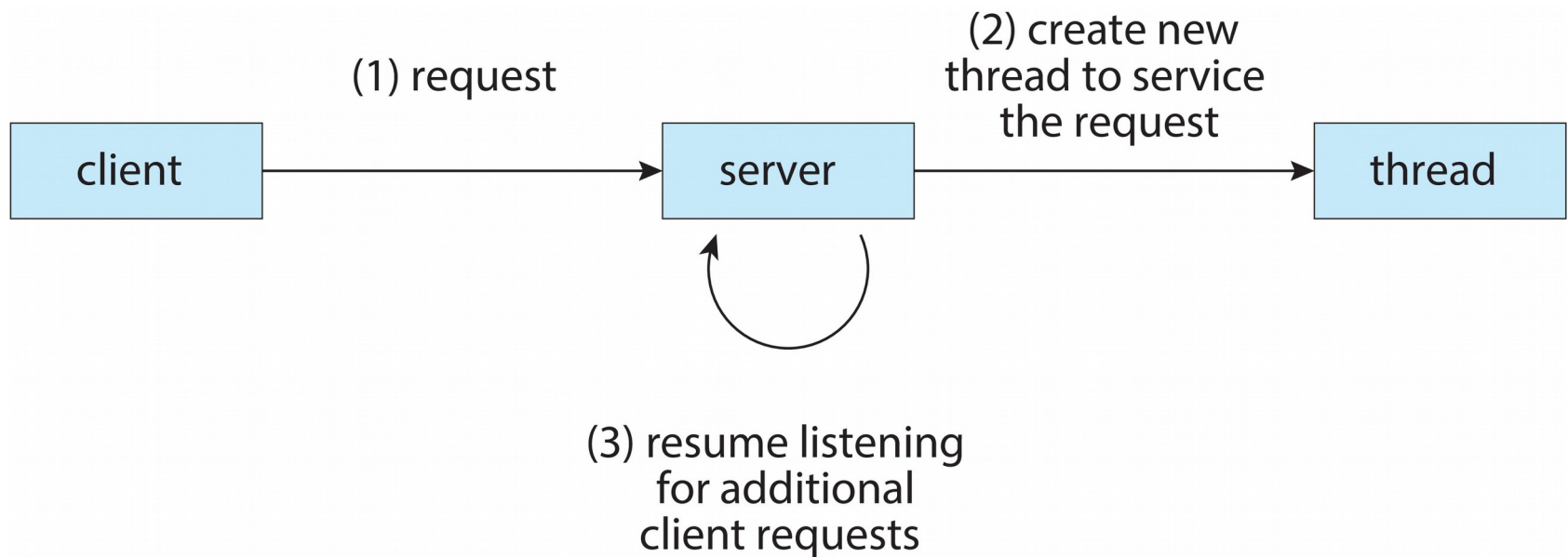
Benefits of Threads

- **Responsiveness –** may allow continued execution if part of process is blocked, or some slow operation in a different thread - especially important for user interfaces

- **Resource Sharing –** threads share same address space in single process, easier than processes using shared memory or message passing

- **Cost –** cheaper than process creation, thread switching lower overhead than context switching

- **Scalability –** process can take advantage of multi-core architectures

Concurrency vs. Parallelism

- Two or more sequences of instructions are said to be **concurrent** if no matter what order they are executed in relation to each other, the final result of their combined computation is the same.

- This means that they can be executed simultaneously on different processors, or interleaved on a single processor in any order, and whatever outputs they produce will be the same.

- A system with two ore more concurrent processes is called a **concurrent program** or a **concurrent system**.

- Two processes or threads execute **in parallel** if they execute at the same time on different processors.

- **Parallel programs** are those containing instruction sequences that can be executed in parallel. A parallel program is always a concurrent program, but a system can have concurrency even though it is not a parallel program.
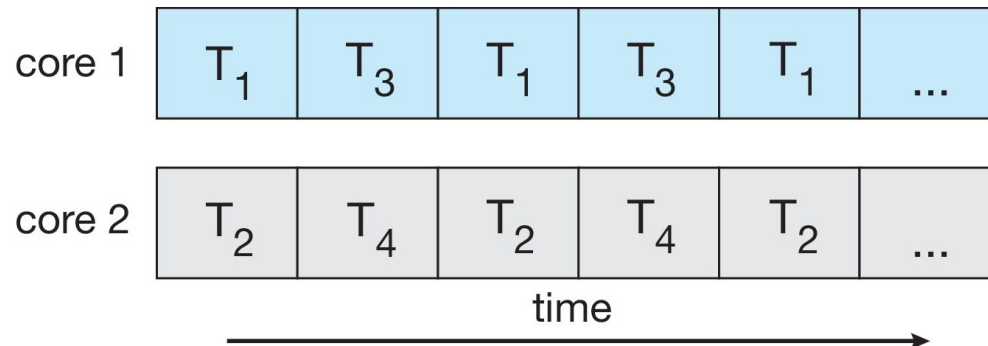
Concurrency vs. Parallelism

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

single core	T_1	T_2	T_3	T_4	T_1	T_2	T_3	T_4	T_1	...

