Chapter 4: Threads

Chapter 4: Threads

- Overview
- Multithreading Models
- Threading Issues
- Pthreads
- Windows XP Threads
- Linux Threads
- Java Threads

Threads

- A thread is the basic unit of CPU utilization
- A thread is a flow of control within a process.
- A thread comprises
  - A thread ID
  - Program counter
  - Register set
  - Stack
- Shares with the other threads belonging to the same process
  - Code section
  - Data section
  - Other operating systems resources: files and signals

Single and Multithreaded Processes

code data files

registers stack

thread

single-threaded process

code data files

registers registers registers

stack stack stack

thread

multithreaded process

Benefits

- **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.
  - For instance, a multithreaded web browser could still allow user interaction in one thread while an image was being loaded in another thread.
- **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.

Benefits

- **Economy**
  - Allocating memory and resources for process creation is costly. Because threads share resources of the process to which they belong, it is more economical to create and context-switch threads.
  - Empirically finding the difference in overhead can be difficult, but in general it is much more time consuming to create and manage processes than threads.
  - In Solaris, for example, creating a process is about **thirty** times slower than is creating a thread, and context switching is about **five** times slower.

Benefits

- **Utilization of multiprocessor architectures.**
  - The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be runniing in parallel on different processors.
  - A single threaded process can only run on one CPU, no matter how many are available.
  - Multithreading on a multi-CPU machine increases concurrency.

User and Kernel Threads

- Support for threads may be provided either at the user level, for user threads, or by the kernel, for kernel threads.
  - User threads are supported above the kernel and are managed without kernel support,
  - Kernel threads are supported and managed directly by the operating system.
- Virtually all contemporary operating systems-including Windows XP, Linux, Mac OS x, Solaris, and Tru64 UNIX (formerly Digital UNIX)-support kernel threads.
- Ultimately, there must exist a relationship between user threads 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

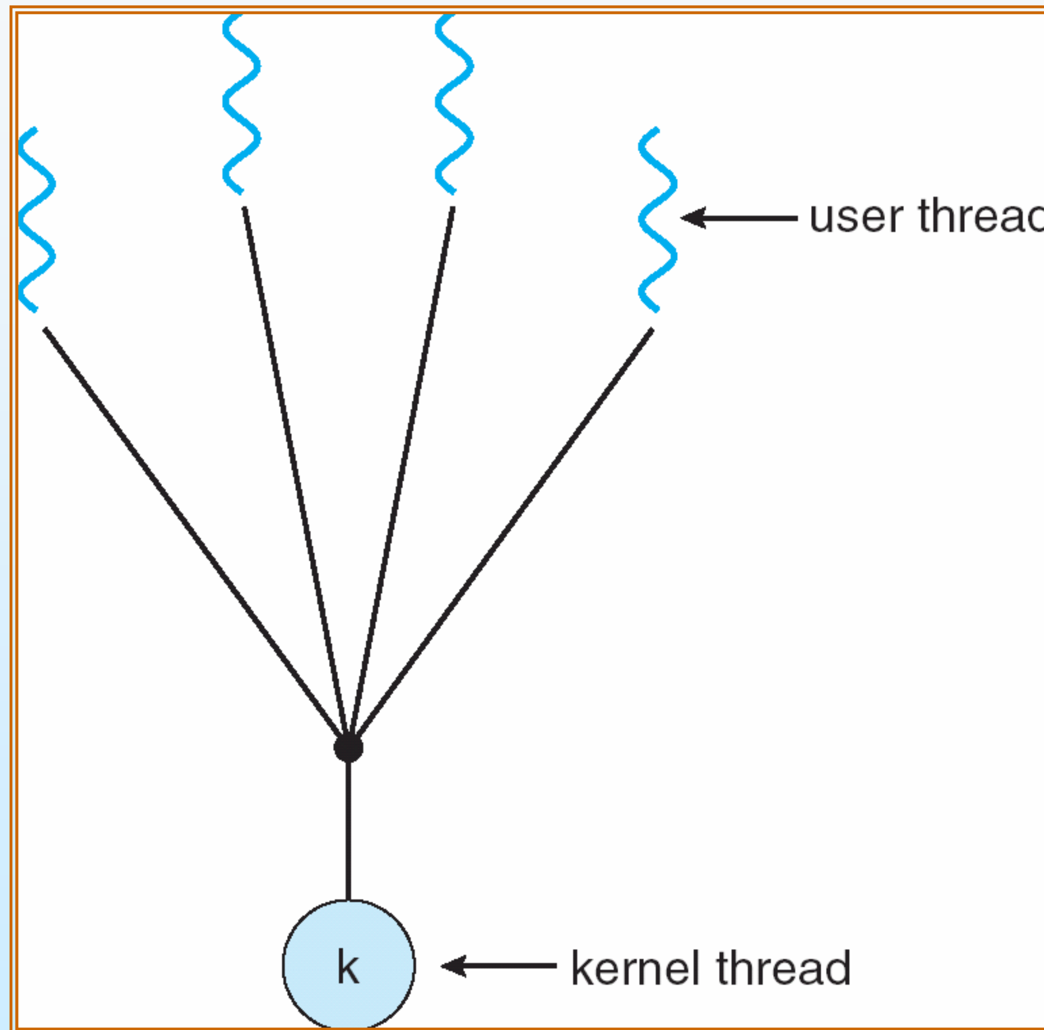
Multithreading Models

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

Many-to-One

- Many user-level threads mapped to single kernel thread
  - It maps many user-level threads to one kernel thread.
  - Thread management is done by the thread library in user space, so it is efficient; but the entire process will block if a thread makes a blocking system call.
  - Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors.
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads

