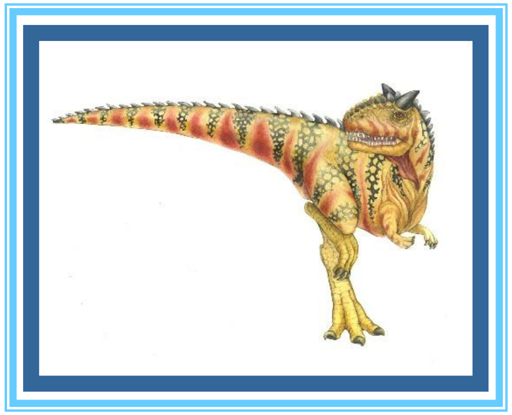
Operating Systems



Chapter 4:
Threads

Contact Information

Instructor: Engr. Shamila Nasreen

Lecturer

Department of Software Engineering

MUST

Email: shamila_nasreen131@yahoo.com

Office hours: Wednesday: 8:30-10:00 AM

Thursday: 8:30-10:00 AM

Books

- **Text Book:**

  Silberschatz, Galvin, “Operating Sytems Concepts” 8th Edition, John Wiley, 2007

Reference Book:

1. William Stallings, “Operating Systems”
2. Harvey M. Deitel, “Operating Systems Concepts”

Today’s Lecture Agenda

- Threads & Processes
- Single threaded processes
- Multi Threaded Processes
- Thread States
- Thread Synchronization
- Threads Types
  - User Level Threads
  - Kernel Level threads

Operating Systems: Internals and Design Principles

The basic idea is that the several components in any complex system will perform particular subfunctions that contribute to the overall function.

—THE SCIENCES OF THE ARTIFICIAL,

Herbert Simon

Threads and Processes

- Most operating systems therefore support two entities:
  - the process,
    - which defines the address space and general process attributes
  - the thread,
    - which defines a sequential execution stream within a process
- A thread is bound to a single process.
  - For each process, however, there may be many threads.
- Threads are the unit of scheduling
- Processes are *containers* in which threads execute

Processes and Threads

Traditional processes have two characteristics:

Resource Ownership

Process includes a virtual address space to hold the process image

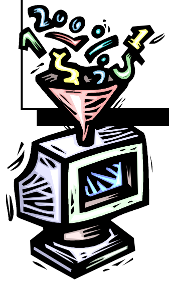
- the OS provides protection to prevent unwanted interference between processes with respect to resources

Scheduling/Execution

Follows an execution path that may be interleaved with other processes

- a process has an execution state (Running, Ready, etc.) and a dispatching priority and is scheduled and dispatched by the OS
- Traditional processes are *sequential*; i.e. only *one* execution path

Processes and Threads



- ***Multithreading -*** The ability of an OS to support multiple, concurrent paths of execution within a single process
- The unit of resource ownership is referred to as a ***process*** or ***task***
- The unit of dispatching is referred to as a ***thread*** or ***lightweight process***

Single Threaded Approaches

- A single execution path per process, in which the concept of a thread is not recognized, is referred to as a single-threaded approach
- MS-DOS, some versions of UNIX supported only this type of process.

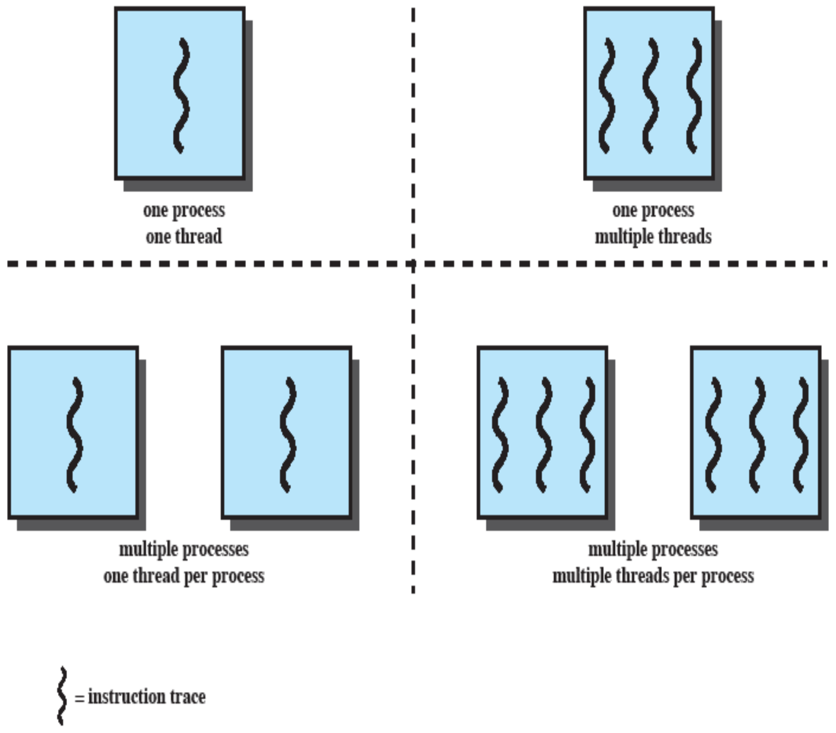


Figure 4.1 Threads and Processes [ANDE97]

Multithreaded Approaches

- The right half of Figure 4.1 depicts multithreaded approaches
- A Java run-time environment is a system of *one* process with multiple threads; Windows, some UNIX, support *multiple* multithreaded processes.

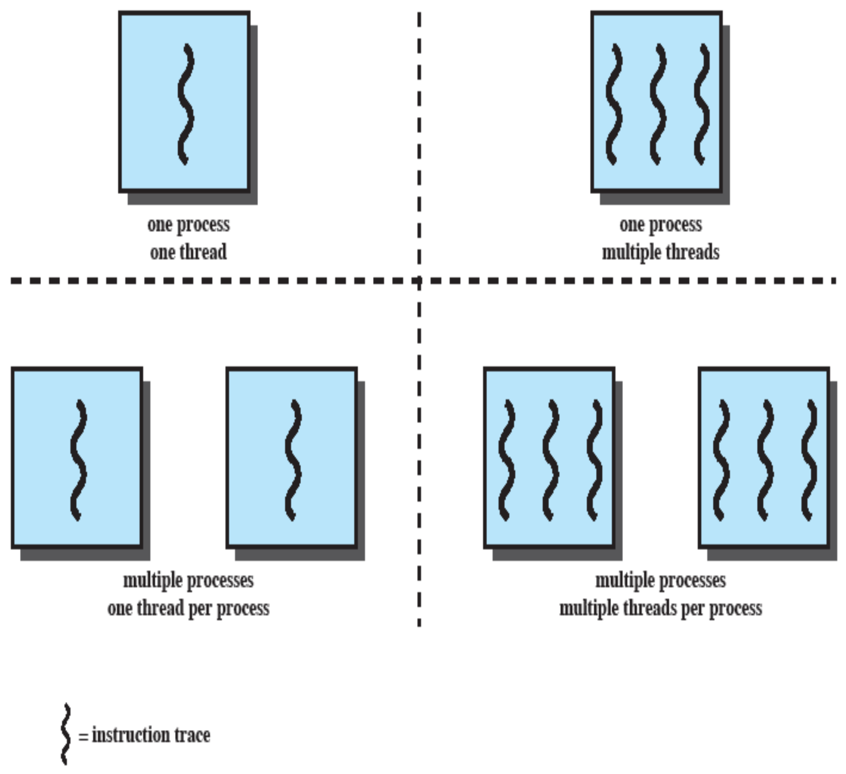
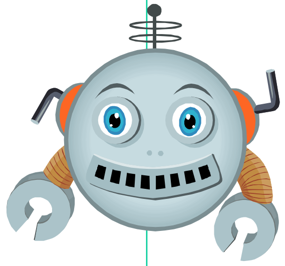