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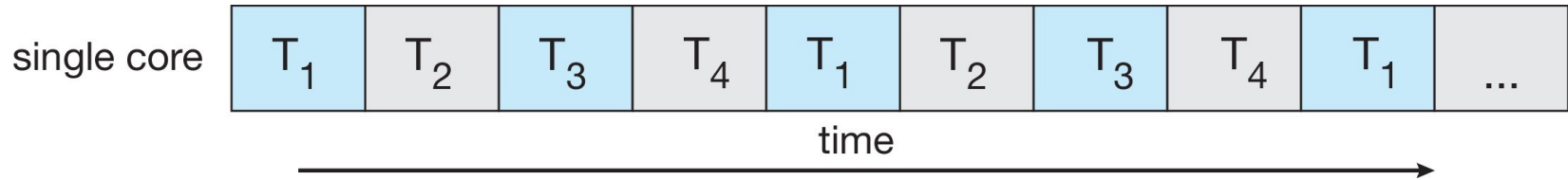
- **Parallelism on a multi-core system:**

core 1	T_1	T_3	T_1	T_3	T_1	...

core 2	T_2	T_4	T_2	T_4	T_2	...

time →

Multi-core Programming

- **Multi-core** or **multi-processor** systems challenge programmers to take advantage of hardware, but it is not easy:

  - **How to decompose a single task into many independent parallel tasks**

  - **How to load-balance the tasks**

  - **How to split data onto separate cores/processors**

  - **How to identify data dependency and handle synchronization**

 - **How to test and debug parallel programs**

Multi-core Programming (cont.)

- Types of inherent parallelism:

  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each

    - an image on which the same operation is applied to all pixels

    - a payroll with taxes to be calculated for all individuals

    - a set of points to be rotated through same angle in space

  - **Task parallelism** – distributing threads across cores, each thread performing unique operation

    - same data set evaluated by multiple algorithms for some property (census data analyzed for demographics, financials, geographic, etc)

Data and Task Parallelism
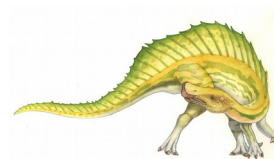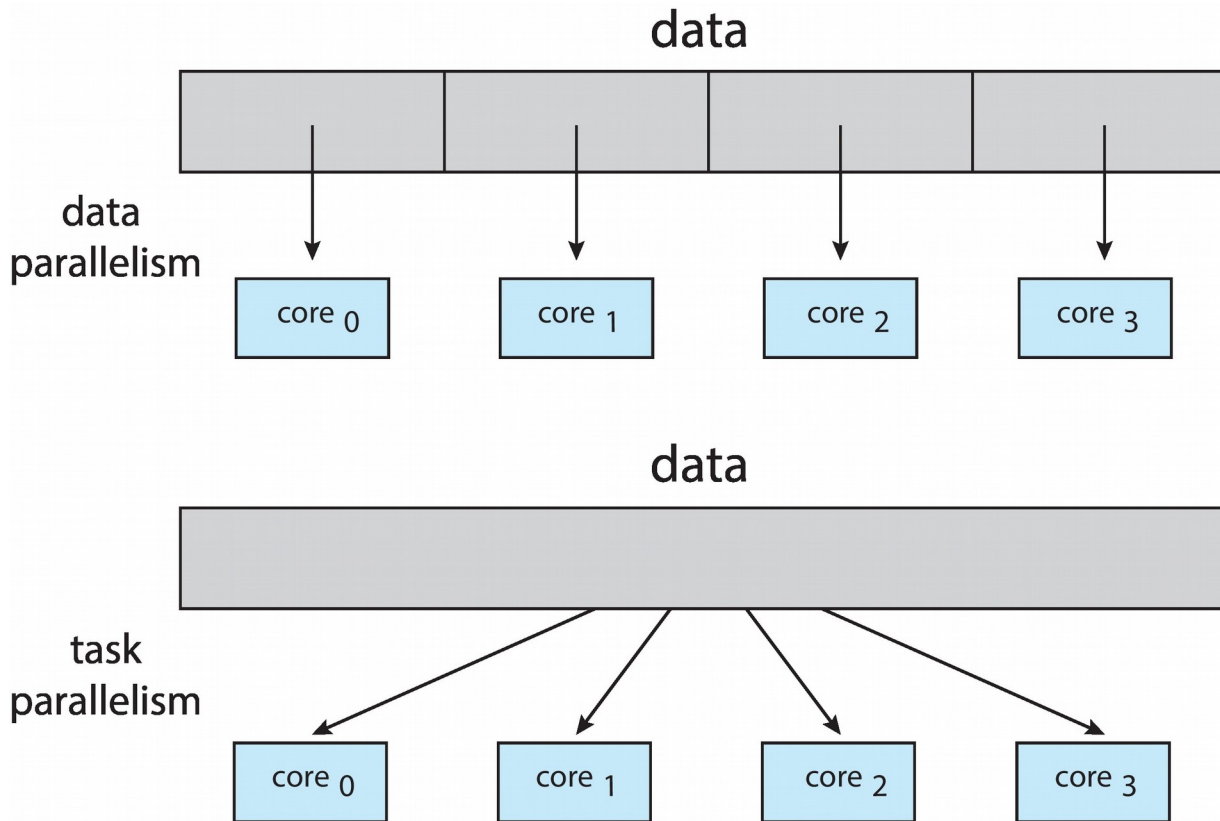
Amdahl's Law

