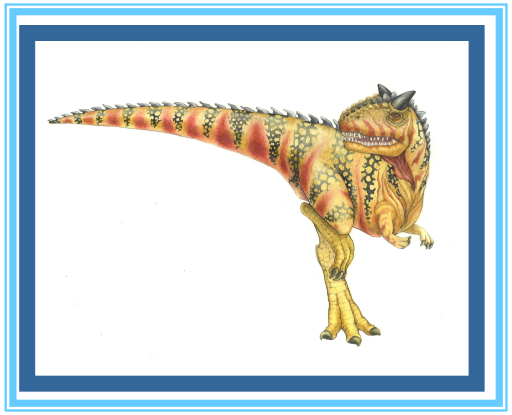
Chapter 4: Threads



Changes by MA Doman 2013

Chapter 4: Threads

- Overview
- Multithreading Models
- Thread Libraries
- Threading Issues
- Operating System Examples
- Windows XP Threads
- Linux Threads

Objectives

- To introduce the notion of a thread — a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To discuss the APIs for the Pthreads, Win32, and Java thread libraries
- To examine issues related to multithreaded programming

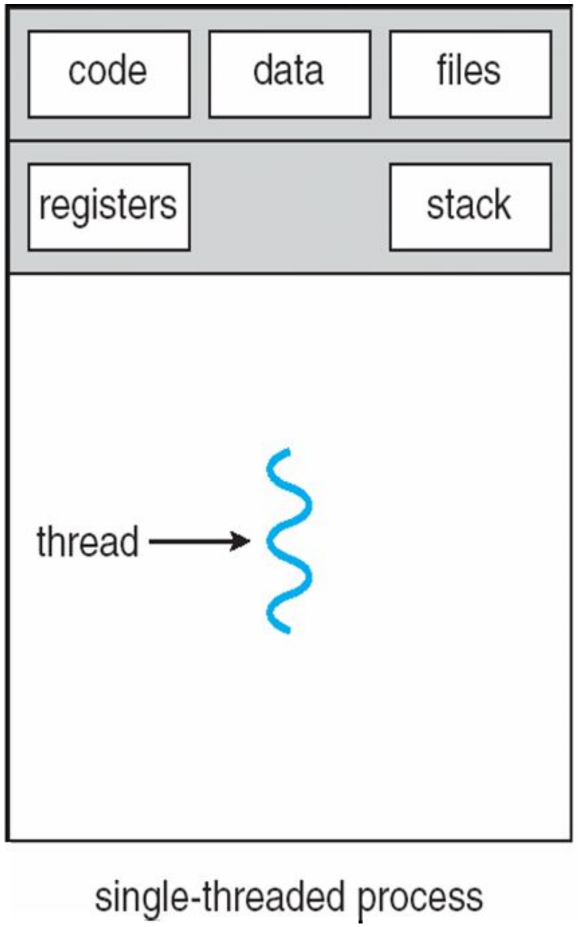
Thread Overview

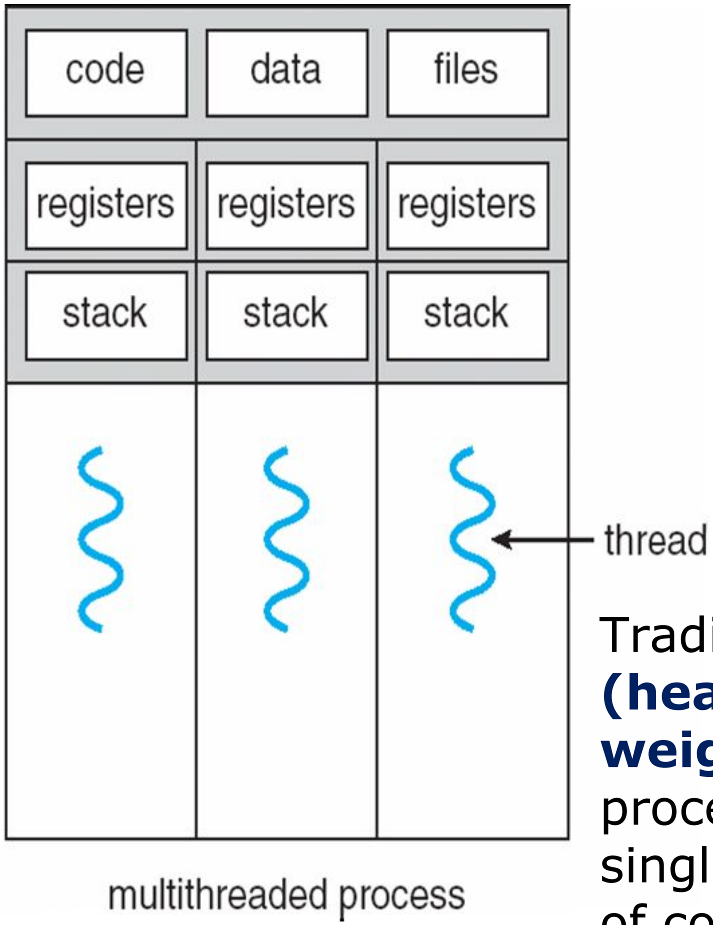
- Threads are mechanisms that permit an application to perform multiple tasks concurrently.