Figure 4.1 Threads and Processes [ANDE97]

One or More Threads in a Process

Each thread has:

- an execution state (Running, Ready, etc.)
- saved thread context when not running (TCB)
- an execution stack
- some per-thread static storage for local variables
- access to the shared memory and resources of its process (all threads of a process share this)



Threads vs. Processes

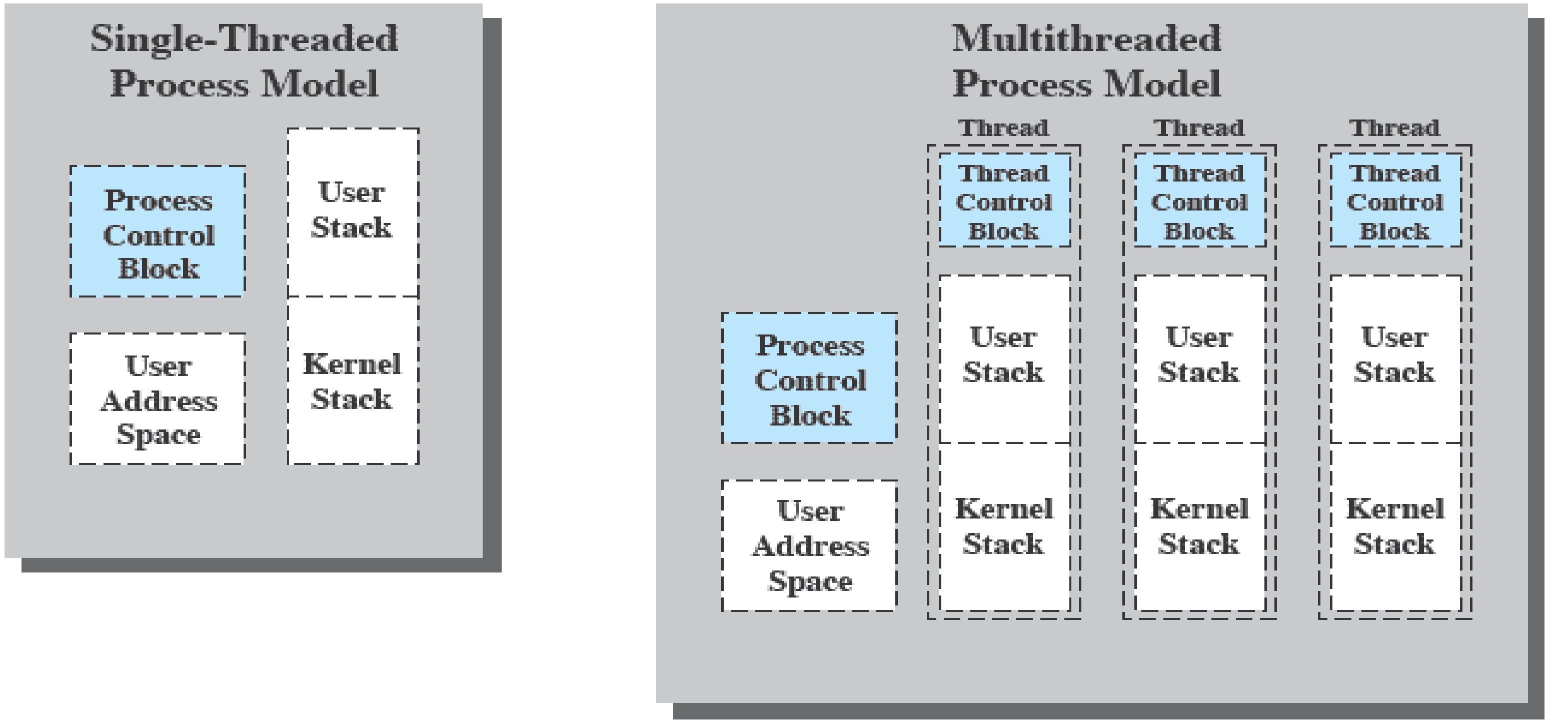
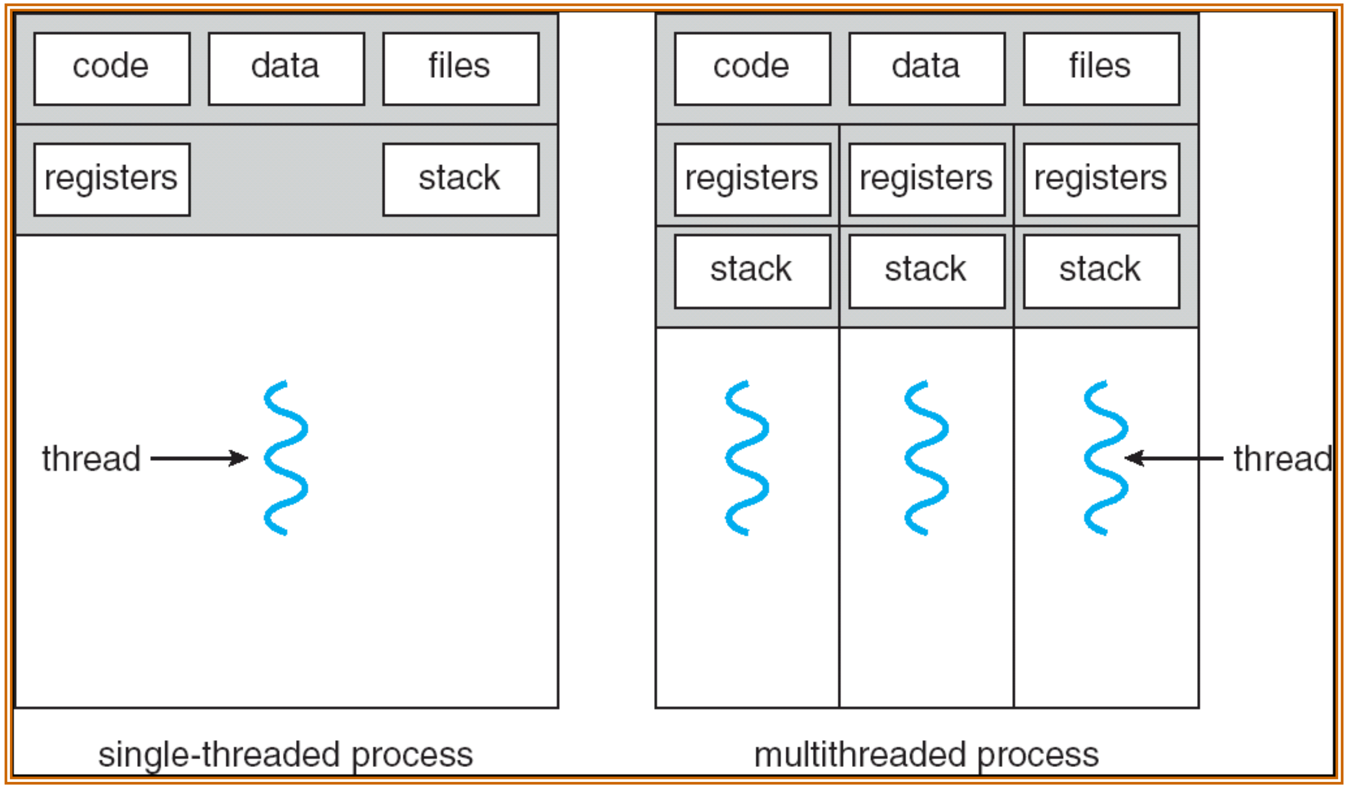