- In 1967, Gene Amdahl argued that there was an inherent limitation to the amount of speedup that could be obtained by performing a computation using more processors. His argument is known as "Amdahl's Law". If

  - *S, 0* <= *S* <= *1,* is the fraction of operations that must be executed serially (in sequence), and

  - *N* is the number of processing cores, then the speed-up is bounded above:
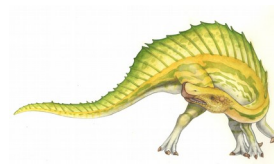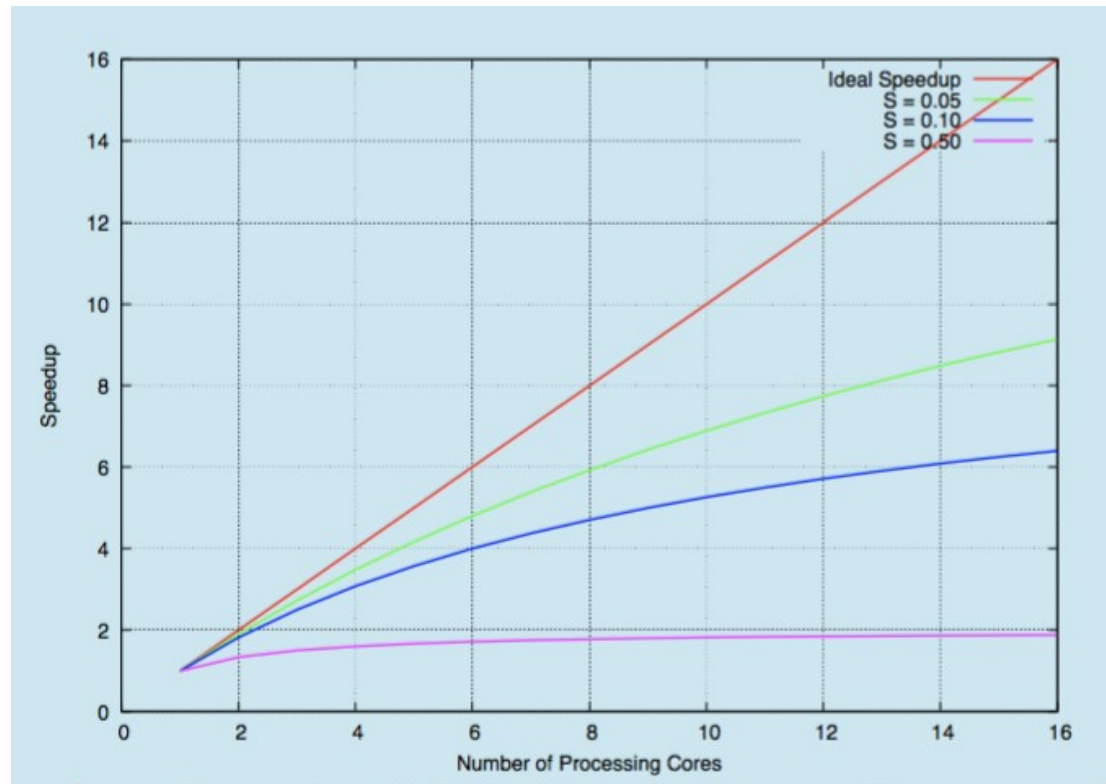
$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- Example: if program is 75% parallel / 25% serial, (S=0.25) moving from 1 to 2 cores (N=2) results in speedup of 1/((1/4) + (3/4)/2)) = 1.6

- As *N* approaches infinity, speedup approaches 1 / *S*

Serial portion of an application limits maximum performance gained by adding additional cores