- Thread is a basic unit of CPU utilization
  - Thread ID
  - Program counter
  - Register set
  - Stack

- A single program can contain multiple threads
  - Threads share with other threads belonging to the same process
    - 4 Code, data, open files…….

Single and Multithreaded Processes





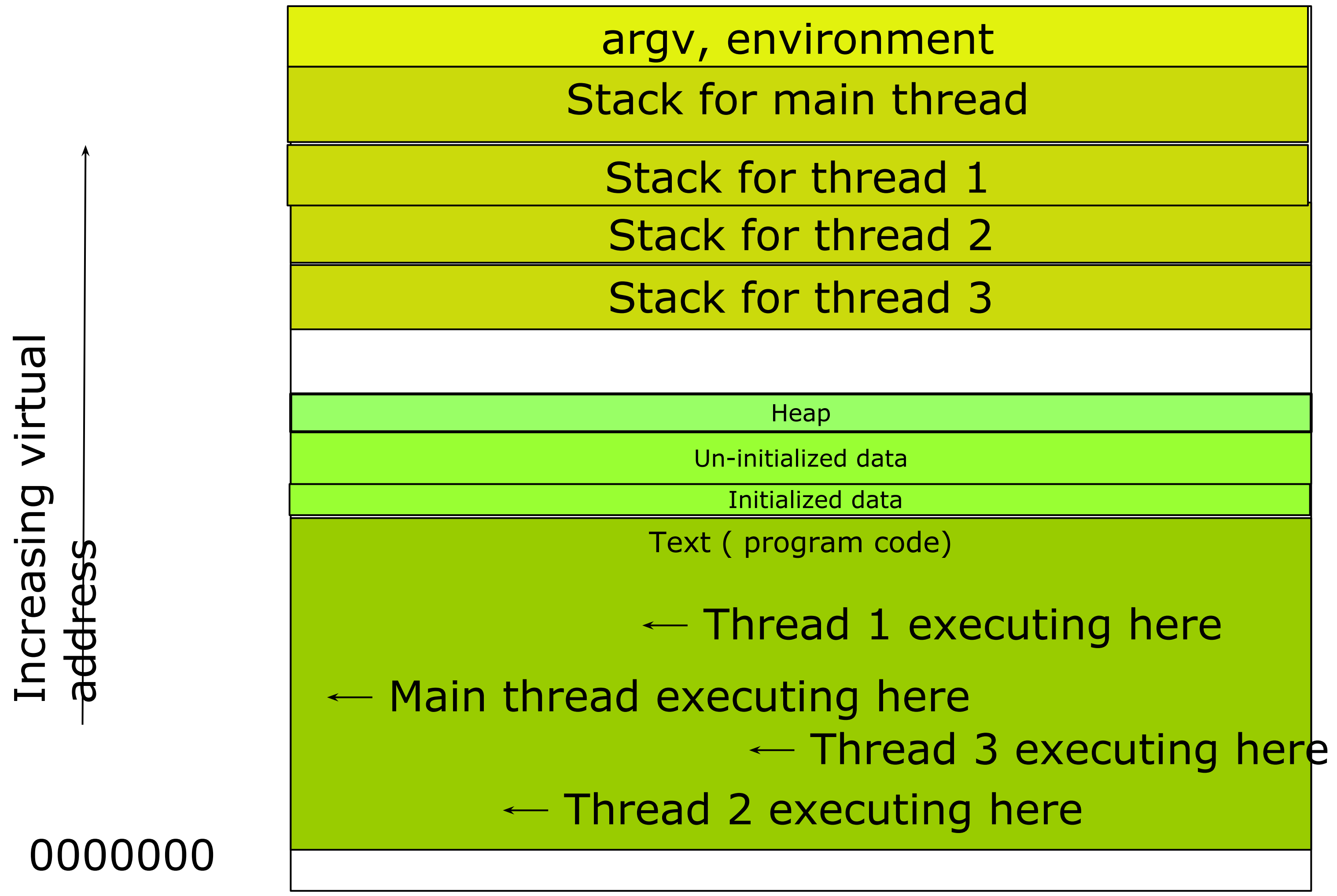
Traditional **(heavy weight)** process has a single thread of control

heavyweight process

lightweight process

Threads in Memory

Memory is allocated for a process in segments or parts:



Threads

Threads share….

- Global memory
- Process ID and parent process ID
- Controlling terminal
- Process credentials (user)
- Open file information
- Timers
- ………