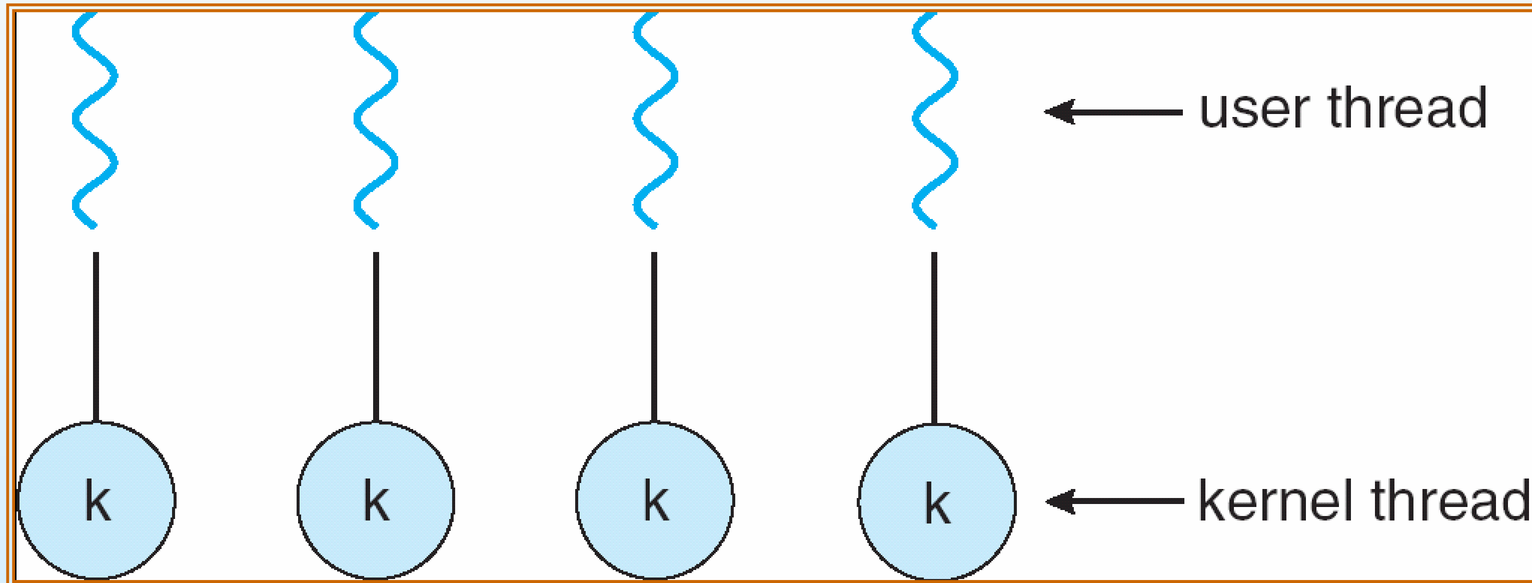
Many-to-One Model



One-to-One

- Each user-level thread maps to kernel thread
  - It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call;
  - It also allows multiple threads to run in parallel on multiprocessors.
  - **The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread**. Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system.
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