Amdahl's Law Graphically
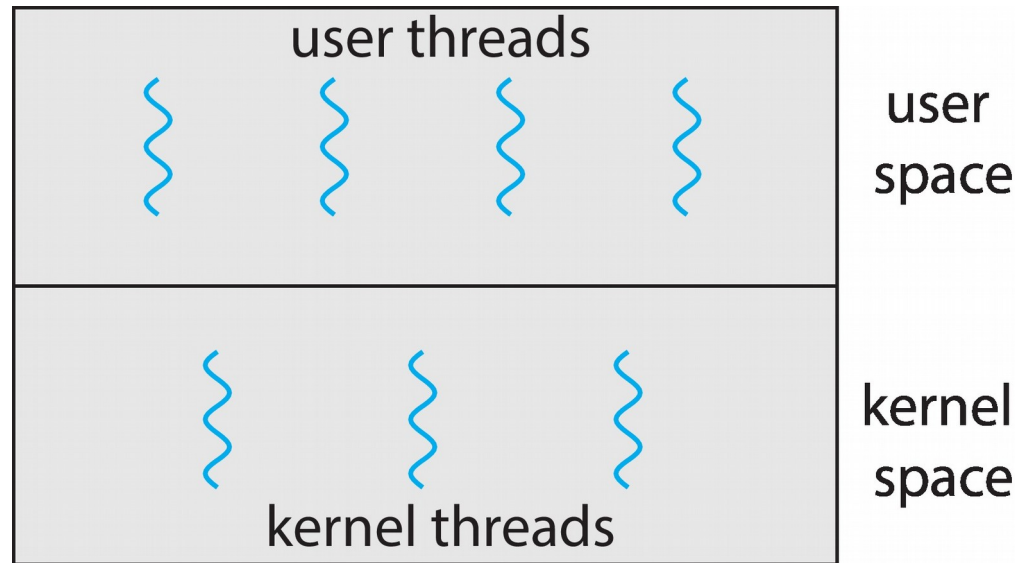
User Threads and Kernel Threads

- **User threads** are supported by user-level libraries

- Three primary user thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads

- **Kernel threads** are supported directly by the kernel
  - Examples – virtually all modern operating systems, including:
    - Windows
    - Linux
    - Mac OS X
    - iOS
    - Android

User and Kernel Threads

- When threads are provided as user threads, they still must be mapped onto kernel threads.

- There is not necessarily an equal number of user and kernel threads.

Multi-threading Models

- How to map user threads to kernel threads?
- Three different models:
  - Many-to-One: many user-level threads map to single kernel thread

  - One-to-One: one user-level thread maps to one kernel thread

  - Many-to-Many: many user-level threads map to many kernel threads

Many-to-One

- Many user-level threads mapped to single kernel thread.

- Weaknesses:

  - One thread blocking causes all to block

  - Multiple threads may not run in parallel on multi-core system because only one may be in kernel at a time

- Few systems currently use this model because modern systems have many cores which are not utilized well.

- Examples:

  - **Solaris Green Threads**

  - **GNU Portable Threads**
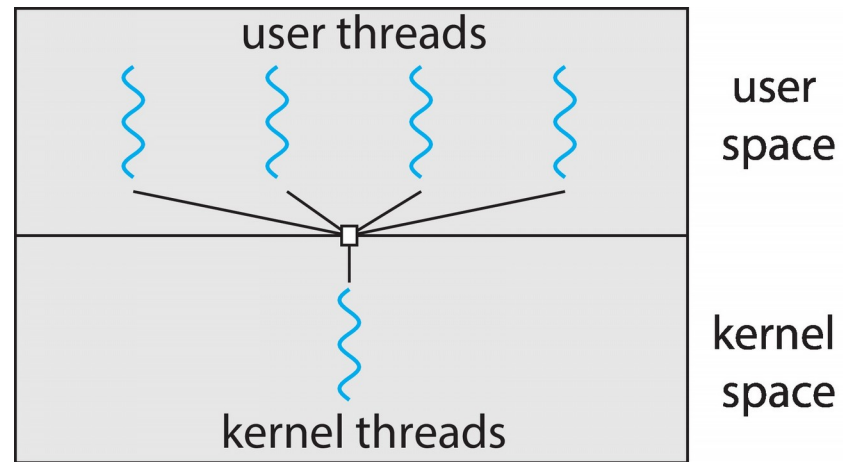
One-to-One

- Each user-level thread maps to one 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:

  - Creating a user thread requires creating a kernel thread, and too many kernel threads can degrade performance of system.