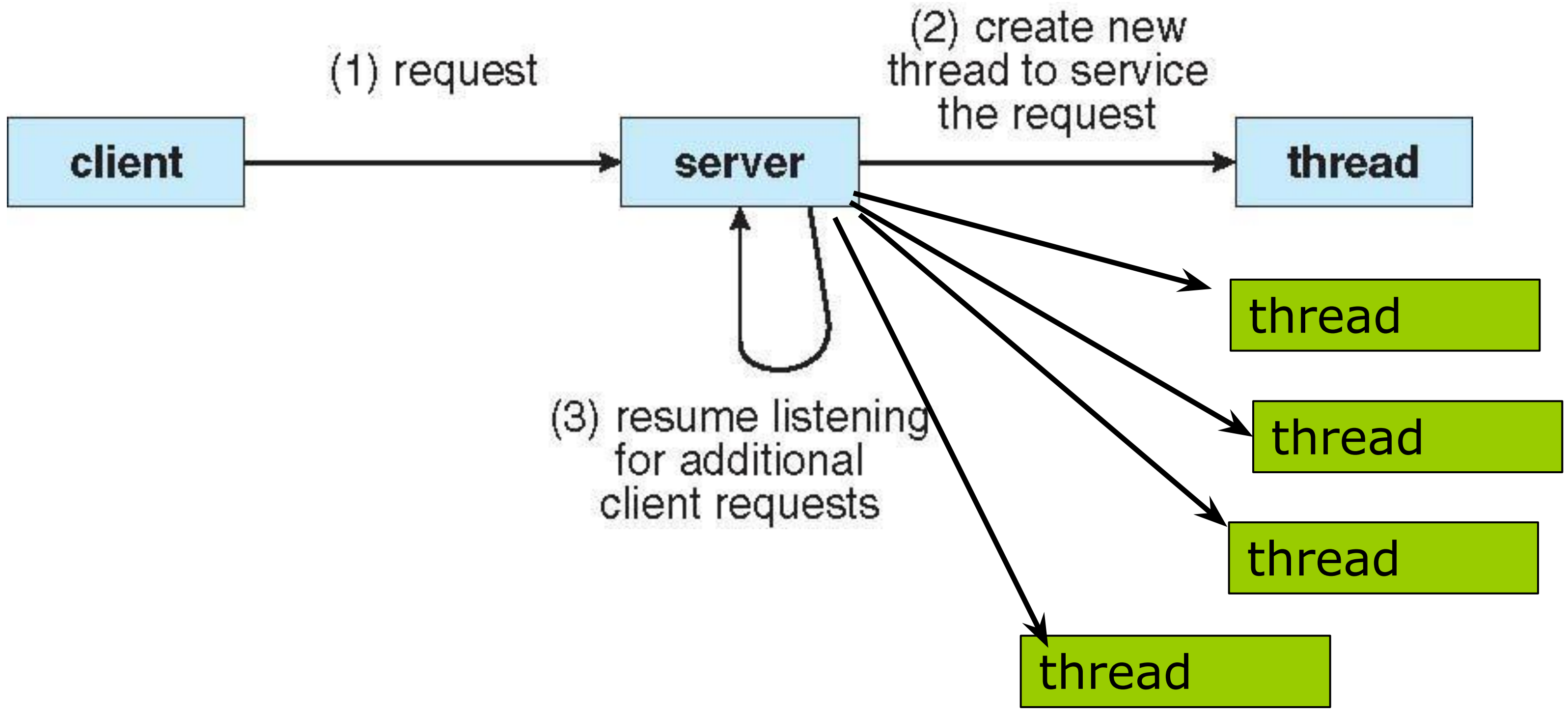
Threads specific Attributes….

- Thread ID
- Thread specific data
- CPU affinity
- Stack (local variables and function call linkage information)
- ……

Benefits

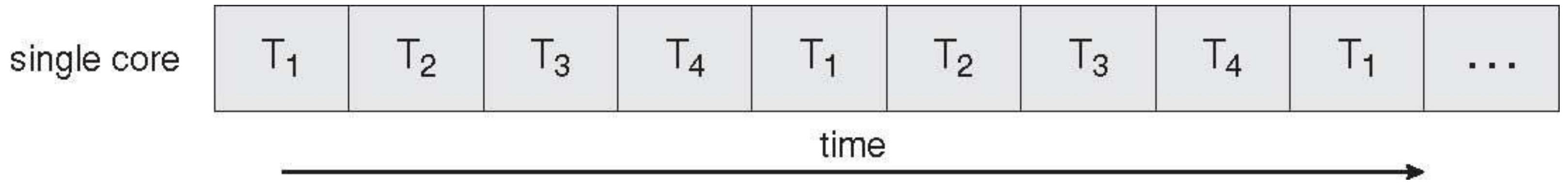
- **Responsiveness**
  Interactive application can delegate background functions to a thread and keep running
- **Resource Sharing**
  Several different threads can access the same address space
- **Economy**
  Allocating memory and new processes is costly. Threads are much ‘cheaper’ to initiate.
- **Scalability**
  Use threads to take advantage of multiprocessor architecture