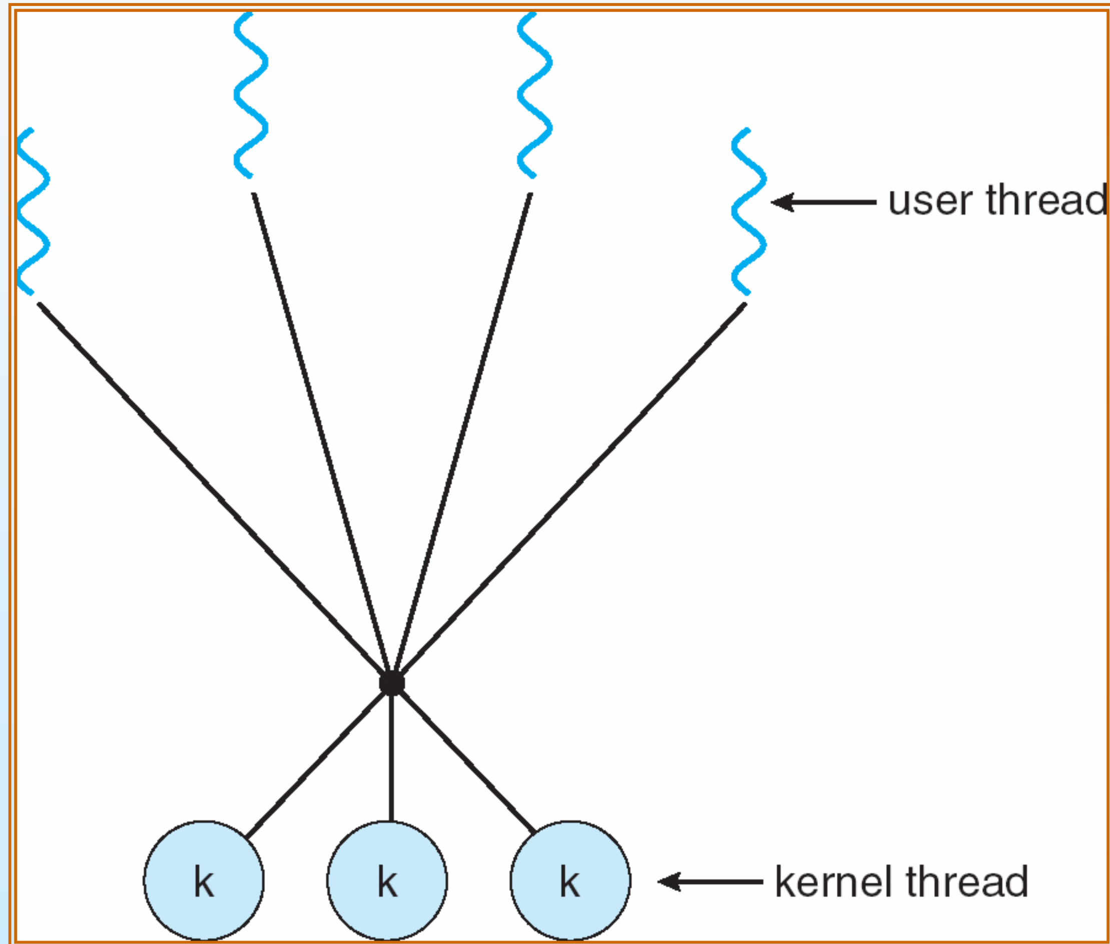
One-to-one Model



Many-to-Many Model

- Multiplexes many user-level threads to
  - a smaller or equal number of kernel threads.
  - The number of kernel threads may be specific to either a particular application or a particular machine application may be allocated more kernel threads on a multiprocessor than on a uniprocessor).
- Allows the operating system to create a sufficient number of kernel threads
- Whereas the many-to-one model allows the developer to create as many user threads as she wishes, true concurrency is not gained because the kernel can schedule only one thread at a time.
- The one-to-one model allows for greater concurrency, but the developer has to be careful not to create too many threads within an application
- The many-to-many model suffers from neither of these shortcomings:
  - Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.
  - Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package

Many-to-Many Model



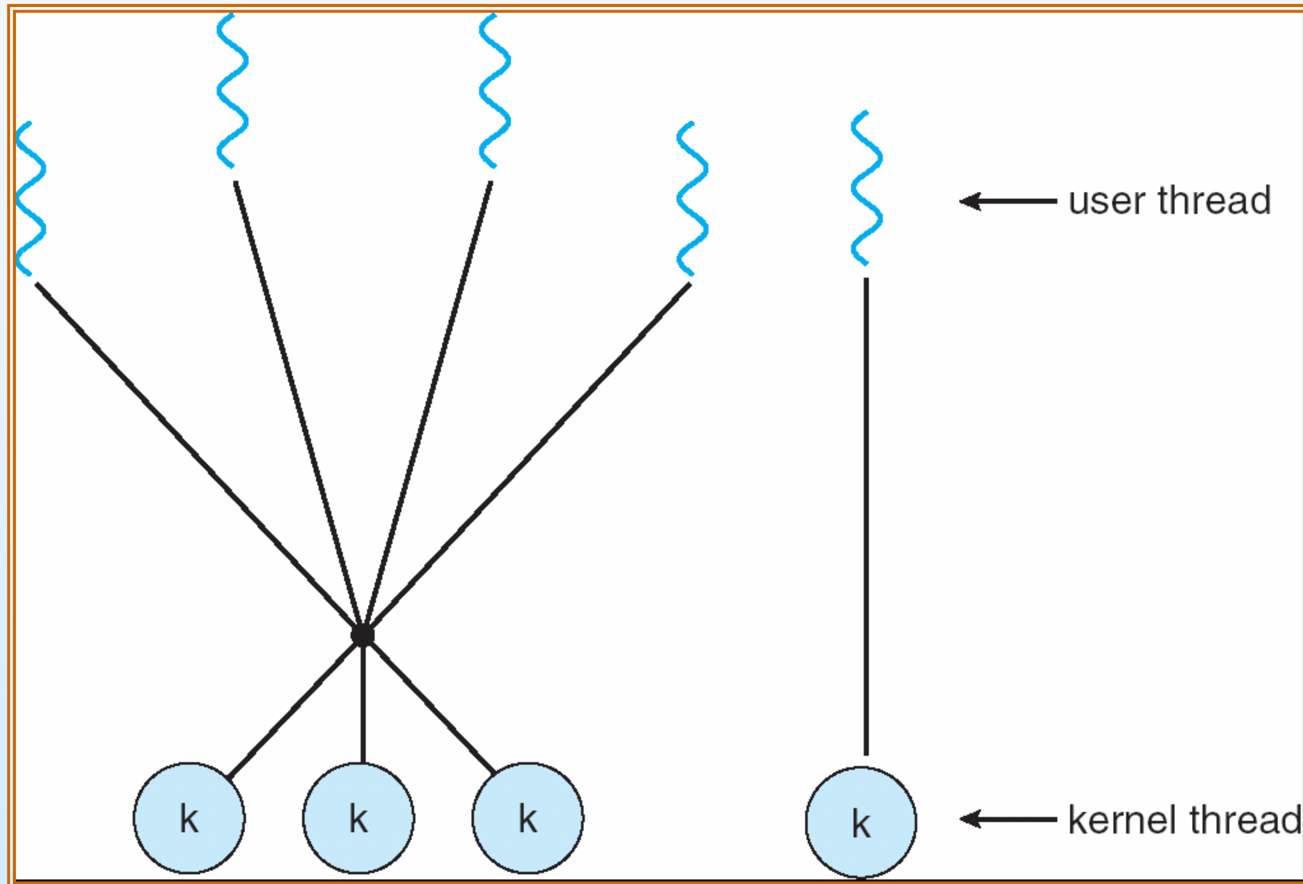
Two-level Model

- One popular variation on the 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.
- This variation, sometimes referred to as the ***two-level*** *model*
  - IRIX, HP-UX, and Tru64 UNIX.
- 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.

Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier

Two-level Model



Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Thread cancellation
- Signal handling
- Thread pools
- Thread specific data
- Scheduler activations

Semantics of fork() and exec()

- The semantics of the fork() and exec() system calls change in a multithreaded program.
- If one thread in a program calls fork(), does the new process duplicate all threads, or is the new process single-threaded?
- Some UNIX systems have chosen to have two versions of fork (), one that duplicates all threads and another that duplicates only the thread that invoked the fork() system call.
- The exec () system call typically works in the same way as described.
  - That is, if a thread invokes the exec() system call, the program specified in the parameter to exec() will replace the entire process-including all threads.

Thread Cancellation

- Terminating a thread before it has finished
  - For example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled.
  - Another situation might occur when a user presses a button on a web browser that stops a web page from loading any further. Often, a web page is loaded using several threads-each image is loaded in a separate thread. When a user presses the *stop* button on the browser, all threads loading the page are canceled.

Thread Cancellation

- A thread that is to be canceled is often referred to as the target thread.
- Cancellation of a target thread may occur in two different scenarios:
  - 1. Asynchronous cancellation. One thread immediately terminates the target thread.
  - 2. Deferred cancellation. The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

Thread Cancellation

- The difficulty with cancellation occurs in situations where resources have been allocated to a canceled thread or where a thread is canceled while in the midst of updating data it is sharing with other threads.
  - This becomes especially troublesome with asynchronous cancellation.
  - Often, the operating system will reclaim system resources from a canceled thread but will not reclaim all resources. Therefore, canceling a thread asynchronously may not free a necessary system-wide resource.
- With deferred cancellation, in contrast, one thread indicates that a target thread is to be canceled, but cancellation occurs only after the target thread has checked a flag to determine if it should be canceled or not. This allows a thread to check whether it should be canceled at a point when it can be canceled safely. Pthreads refers to such points as cancellation points.

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 of and the reason for the event being signaled. All signals, whether synchronous or asynchronous, follow the same pattern.
- 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

Signals

- Examples of synchronous signals include illegal memory access and division by O.
  - If a running program performs either of these actions, a signal is generated.
  - Synchronous Signals are delivered to the same process that performed the operation that caused the signal (that is the reason they are considered synchronous).
- When a signal is generated by an event external to a running process, that process receives the signal asynchronously.
  - Examples of such signals include terminating a process with specific keystrokes (such as <control> <*C*» and having a timer expire.
  - Typically, an asynchronous signal is sent to another process.

Signal Handlers

- Every signal may be *handled* by one of two possible handlers:
  - A default signal handler
  - A user-defined signal handler
- Every signal has a default signal handler that is run by the kernel when handling that signal.
  - This default action can be overridden by a user-defined signal handler that is called to handle the signal.
- Signals may be handled in different ways.
  - Some signals (such as changing the size of a window) may simply be ignored; others (such as an illegal memory access) may be handled by terminating the program.

Signal Handlers

- Handling signals in single-threaded programs is straightforward; signals are always delivered to a process.
- However, delivering signals is more complicated in multithreaded programs, where a process may have several threads.
- Where, then, should a signal be delivered? In general, the following options exist:

  1. Deliver the signal to the thread to which the signal applies.
  2. Deliver the signal to every thread in the process.
  3. Deliver the signal to certain threads in the process.
  4. Assign a specific thread to receive all signals for the process.

UNIX signals

- Most multithreaded versions of UNIX allow a thread to specify which signals it will accept and which it will block.
- Therefore, in some cases, an asynchronous signal may be delivered only to those threads that are not blocking it.
- However, because signals need to be handled only once, a signal is typically delivered only to the first thread found that is not blocking it.
- The standard UNIX function for delivering a signal is kill (aid_t aid, int signal); here, we specify the process (aid) to which a particular signal is to be delivered.
- However, POSIX Pthreads also provides the pthread_kill (pthread_t tid, int signal) function, which allows a signal to be delivered to a specified thread (tid.)

Windows

- Although Windows does not explicitly provide support for signals, they can be emulated using **asynchronous procedure calls** (APCs).
- The APC facility allows a user thread to specify a function that is to be called when the user thread receives notification of a particular event.
- As indicated by its name, an APC is roughly equivalent to an asynchronous signal in UNIX.
- However whereas UNIX must contend with how to deal with signals in a multithreaded environment, the APC facility is more straightforward, as an APC is delivered to a particular thread rather than a process.