- Examples
  - Windows
 - Linux
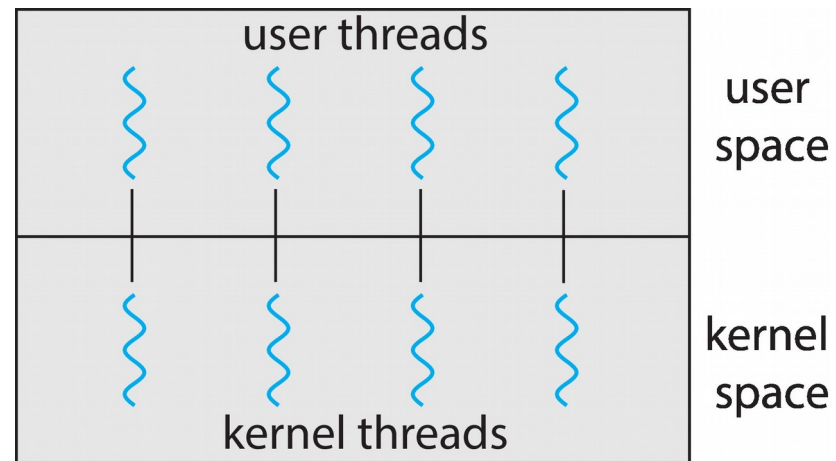
Many-to-Many Model

- Allows many user level threads to be multiplexed onto an equal or smaller number of kernel threads

- Allows the operating system to create a sufficient number of kernel threads

- Program can have as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. If thread blocks, kernel can schedule a different thread.

- Windows with the ThreadFiber package

- Otherwise not very common
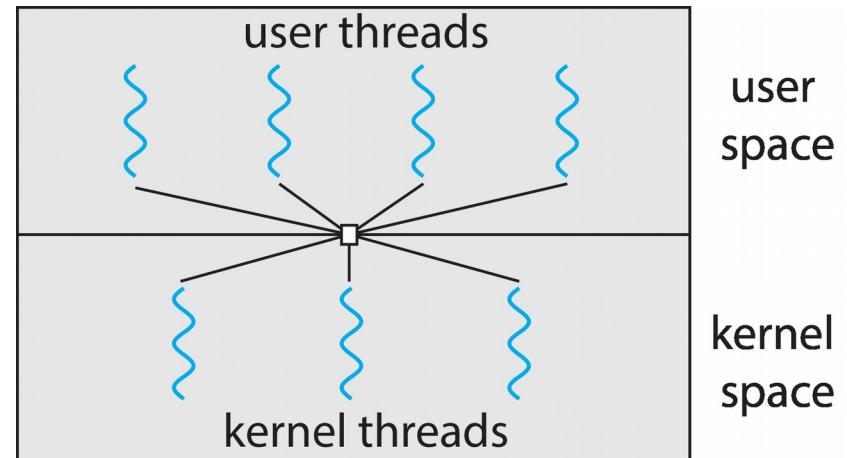
Two-level Model

- Similar to the many-to-many, except that it allows a user thread to be **bound** to a kernel thread.
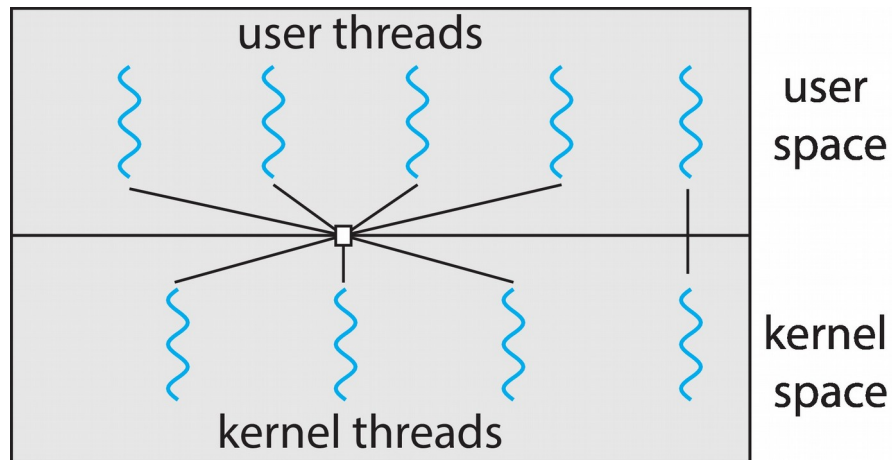
Thread Libraries

- **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

- Three prevalent libraries: POSIX threads (Pthreads), Windows, and Java threads.

Pthreads

- May be provided either as user-level or kernel-level

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization

- *Specification*, not *implementation*

- API specifies behavior of the thread library, implementation is up to development of the library

- Common in UNIX operating systems (Linux & Mac OS X)

Pthreads Example 1

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void* hello_world( void* unused)
{
    printf("The child says, \"Hello world!\"\n");
    pthread_exit(NULL) ;
}


int main( int argc, char *argv[])
{
    pthread_t  child_thread;

    /* Create the thread and launch it. */
    if ( 0 != pthread_create(&child_thread, NULL,
        hello_world, NULL ) ){
          printf("pthread_create failed.\n");
          exit(1);
    }
    printf("This is the parent thread.\n");
    /* Wait for the child thread to terminate. */
    pthread_join(child_thread, NULL);
    return 0;
}
```

Pthreads Example 2