Figure 4.2 Single Threaded and Multithreaded Process Models

Single and Multithreaded Processes

1.Process = [Code + Data + Heap + Stack + PC + PCB + CPU Registers]

1.Thread = [Parent Code, Data and Heap + Stack, PC and CPU Registers]



Benefits of Threads

- The benefits of multithreaded programming can be broken down into four major categories:
- **Responsiveness.** Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
- **Resource Sharing:** By default, threads share the memory and the resources of the process to which they belong.
  - The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.
- **Economy:** Allocating memory and resources for process creation is costly.
  - threads share resources of the process to which they belong, it is more economical to create and context-switch threads.
- **Utilization of multiprocessor architectures:** The benefits of multithreading can be greatly increased in a multiprocessor architecture.
  - threads may be running in parallel on different processors.

Benefits of Threads

Takes less time to create a new thread than a process

Less time to terminate a thread than a process

Switching between two threads takes less time than switching between processes

Threads enhance efficiency in communication between programs

Thread Use in a Single-User System

- Foreground and background work
- Asynchronous processing
- Speed of execution
- Modular program structure

Threads

- In an OS that supports threads, scheduling and dispatching is done on a thread basis

Most of the state information dealing with execution is maintained in thread-level data structures

  - suspending a process involves suspending all threads of the process
  - termination of a process terminates all threads within the process

Thread Execution States

The key states for

a thread are:

- Running
- Ready
- Blocked
- Suspend???

Thread operations associated with a change in thread state are:

- Spawn (create)
- Block
- Unblock
- Finish

Thread transitions

- **Spawn:** Typically, when a new process is spawned, a thread for that process is also spawned.
  - Subsequently, a thread within a process may spawn another thread within the same process, providing an instruction pointer and arguments for the new thread.
  - The new thread is provided with its own register context and stack space and placed on the ready queue.
- **Block:** When a thread needs to wait for an event, it will block (saving its user registers, program counter, and stack pointers).
  - The processor may now turn to the execution of another ready thread in the same or a different process.
- **Unblock:** When the event for which a thread is blocked occurs, the thread is moved to the Ready queue.
- **Finish:** When a thread completes, its register context and stacks are deallocated.

Multithreading on a Uniprocessor

- On a uniprocessor, multiprogramming enables the interleaving of multiple threads within multiple processes.
- Execution passes from one thread to another either when the currently running thread is blocked or when its time slice is exhausted.

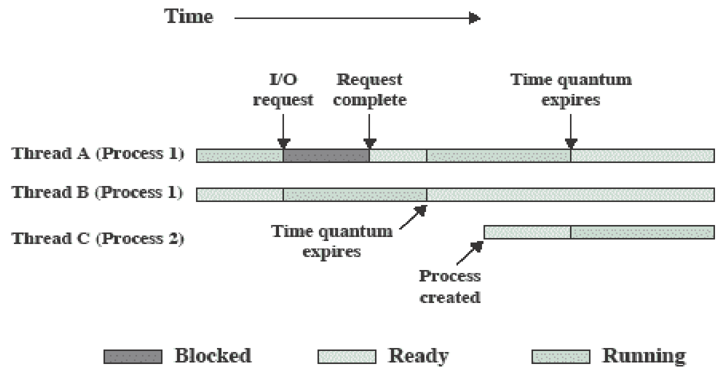
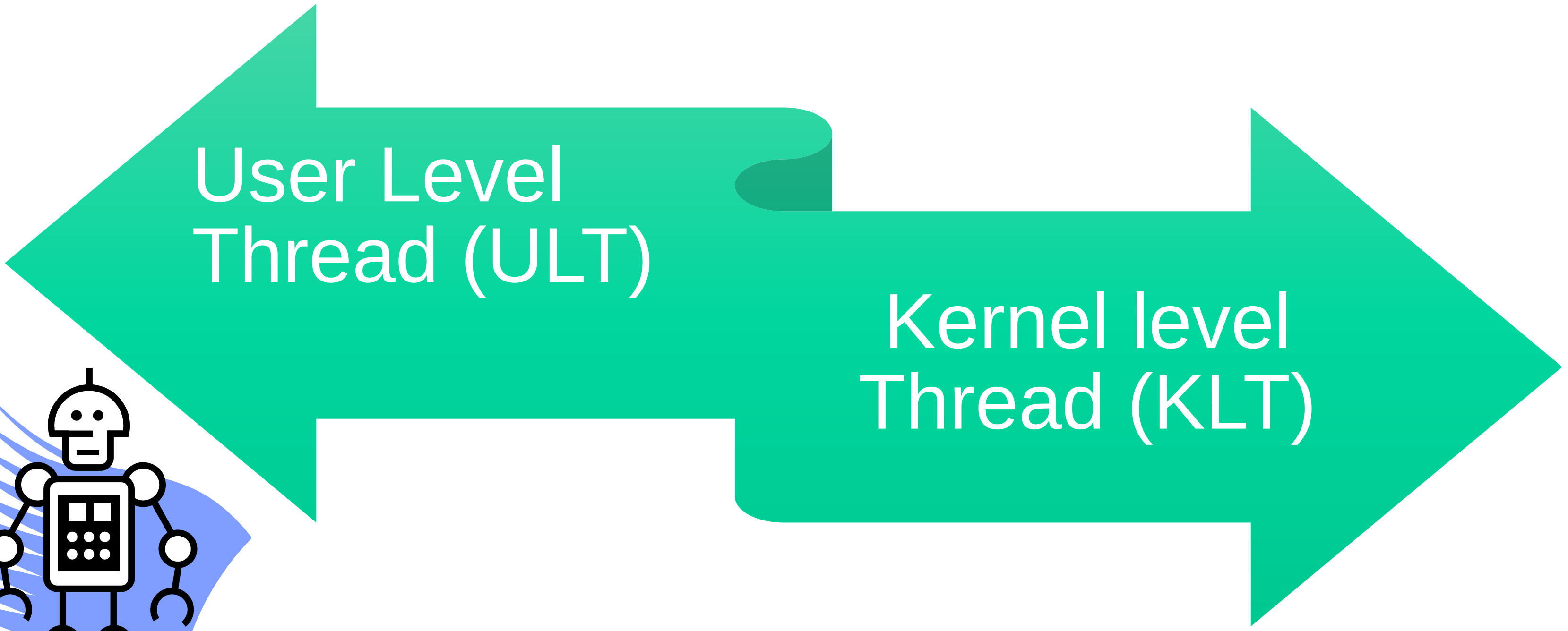