Thread Pool 1

- In a multithreading web server.
  - whenever the server receives a request, it creates a separate thread to service the request.
- Whereas creating a separate thread is certainly superior to creating a separate process, a multithreaded server nonetheless has potential problems.
  - The first concerns the **amount of time required to create** the thread prior to servicing the request, together with the fact that this thread will be discarded once it has completed its work.
  - The second issue is more troublesome: If we allow all concurrent requests to be serviced in a new thread, we have **not placed a bound on the number of threads** concurrently active in the system.
  - Unlimited threads could exhaust system resources, such as CPU time or memory.
- One solution to this issue is to use a **thread pool.**

Thread Pools 2

- 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
- The number of threads in the pool can be set heuristically based on factors such as the number of CPUs in the system, the amount of physical memory, and the expected number of concurrent client requests.
- More sophisticated thread-pool architectures can dynamically adjust the number of threads in the pool according to usage patterns.
- Such architectures provide the further benefit of having a smaller pool - thereby consuming less memory - when the load on the system is low.

Thread Specific Data

- Threads belonging to a process share the data of the process.
- Indeed, this sharing of data provides one of the benefits of multithreaded programming.
- However, in some circumstances, each thread might need its own copy of certain data.
- We will call such data thread-specific data.
- For example, in a transaction-processing system, we might service each transaction in a separate thread.
- Furthermore, each transaction may be assigned a unique identifier. To associate each thread with its unique identifier we could use thread specific data.
- Most thread libraries, including Win32 and Pthreads, provide some form of support for thread-specific data. Java provides support as well.

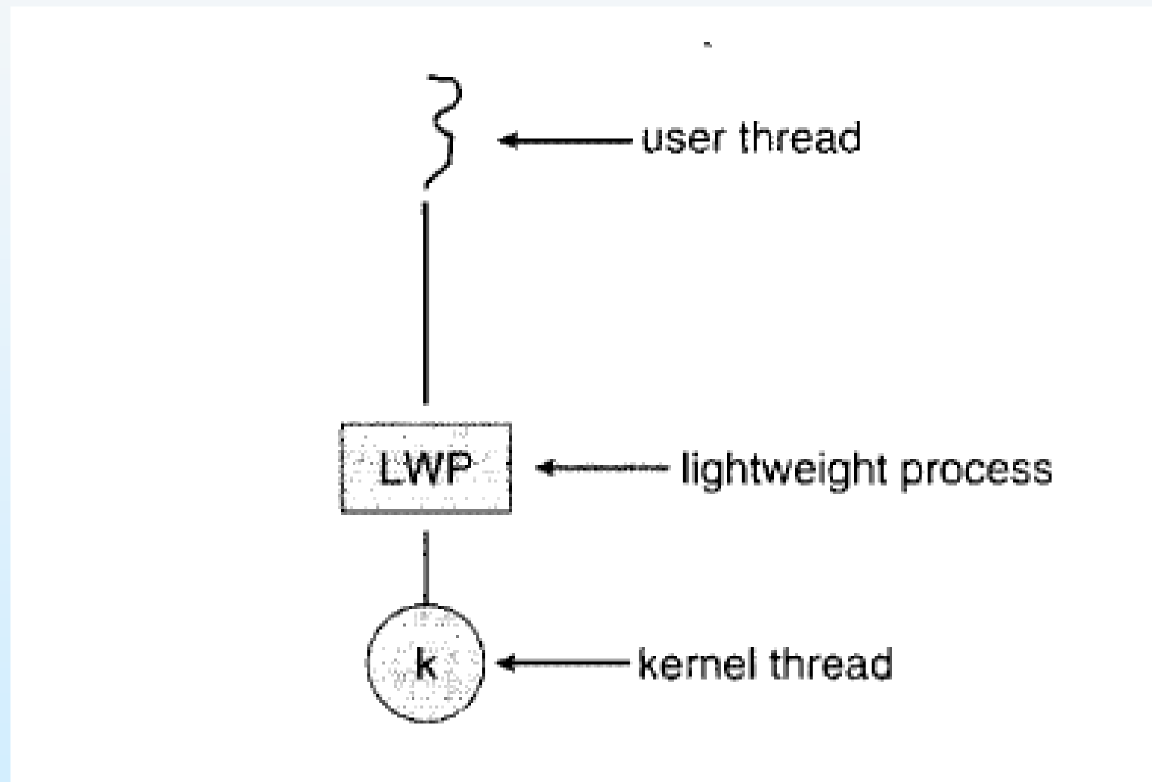
Scheduler Activations

- A final issue to be considered with multithreaded programs concerns communication between the kernel and the thread library which may be required by the many-to-many and two-level models.
  - **Such coordination allows the number of kernel threads to be dynamically adjusted to help ensure the best performance.**
- Many systems implementing either the many-to-many or two-level model place an intermediate data structure between the user and kernel threads.
- This data structure is typically known as **lightweight process - 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.

Scheduler Activations

- 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), and the application can schedule user threads onto an available virtual processor.
  - Furthermore, 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.
- The communication allows an application to maintain the correct number of kernel threads

Lightweight Process



Threaded Programs

- Several common models for threaded programs exist:
  - ***Manager/worker:*** a single thread, the ***manager*** assigns work to other threads, the ***workers***. Typically, the manager handles all input and parcels out work to the other tasks. At least two forms of the manager/worker model are common: **static worker pool** and **dynamic worker pool**.
  - ***Pipeline:*** a task is broken into a series of suboperations, each of which is handled in series, but concurrently, by a different thread. An automobile assembly line best describes this model.
  - ***Peer:*** similar to the manager/worker model, but after the main thread creates other threads, it participates in the work.