Multithreaded Server Architecture

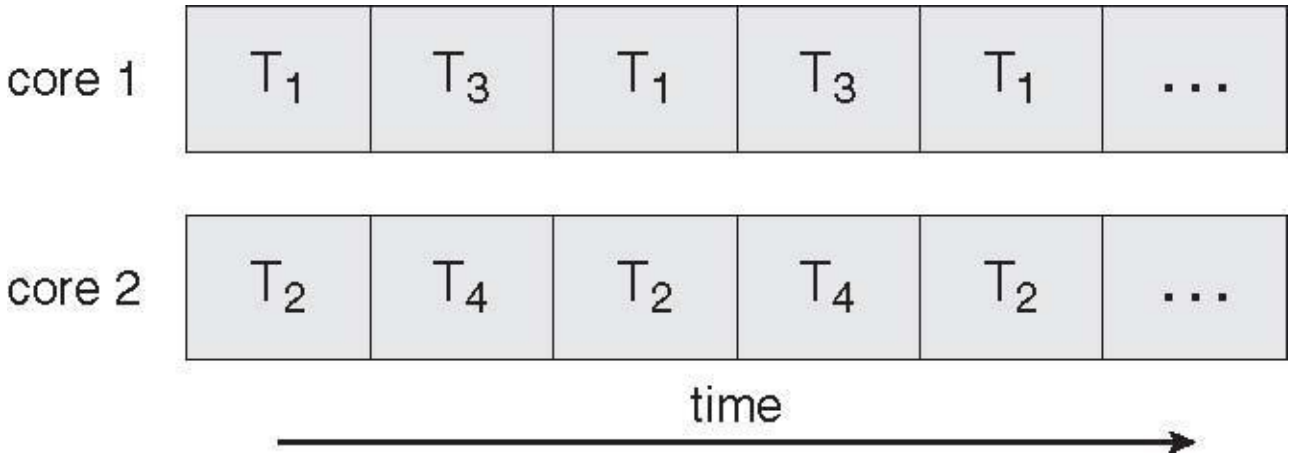

(1) request
(2) create new thread to service the request
(3) resume listening for additional client requests
client
server
thread
thread
thread
thread
thread

Multicore Programming

Concurrent Execution on a Single-core System



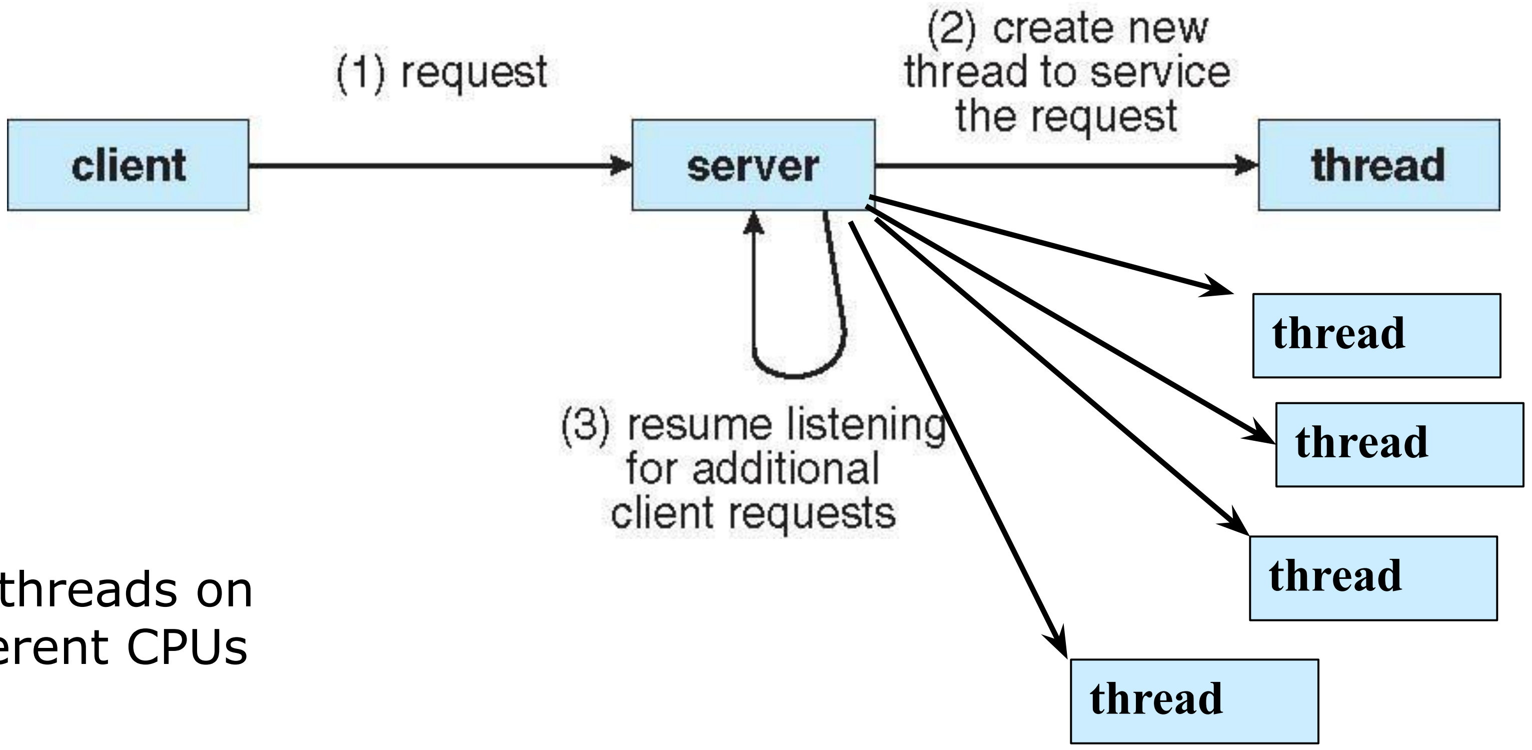
Parallel Execution on a Multicore System



Multicore Programming

- Multicore systems putting pressure on programmers, challenges include
  - **Dividing activities**
    - What tasks can be separated to run on different processors
  - **Balance**
    - Balance work on all processors
  - **Data splitting**
    - Separate data to run with the tasks
  - **Data dependency**
    - Watch for dependences between tasks
  - **Testing and debugging**
    - Harder!!!!