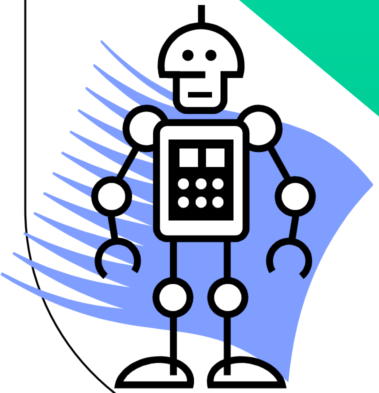
Figure 4.4 Multithreading Example on a Uniprocessor

Thread Synchronization

- It is necessary to synchronize the activities of the various threads
    - all threads of a process share the same address space and other resources
    - any alteration of a resource by one thread affects the other threads in the same process

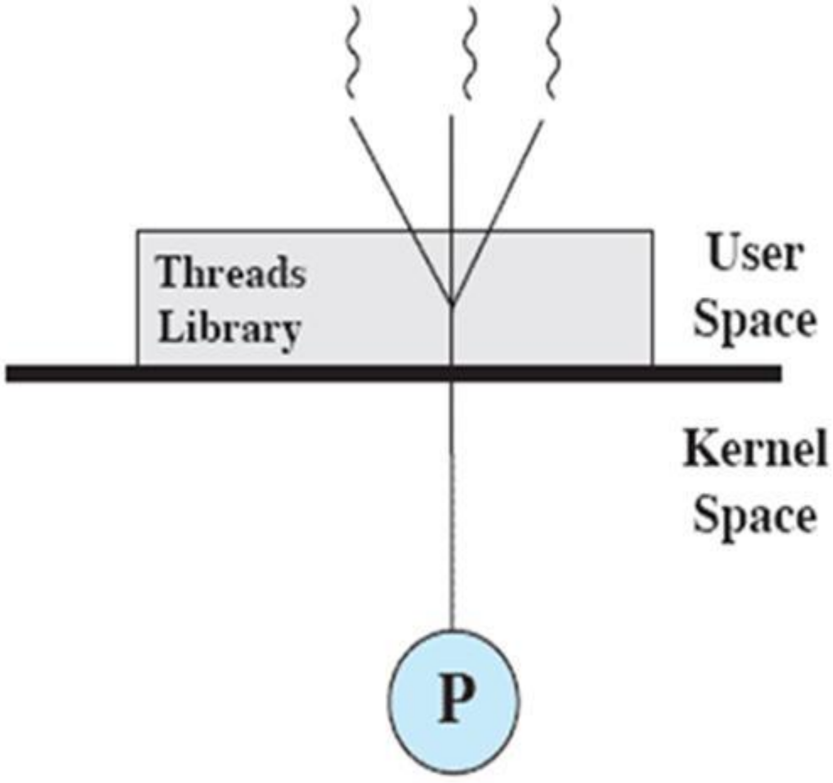
Types of Threads





User-Level Threads (ULTs)

- Thread management is done by the application(user level library)
- The kernel is not aware of the existence of threads
- Not the kind we've discussed so far.
- still require a kernel system call to operate



(a) Pure user-level

Relationships Between ULT States and Process States

Possible transitions from 4.6a:

4.6a→4.6b
4.6a→4.6c
4.6a→4.6d

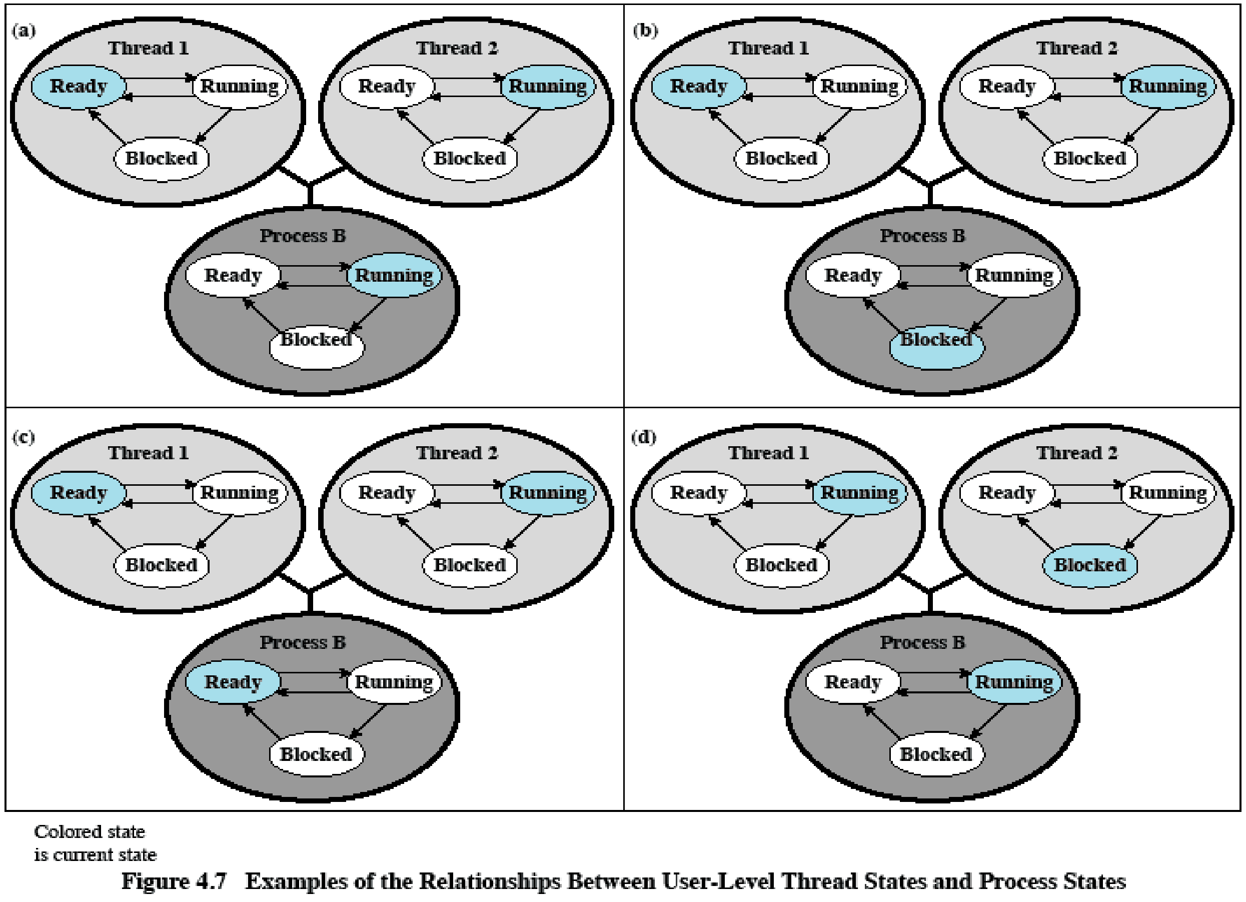
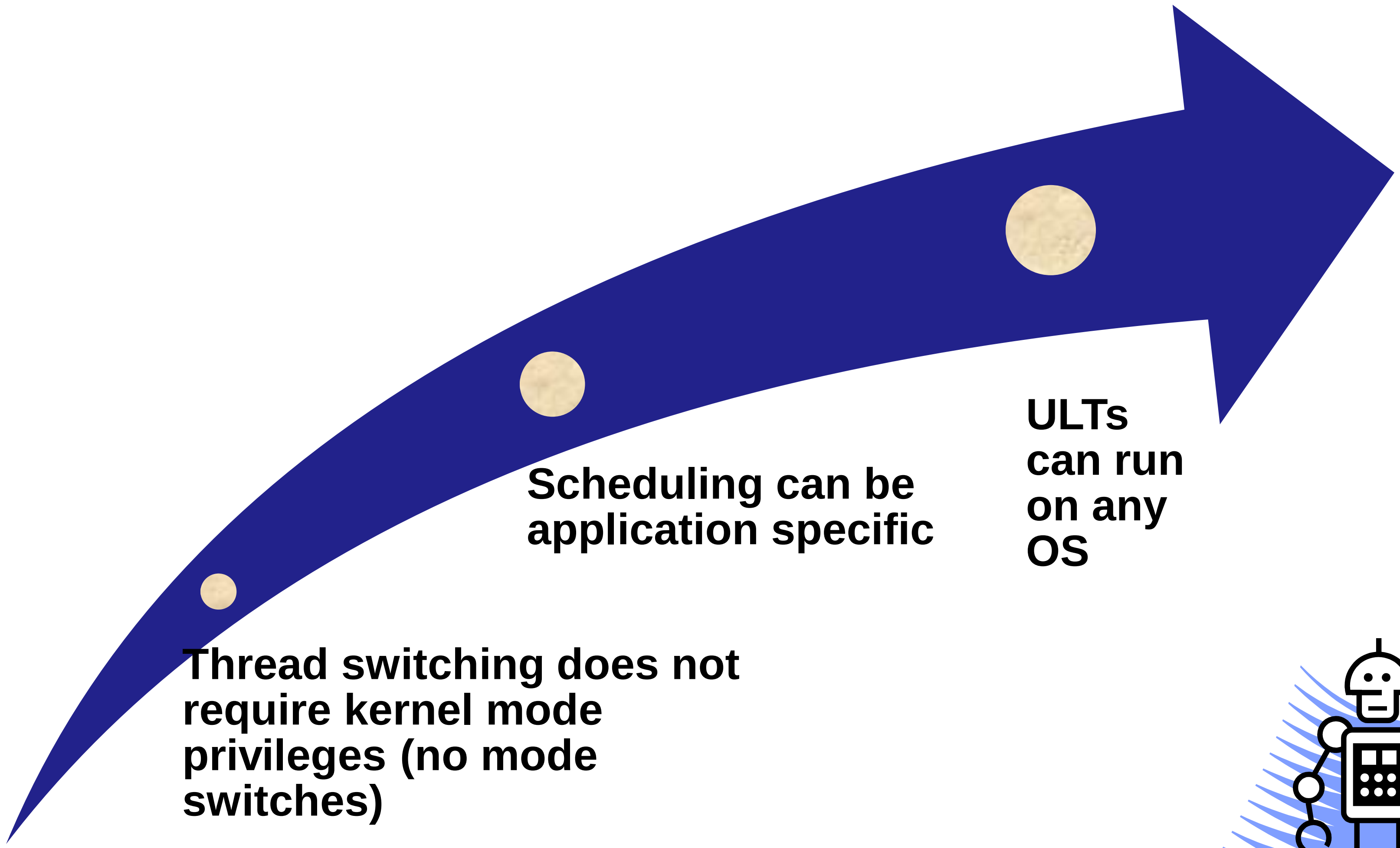

(a)
Thread 1
Ready
Running
Blocked
Thread 2
Ready
Running
Blocked
Process B
Ready
Running
Blocked
(b)
(c)
(d)
Colored state is current state
Figure 4.7 Examples of the Relationships Between User-Level Thread States and Process States

Figure 4.6 Examples of the Relationships between User-Level Thread States and Process States

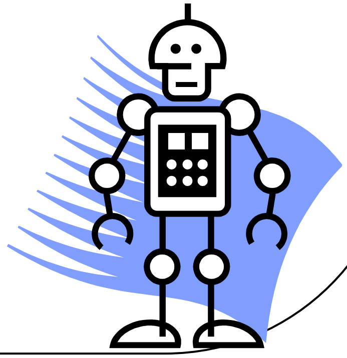
Advantages of ULTs



Thread switching does not require kernel mode privileges (no mode switches)

Scheduling can be application specific

ULTs can run on any OS



Disadvantages of ULTs

- In a typical OS many system calls are blocking
  - as a result, when a ULT executes a system call, not only is that thread blocked, but all of the threads within the process are blocked
- In a pure ULT strategy, a multithreaded application cannot take advantage of multiprocessing