Pthreads

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

Threads Performace

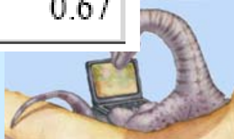
• The primary motivation for using Pthreads is to realize potential program performance gains.

• When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead.

• **Managing threads requires fewer system resources than managing processes.**

• For example, the following table compares timing results for the **fork()** subroutine and the **pthreads_create()** subroutine. Timings reflect 50,000 process/thread creations, were performed with the time utility, and units are in seconds, no optimization flags.

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
AMD 2.4 GHz Opteron (8cpus/node)	41.07	60.08	9.01	0.66	0.19	0.43
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	64.24	30.78	27.68	1.75	0.69	1.10
IBM 1.5 GHz POWER4 (8cpus/node)	104.05	48.64	47.21	2.01	1.00	1.52
INTEL 2.4 GHz Xeon (2 cpus/node)	54.95	1.54	20.78	1.64	0.67	0.90
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.54	1.07	22.22	2.03	1.26	0.67



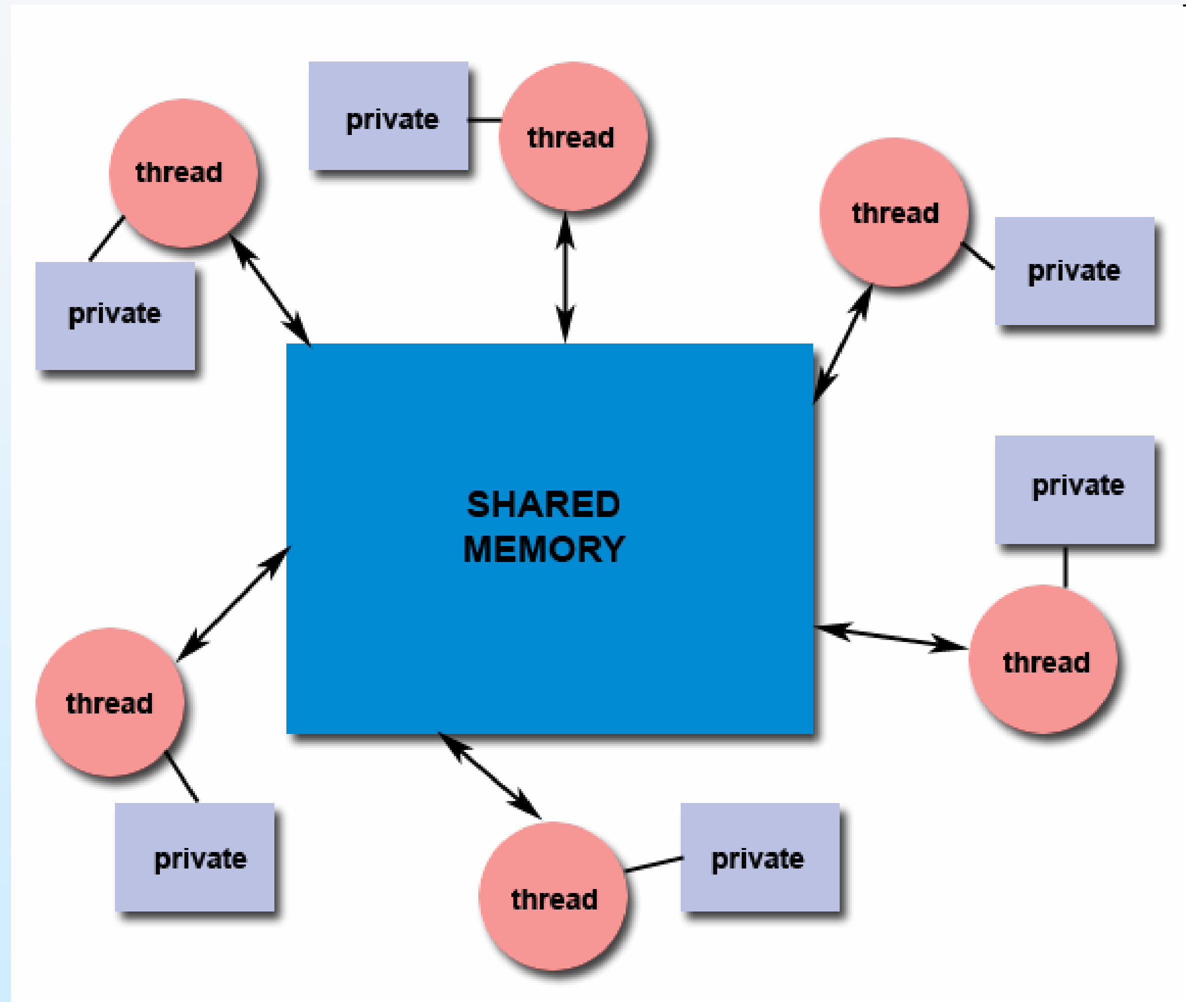
Threads: mutual exclusion

- Mutex is an abbreviation for "mutual exclusion". Mutex variables are one of the primary means of implementing thread synchronization and for protecting shared data when multiple writes occur.
- A mutex variable acts like a "lock" protecting access to a shared data resource. The basic concept of a mutex as used in Pthreads is that only one thread can lock (or own) a mutex variable at any given time. Thus, even if several threads try to lock a mutex only one thread will be successful.
- No other thread can own that mutex until the owning thread unlocks that mutex.
- Threads must "take turns" accessing protected data.
- Mutexes can be used to prevent "race" conditions. An example of a race condition involving a **bank transaction** is shown below:

Thread 1	Thread 2	Balance
Read balance: $1000		$1000
	Read balance: $1000	$1000
	Deposit $200	$1000
Deposit $200		$1000
Update balance $1000+$200		$1200
	Update balance $1000+$200	$1200

Shared Memory Model

- All threads have access to the same global, shared memory
- Threads also have their own private data
- Programmers are responsible for synchronizing access (protecting) globally shared data.