Threads Assist Multicore Programming


(1) request
(2) create new thread to service the request
client
server
thread
thread
thread
thread
thread
(3) resume listening for additional client requests

Let threads on different CPUs

Types of Parallelism

- **Data parallelism** focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core.
- **Example**: Consider, for example, summing the contents of an array of size *N*.
- **Task parallelism** involves distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a unique operation.
- Different threads may be operating on the same data, or they may be operating on different data.
- **Example:** Two threads, each performing a unique statistical operation on the array of elements

Multithreading Models

- Support provided at either
  - User level -> **user threads**

  Supported above the kernel and managed without kernel support

  - Kernel level -> **kernel threads**

  Supported and managed directly by the operating system

  What is the relationship between user and kernel threads?

User Threads

- Thread management done by user-level threads library
- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads

Kernel Threads

- Supported by the Kernel
- Examples
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

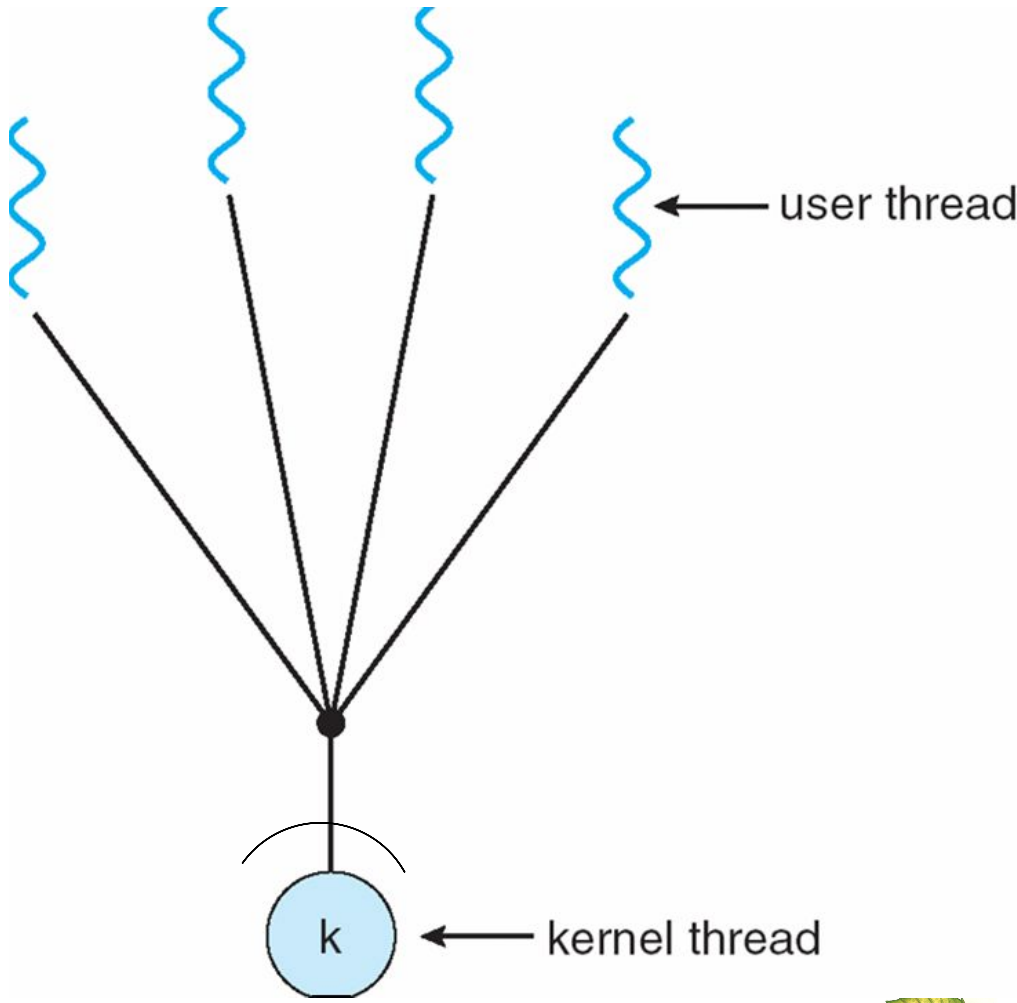
Multithreading Models

User Thread – to - Kernel Thread

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

Many-to-One

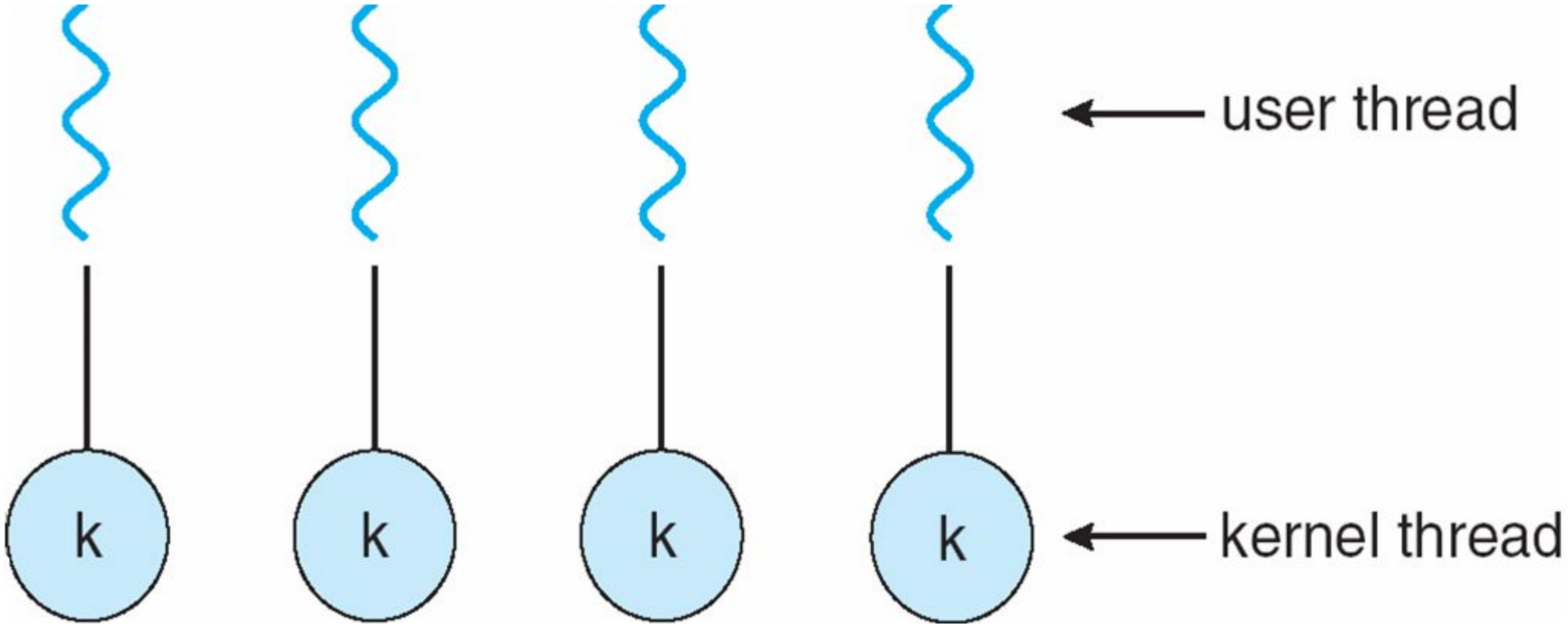
Many user-level threads mapped to single kernel thread



One-to-One

Each user-level thread maps to kernel thread

- Examples
  - Windows NT/XP/2000
  - Linux

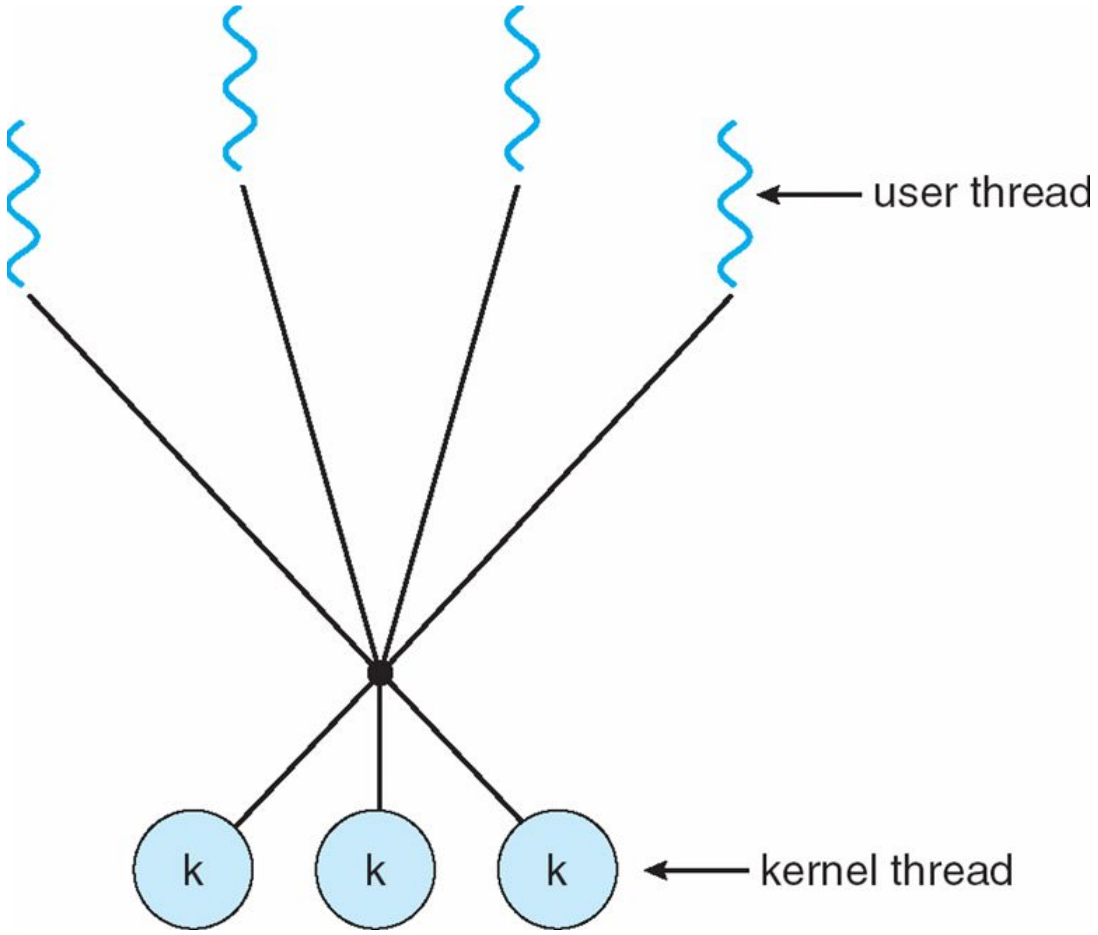


Many-to-Many Model

Allows many user level threads to be mapped to many kernel threads

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

- Example
  - Windows NT/2000 with the *ThreadFiber* package



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 main thread libraries in use today:
  - POSIX Pthreads
  - Win32
  - Java