```c
#include <pthread.h>
#include <stdio.h>

#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */

    /* set the default attributes of the thread */
    pthread_attr_init(&attr);
    /* create the thread */
    pthread_create(&tid, &attr, runner, argv[1]);
    /* wait for the thread to exit */
    pthread_join(tid,NULL);

    printf("sum = %d\n",sum);
}
```

Pthreads Example 2 (cont)

```
/* The thread will execute in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```
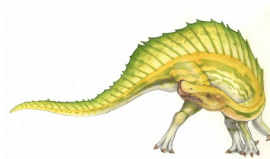
Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Windows Multi-threaded C Program

```c
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* The thread will execute in this function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 1; i <= Upper; i++)
      Sum += i;
   return 0;
}
```

```c
int main(int argc, char *argv[])
{
  DWORD ThreadId;
  HANDLE ThreadHandle;
  int Param;

  Param = atoi(argv[1]);
  /* create the thread */
  ThreadHandle = CreateThread(
    NULL, /* default security attributes */
    0, /* default stack size */
    Summation, /* thread function */
    &Param, /* parameter to thread function */
    0, /* default creation flags */
    &ThreadId); /* returns the thread identifier */

   /* now wait for the thread to finish */
  WaitForSingleObject(ThreadHandle,INFINITE);

  /* close the thread handle */
  CloseHandle(ThreadHandle);

  printf("sum = %d\n",Sum);
}
```

Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads

- Creation and management of threads done by compilers and run-time libraries rather than programmers

- Five methods explored

  - Thread Pools

  - Fork-Join

  - OpenMP

  - Grand Central Dispatch

  - Intel Threading Building Blocks

Thread Pools

- Create a number of threads in a pool where they await work

- Advantages:

  - Usually slightly faster to service a request with an existing thread than create a new thread

  - Allows the number of threads in the application(s) to be bound to the size of the pool

  - Separating task to be performed from mechanics of creating task allows different strategies for running task

    ‣ i.e.Tasks could be scheduled to run periodically
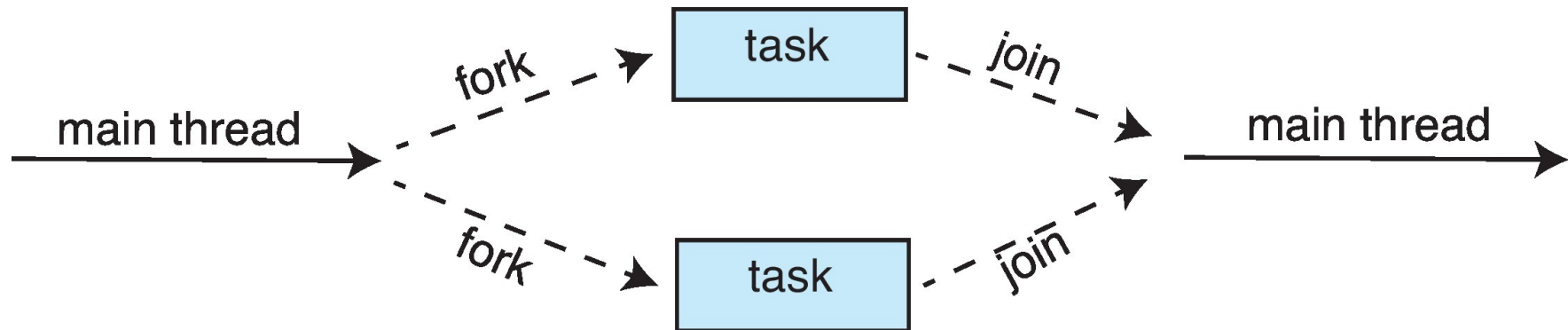
- Windows API supports thread pools:

```
DWORD WINAPI PoolFunction(AVOID Param) {
    /*
     * this function runs as a separate thread.
     */
}
```

Fork-Join Parallelism

■ Multiple threads (tasks) are **forked**, and then **joined**.
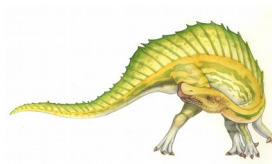
Fork-Join Parallelism