Thread safeness

Thread-safeness: in a nutshell, refers an application's ability to execute multiple threads simultaneously without creating "race" conditions.

Main Program

Thread 1
call subA

Thread 2
call subA

Thread 3
call subA

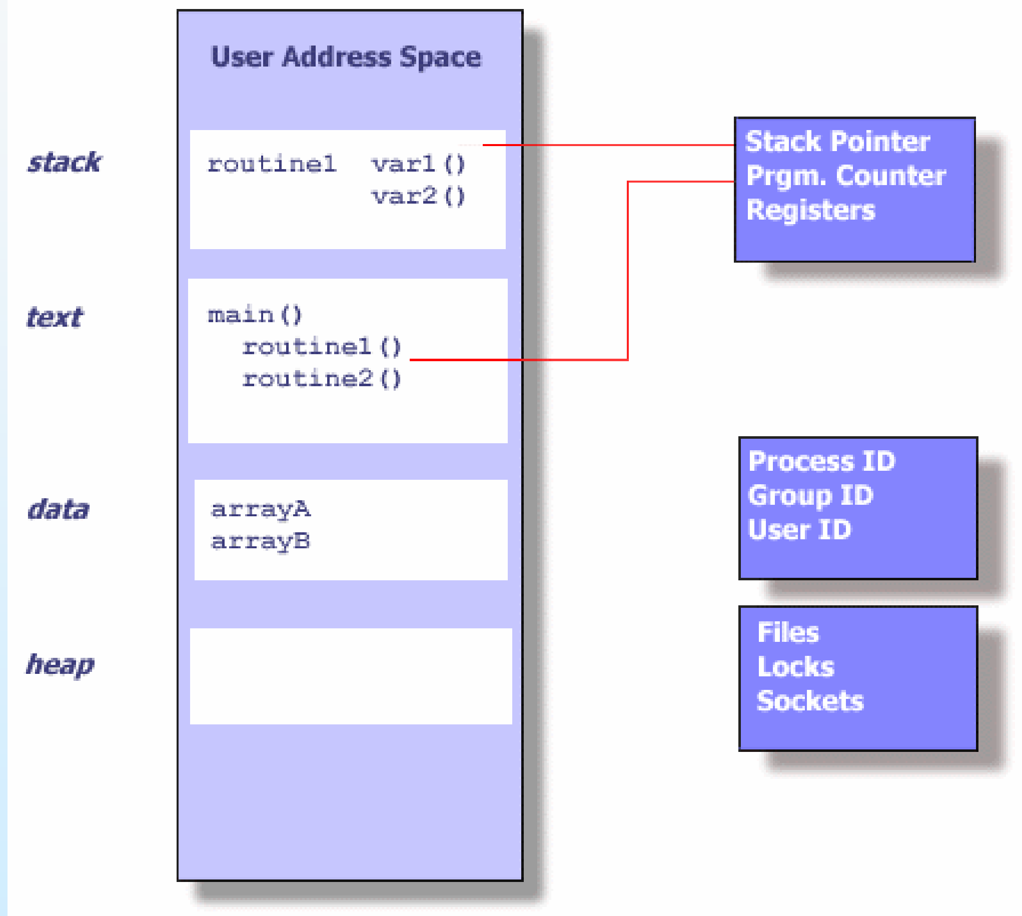
subA
....
....
modify(memloc 0x4450A)