Thread Libraries

- Three main thread libraries in use today:
  - **POSIX Pthreads**
    - May be provided either as user-level or kernel-level
    - A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
    - API specifies behavior of the thread library, implementation is up to development of the library
  - **Win32**
    - Kernel-level library on Windows system
  - **Java**
    - Java threads are managed by the JVM
    - Typically implemented using the threads model provided by underlying OS

POSIX Compilation on Linux

On Linux, programs that use the Pthreads API must be compiled with ***–pthread*** or ***–lpthread***

POSIX: Thread Creation

```
#include <pthread.h>

pthread_create (thread, attr, start_routine, arg)
```

returns : 0 on success, some error code on failure.

POSIX: Thread ID

```
#include <pthread.h>

pthread_t pthread_self()
```

returns : ID of current (this) thread

POSIX: Wait for Thread Completion

```
#include <pthread.h>


pthread_join (thread, NULL)
```

- **returns :** 0 on success, some error code on failure.

POSIX: Thread Termination

```
#include <pthread.h>

Void pthread_exit (return_value)
```

Threads terminate in one of the following ways:

- The thread's start functions performs a return specifying a return value for the thread.
- Thread receives a request asking it to terminate using pthread_cancel()
- Thread initiates termination pthread_exit()
- Main process terminates

Thread Cancellation

Terminating a thread before it has finished

Having made the cancellation request, *pthread_cancel()* returns immediately; that is it does not wait for the target thread to terminate.

So, what happens to the target thread?

What happens and when it happens depends on the thread's cancellation state and type.

Thread Cancellation

Thread attributes that indicate cancellation state and type

- Two general states
  - Thread can not be cancelled.
    - PTHREAD_CANCEL_DISABLE
  - Thread can be cancelled
    - PTHREAD_CANCEL_ENABLE
    - Default

Thread Cancellation

- When thread is cancelable, there are two general <u>types</u>
  - **Asynchronous cancellation** terminates the target thread immediately
    - 4 PTHREAD_CANCEL_ASYNCHRONOUS
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
    - 4 PTHREAD_CANCEL_DEFERRED
    - 4 Cancel when thread reaches ‘cancellation point’

Thread Termination: Cleanup Handlers

Functions automatically executed when the thread is canceled.

- Clean up global variables
- Unlock and code or data held by thread
- Close files
- Commit or rollback transactions

Add and remove cleanup handlers based on thread logic

pthread_cleanup_push()

pthread_cleanup_pop()

Implicit Threading

- Thread Pool: To control number of threads in system.
- OpenMP: OpenMP is a set of compiler directives that provides support for parallel programming in shared-memory environments.
- 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.

Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Signal handling
- Thread cancellation
- Scheduler Activation
- Thread-specific data/ thread local storage

Semantics of *fork()* ,*exec()*, *exit()*

Does **fork()** duplicate only the calling thread or all threads?

- Threads and ***exec()***

  With *exec(),* the calling program is replaced in memory. All threads, except the once calling exec(), vanish immediately. No thread-specific data destructors or cleanup handlers are executed.

- Threads and ***exit()***

  If any thread calls *exit()* or the main thread does a return, ALL threads immediately vanish. No thread-specific data destructors or cleanup handlers are executed.

Semantics of *fork()* ,*exec()*, *exit()*

- Threads and ***fork()***

  When a multithread process calls *fork()*, only the calling thread is replicated. All other threads vanish in the child. No thread-specific data destructors or cleanup handlers are executed.

  Problems:

  - The global data may be inconsistent:
    - Was another thread in the process of updating it?
    - Data or critical code may be locked by another thread. That lock is copied into child process, too.
  - Memory leaks
  - Thread (other) specific data not available

Recommendation: In multithreaded application, only use *fork()* after *exec()*

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
- 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 threa to receive all signals for the process

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

Thread Safety

A function is *thread-safe* if it can be safely invoked by multiple threads at the same time.

Example of a not safe function:

```
static int glob = 0;


static void Incr (int  loops)
{
   int loc, j;
   for (j = 0; j<loops; j++ {
          loc = glob;
          loc++;
          glob = loc;
   }
}
```

Employs global or static values that are shared by all threads

Thread Safety

Ways to render a function thread safe

- Serialize the function
    - Lock the code to keep other threads out
    - Only one thread can be in the sensitive code at a time
- Lock only the critical sections of code
    - Only let one thread update the variable at a time.
- Use only thread safe system functions
- Make function reentrant
    - Avoid the use of global and static variables
    - Any information required to be safe is stored in buffers allocated by the call

Thread Specific Data

- Makes existing functions thread-safe .
  - May be slightly less efficient than being reentrant
- Allows each thread to have its own copy of data
  - Provides per-thread storage for a function
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

POSIX compliation

On Linux, programs that use the Pthreads API must be compiled with ***–pthread.*** The effects of this option include the following:

- _REENTRANT preprocessor macro is defined. This causes the declaration of a few reentrant functions to be exposed.