■ General algorithm for fork-join strategy:

```
Task(problem)
    if problem is small enough
        solve the problem directly
    else
        subtask1 = fork(new Task(subset of problem)
        subtask2 = fork(new Task(subset of problem)

        result1 = join(subtask1)
        result2 = join(subtask2)

        return combined results
```
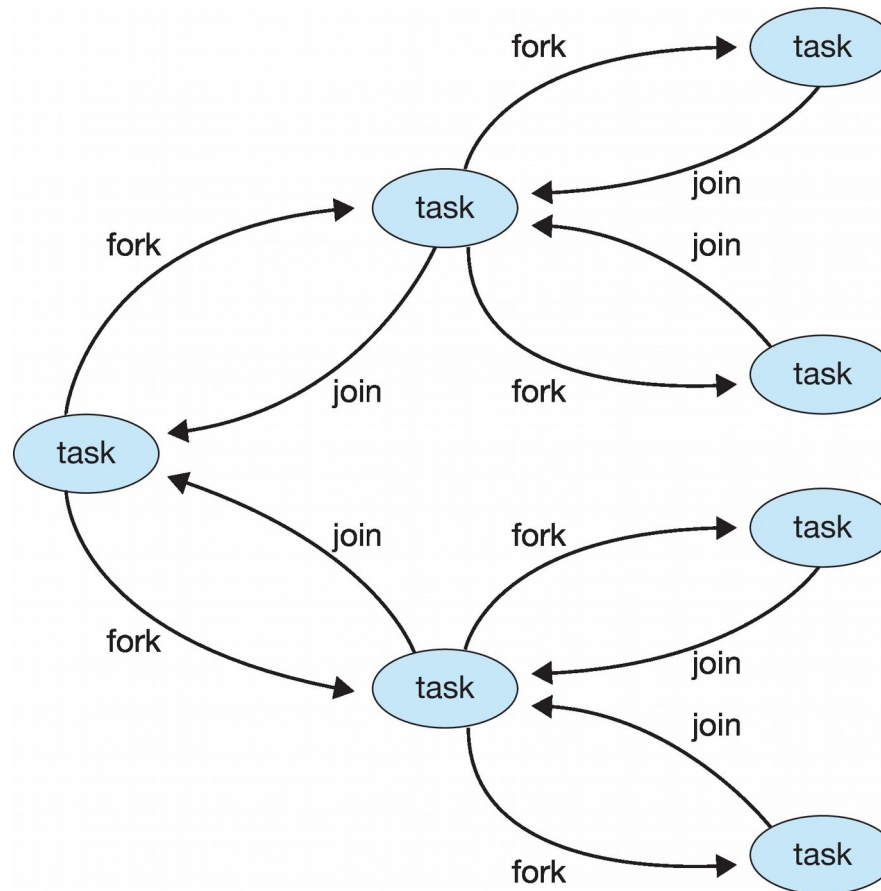
Fork-Join Parallelism

OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN

- Provides support for parallel programming in shared-memory environments

- Identifies **parallel regions** – blocks of code that can run in parallel

#pragma omp parallel

Create as many threads as there are cores

```c
#include <omp.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
   /* sequential code */

   #pragma omp parallel
   {
      printf("I am a parallel region.");
   }

   /* sequential code */

   return 0;
}
```
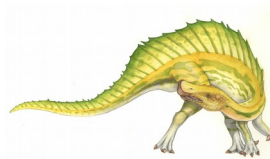
OpenMP Example

- Run the for loop in parallel

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
    c[i] = a[i] + b[i];
}
```

Grand Central Dispatch

- Apple technology for macOS and iOS operating systems

- Extensions to C, C++ and Objective-C languages, API, and run-time library

- Allows identification of parallel sections

- Manages most of the details of threading

- Block is in "^{ }" :

```
^{ printf("I am a block"); }
```

- Blocks placed in dispatch queue

  - Assigned to available thread in thread pool when removed from queue

Grand Central Dispatch (cont)

- Two types of dispatch queues:

  - **serial** – blocks removed in FIFO order, queue is per process, called **main queue**

    ‣ Programmers can create additional serial queues within program

  - **concurrent** – removed in FIFO order but several may be removed at a time

    ‣ Four system wide queues divided by quality of service:

    ○ QOS_CLASS_USER_INTERACTIVE

    ○ QOS_CLASS_USER_INITIATED

    ○ QOS_CLASS_USER_UTILITY

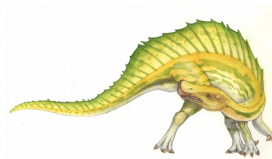   ○ QOS_CLASS_USER_BACKGROUND
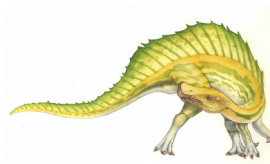
Grand Central Dispatch (3)

- For the Swift language a task is defined as a closure – similar to a block, minus the caret

- Closures are submitted to the queue using the `dispatch_async()` function:

```
let queue = dispatch_get_global_queue
    (QOS_CLASS_USER_INITIATED, 0)

dispatch_async(queue,{ print("I am a closure.") })
```
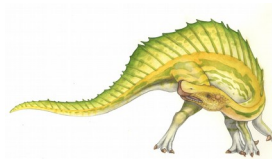
Threading Issues

- 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
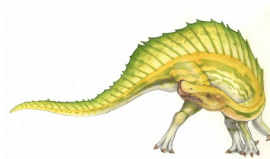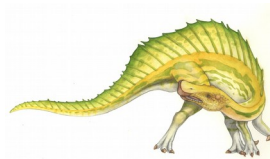
Signal Handling

- **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 particular event
  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
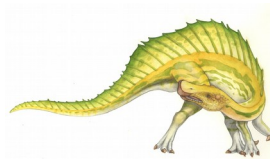
Signal Handling (Cont.)

- 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.)

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

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:M 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

user thread

LWP ← lightweight process
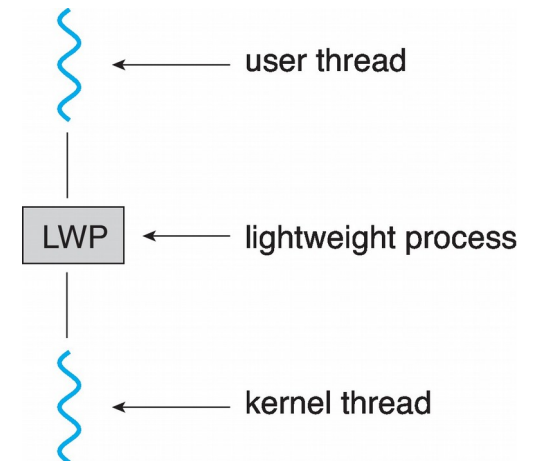
kernel thread

Operating System Examples

- Windows Threads
- Linux Threads
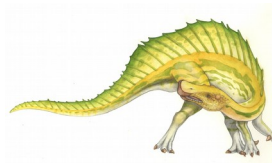
Windows Threads

- 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
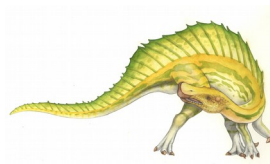
Windows Threads (Cont.)

- 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
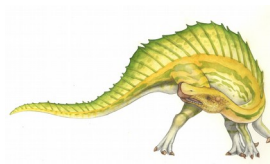
Windows Threads Data Structures
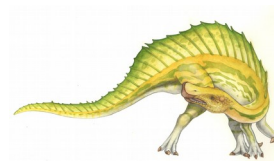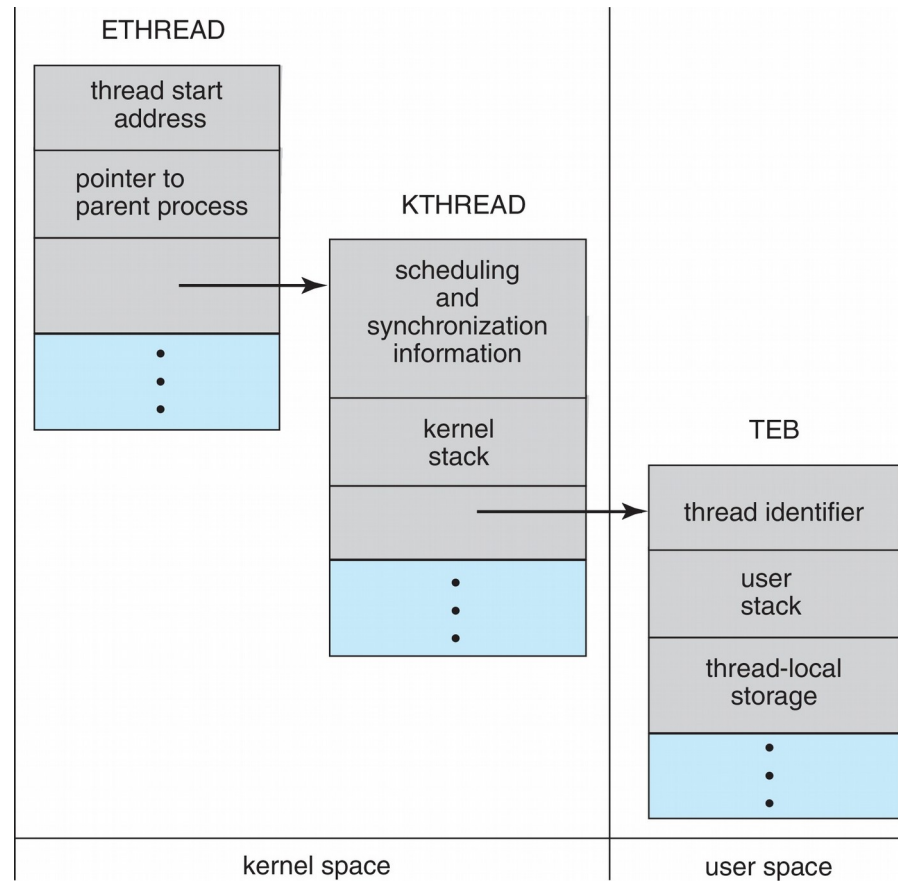
Linux Threads

- 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 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.

- `struct task_struct` points to process data structures (shared or unique)

End of Chapter 4