Overcoming ULT Disadvantages

Jacketing

- converts a blocking system call into a non-blocking system call

Writing an application as multiple processes rather than multiple threads

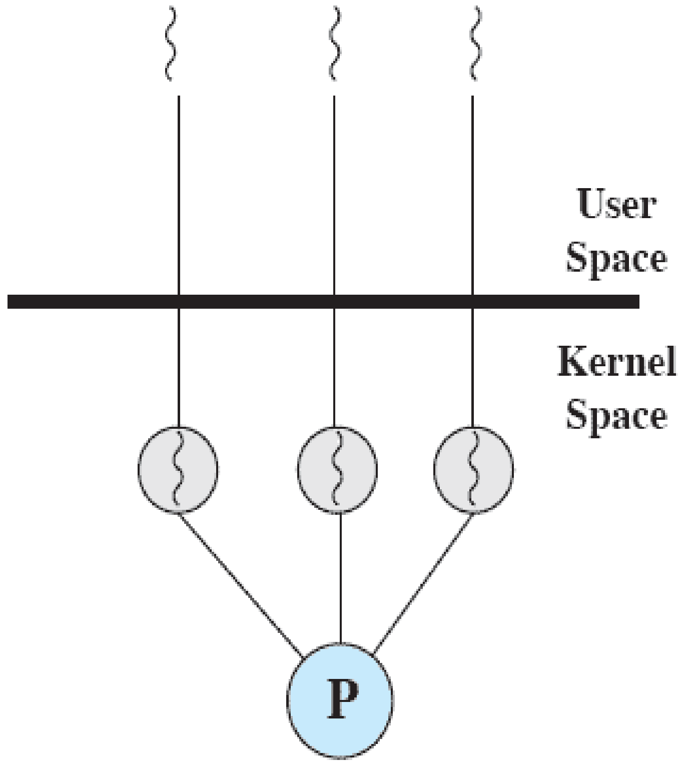
Overcoming ULT Disadvantages

writing an application as multiple processes:

- writing an application as multiple processes rather than multiple threads.
- This approach eliminates the main advantage of threads
  - Each switch becomes a process switch rather than a thread switch, resulting in much greater overhead

- **Jacketing:**
  - The purpose of jacketing is to convert a blocking system call into a non-blocking system call.
  - For example, instead of directly calling a system I/O routine, a thread calls an application-level I/O jacket routine.
    - Within this jacket routine is code that checks to determine if the I/O device is busy.
    - If it is, the thread enters the Blocked state and passes control (through the threads library) to another thread.
    - When this thread later is given control again, the jacket routine checks the I/O device again.

Kernel-Level Threads (KLTs)



(b) Pure kernel-level

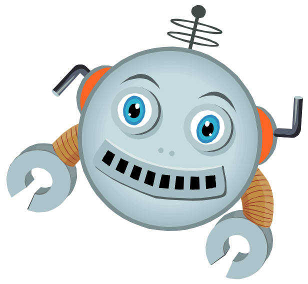
- Thread management is done by the kernel (could call them K*M*T)
  - no thread management is done by the application
  - Windows is an example of this approach

Advantages of KLTs

- The kernel can simultaneously schedule multiple threads from the same process on multiple processors
- This approach overcomes the two principal drawbacks of the ULT approach:

1. First, the kernel can simultaneously schedule multiple threads from the same process on multiple processors.
2. If one thread in a process is blocked, the kernel can schedule another thread of the same process

- kernel routines themselves can be multithreaded.

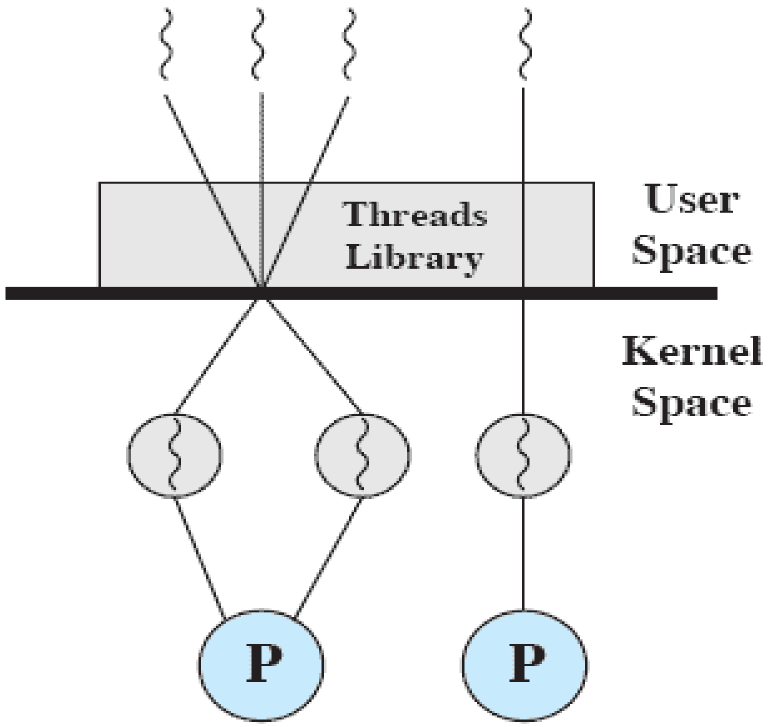


Disadvantage of KLTs

- The transfer of control from one thread to another within the same process requires a mode switch to the kernel

Combined Approaches

- Thread creation is done in the user space
- Bulk of scheduling and synchronization of threads is by the application
- The multiple ULTs from a single application are mapped onto some (smaller or equal) number of KLTs.
- Solaris is an example
- In a combined approach, multiple threads within the same application can run in parallel on multiple processors, and a blocking system call need not block the entire process



(c) Combined

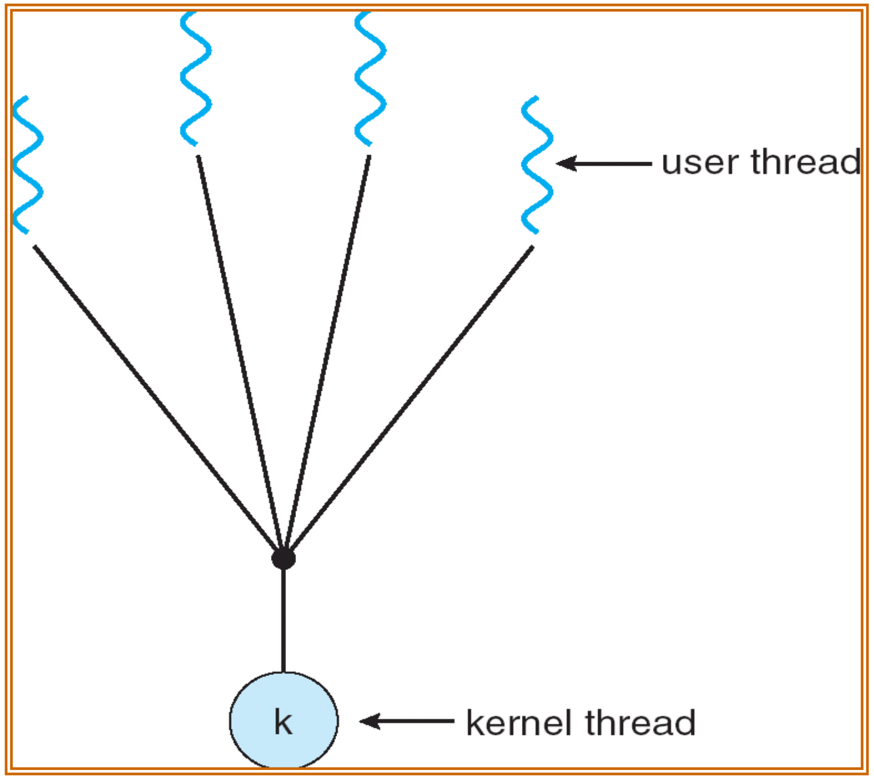
Multithreading Models

Mapping user threads to kernel threads:

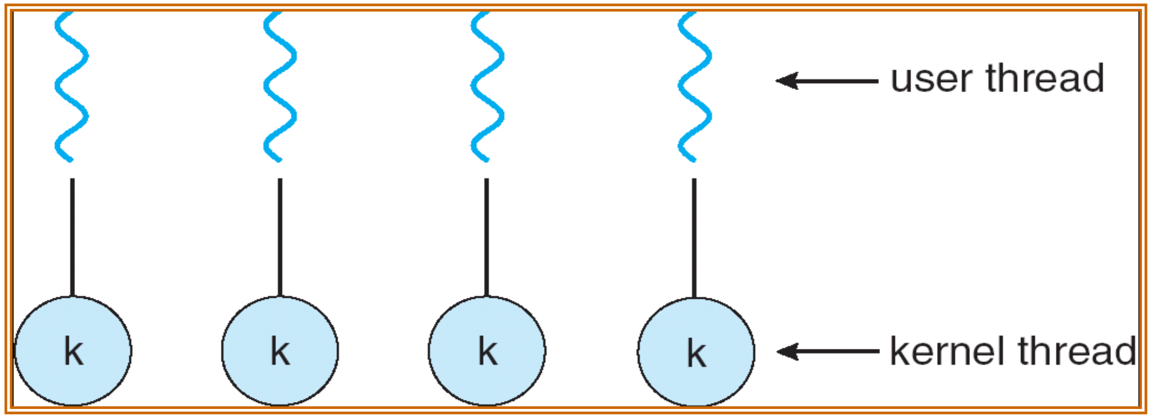
- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

- Many user-level threads mapped to single kernel thread
  - Fast - no system calls required
  - No parallel execution of threads - can’t exploit multiple CPUs
  - All threads block when one uses synchronous I/O
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads



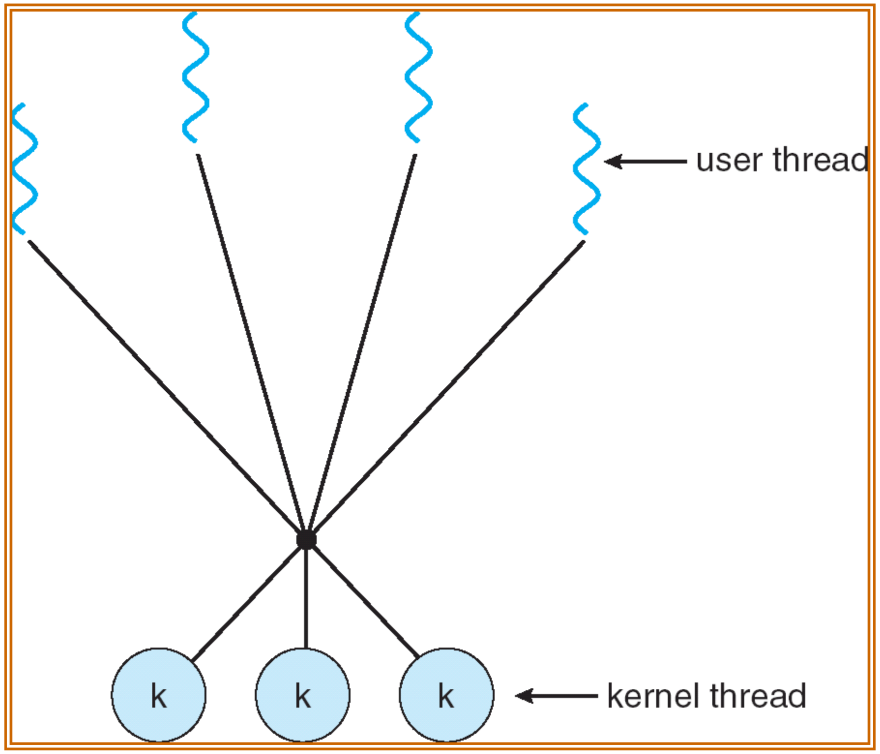
One-to-One



- Each user-level thread maps to kernel thread
  - More concurrency
  - Better multiprocessor performance
  - Each user thread requires creation of kernel thread
  - Each thread requires kernel resources; limits number of total threads
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

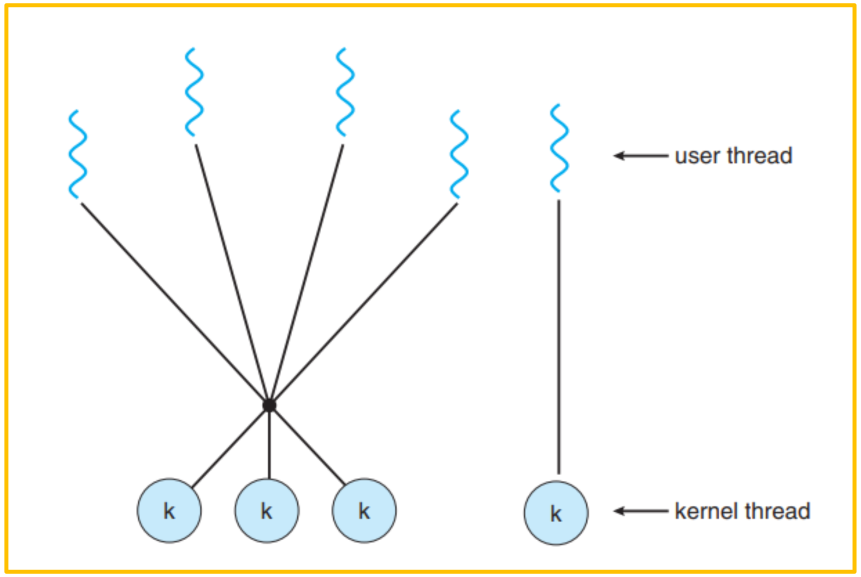
Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
  - Allows the operating system to create a sufficient number of kernel threads
  - If U < L, no benefits of multithreading
  - If U > L, some threads may have to wait for an LWP to run
- Examples:
  - Solaris prior to version 9
  - Windows NT/2000 with the *ThreadFiber* package



Many-to-Many Model(two Level Model)

- Variation of many-to-many model
- still multiplexes many user level threads to a smaller or equal number of kernel threads
- but also allows a user-level thread to be bound to a kernel thread.
  Examples:
  - The Solaris operating system supported the two-level model in versions older than Solaris 9.
  - However, beginning with Solaris 9, this system uses the one-to-one model.