- The program is linked with the *libpthread* library

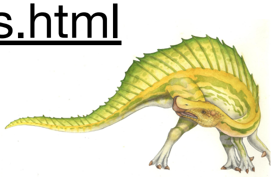
  (the equivalent of ***-lpthread***)

  - The precise options for compiling a multithreaded program vary across implementations (and compilers).

From M. Kerrisk, The Linux Programming Interface

http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html

http://codebase.eu/tutorial/posix-threads-c/



Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Scheduler activations provide upcalls - a communication mechanism from the kernel to the thread library
- This communication allows an application to maintain the correct number kernel threads

Threads vs. Processes

- Advantages of multithreading
  - Sharing between threads is easy
  - Faster creation
- Disadvantages of multithreading
  - Ensure threads-safety
  - Bug in one thread can bleed to other threads, since they share the same address space
  - Threads must compete for memory
- Considerations
  - Dealing with signals in threads is tricky
  - All threads must run the same program
  - Sharing of files, users, etc

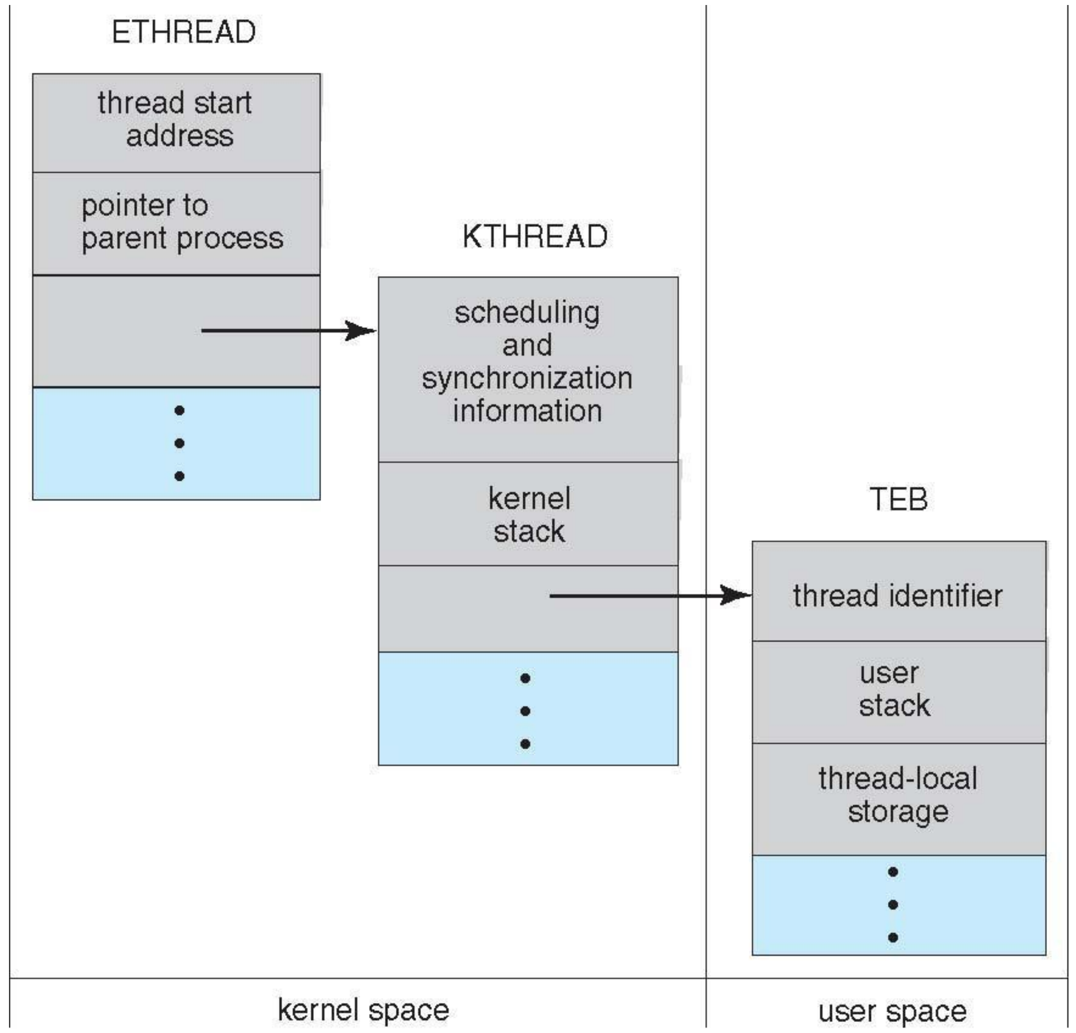
Operating System Examples

- Windows XP Threads
- Linux Thread

Windows XP Threads

- Implements the one-to-one mapping, kernel-level
- Each thread contains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area
- The register set, stacks, and private storage area are known as the context of the threads
- The primary data structures of a thread include:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)

Windows XP Threads



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)

Linux Threads

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.

Chapter 4: Threads

- Overview
- Multithreading Models
- Thread Libraries
- Threading Issues
- Operating System Examples
- Windows XP Threads
- Linux Threads

End of Chapter 4