Activity

- What do you think which threads Model(M-1, 1-1, M-M) these threads types follows?

- ULT’s:??
- KLT’s:??
- Hybrid approach:??

OpenMP

- OpenMP is a set of compiler directives as well as an API for programs written in C, C++, or FORTRAN that provides support for parallel programming in shared-memory environments.
  - identifies **parallel regions** as blocks of code that may run in parallel.
  - Application developers insert compiler directives into their code at parallel regions
  - these directives instruct the OpenMP run-time library to execute the region in parallel.

```
#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;
}
```

When OpenMP encounters the directive
`#pragma omp parallel`

it creates as many threads are there
are processing cores in the system.
Thus, for
a dual-core system, two threads are
created, for a quad-core system four threads

OpenMP

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

OpenMP divides the work contained in the `for` loop among the threads it has created in response to the directive

```
#pragma omp parallel for
```

Grand Central Dispatch (GCD)

- Grand Central Dispatch (GCD)—a technology for Apple's Mac OS X and iOS operating systems
- is a combination of extensions to the C language, an API, and a run-time library that allows application developers to identify sections of code to run in parallel.
- Like OpenMP, GCD manages most of the details of threading.
- GCD identifies extensions to the C and C++ languages known as **blocks**.
- A block is simply a self-contained unit of work.
- It is specified by a caret ˆ inserted in front of a pair of braces *{ }*.
- A simple example of a block is shown below:

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

Grand Central Dispatch (GCD)

- GCD schedules blocks for run-time execution by placing them on a **dispatch queue**. When it removes a block from a queue, it assigns the block to an available thread from the thread pool it manages.
- GCD identifies two types of dispatch queues: ***serial*** and ***concurrent***.
- Serial queues: Blocks placed on a serial queue are removed in FIFO order. Once a block has been removed from the queue, it must complete execution before another block is removed.
  - Each process has its own serial queue (known as its **main queue**). Developers can create additional serial queues that are local to particular processes.
  - Serial queues are useful for ensuring the sequential execution of several tasks

Grand Central Dispatch (GCD)

- Concurrent queues:
  Blocks placed on a concurrent queue are also removed in FIFO order, but several blocks may be removed at a time, thus allowing multiple blocks to execute in parallel.
  - There are three system-wide concurrent dispatch queues, and they are distinguished according to priority: low, default, and high.
  - The following code segment illustrates obtaining the default-priority concurrent queue and submitting a block to the queue using the `dispatch async()` function:

```
dispatch_queue_t queue = dispatch_get_global_queue
   (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_async(queue, ^{ printf("I am a block."); });
```

Threading Issues

- Thread cancellation of target thread
  - Asynchronous or deferred
- Signal handling
- Thread pools
- Thread-specific data
- Scheduler Activations

Threading Issues-Thread Cancellation

- Terminating a thread before it has finished
- A thread that is to be canceled is often referred to as the **target thread**.

  Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately.
    - it is troublesome if a thread to be canceled is in the middle of updating shared data
    - Asynchronous cancellation can occur at any time
  - **Deferred cancellation**
    - sets a flag indicating the thread should cancel itself when it is convenient
    - allows the target thread to periodically check if it should be cancelled – allow threads to be canceled safely
    - Deferred cancellation is the default type.

Threading Issues-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
  - Options:
    - 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

  3. Signal is handled

     A signal may be handled by one of two possible handlers:

     1. A default signal handler
     2. A user-defined signal handler

Thread Pools

- Create a number of threads in a pool where they await work
- A thread pool is a group of pre-instantiated, idle threads which stand ready to be given work.
- preferred over instantiating new threads for each task
- prevents having to incur the overhead of creating a thread a large number of times.
- 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**

Thread Pools

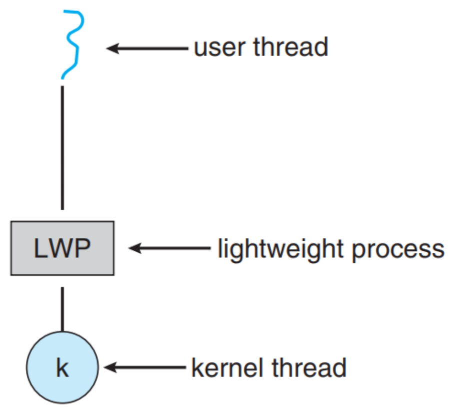
- Implementation will vary by environment, but in simplified terms, you need the following:
  - A way to create threads and hold them in an idle state. This can be accomplished by having each thread wait at a barrier until the pool hands it work.
    - (This could be done with mutexes as well.)
  - A container to store the created threads, such as a queue or any other structure that has a way to add a thread to the pool and pull one out.
  - A standard interface or abstract class for the threads to use in doing work.
    - This might be an abstract class called Task with an execute() method that does the work and then returns.

Thread Specific Data

- Allows each thread to have its own copy of data
- Most data is shared among threads, and this is one of the major benefits of using threads in the first place.
- However sometimes threads need thread-specific data also.
- Most major thread libraries (pThreads, Win32, Java) provide support for thread-specific data, known as thread-local storage or TLS.

Scheduling Activation

- A final issue to be considered with multithreaded programs concerns communication between the kernel and the thread library (required with m-m model and two level model).
- Uses a s data structure—typically known as a **lightweight process**, or **LWP**
  To the user-thread library, the LWP appears to be a virtual processor on which the application can schedule a user thread to run.
- Each LWP is attached to a kernel thread, and it is kernel threads that the operating system schedules to run on physical processors.
- If a kernel thread blocks (such as while waiting for an I/O operation to complete), the LWP blocks as well.
  Up the chain, the user-level thread attached to the LWP also blocks .



Scheduling Activation

- One scheme for communication between the user-thread library and the kernel is known as **scheduler activation**. It works as follows:
  - The kernel provides an application with a set of virtual processors (LWPs).
  - The application can schedule user threads onto an available virtual processor.
  - The kernel must inform an application about certain events. This procedure is known as an **upcall**. Upcalls are handled by the thread library with an **upcall handler**, and upcall handlers must run on a virtual processor.
  - One event that triggers an upcall occurs when an application thread is about to block.
    - In this scenario, the kernel makes an upcall to the application informing it that a thread is about to block and identifying the specific thread.
    - The kernel then allocates a new virtual processor to the application. The application runs an upcall handler on this new virtual processor, which saves the state of the blocking thread and relinquishes the virtual processor on which the blocking thread is running.
    - The upcall handler then schedules another thread that is eligible to run on the new virtual processor.
    - When the event that the blocking thread was waiting for occurs, the kernel makes another upcall to the thread library informing it.