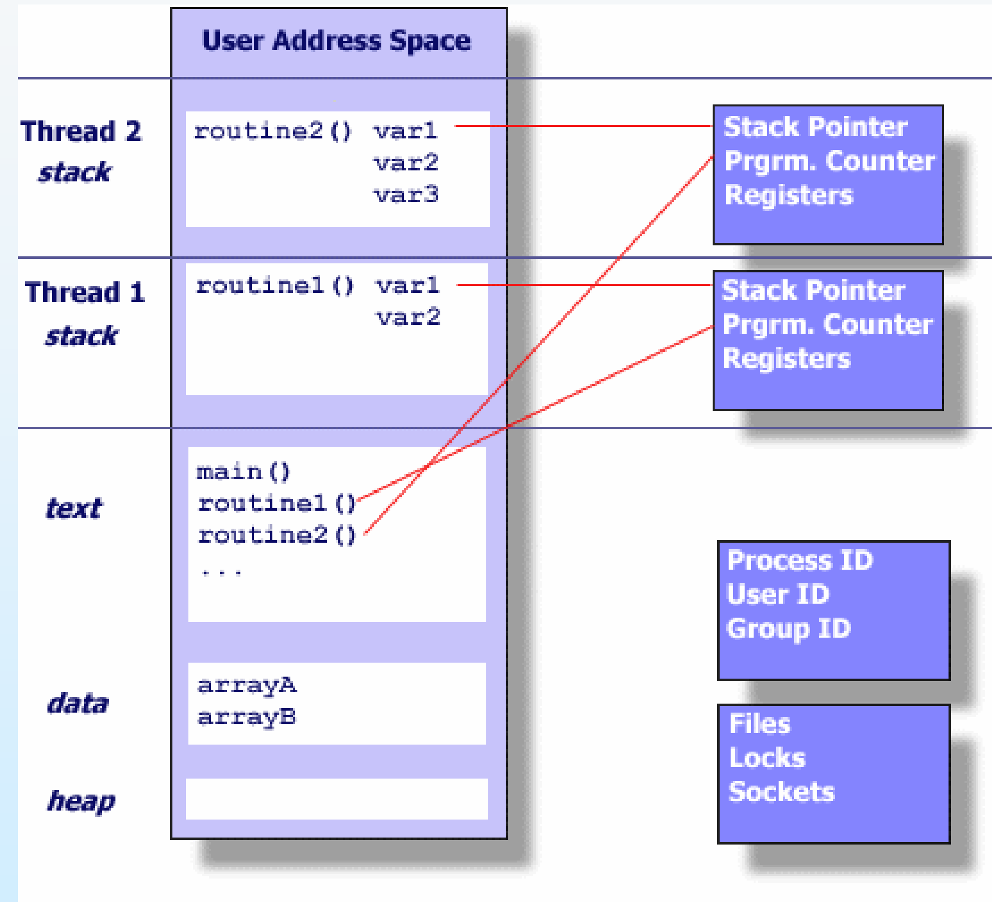
modify(memloc 0x4450A)

modify(memloc 0x4450A)

Global Memory
memloc 0x00000
...
...
memloc 0x4450A
...
...

UNIX threads

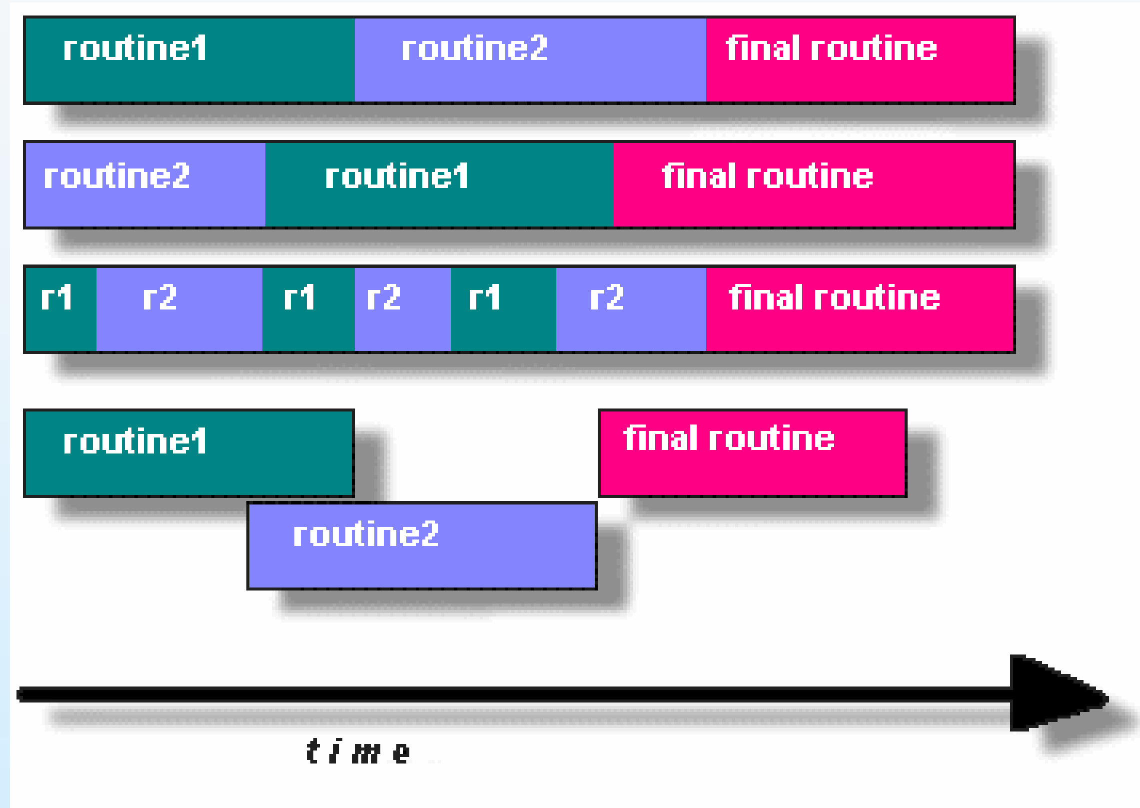




UNIX PROCESS

THREADS WITHIN A UNIX PROCESS

Advantage of threads



Windows XP Threads

- Implements the one-to-one mapping
- 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

ETHREAD

thread start address

pointer to parent process

KTHREAD

scheduling and synchronization information

kernel stack

TEB

thread identifier

user stack

thread-local storage

kernel space

user space

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)

Java Threads

- Java threads are managed by the JVM
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface

Java Thread States

new

new

start()

runnable

sleep()
I/O

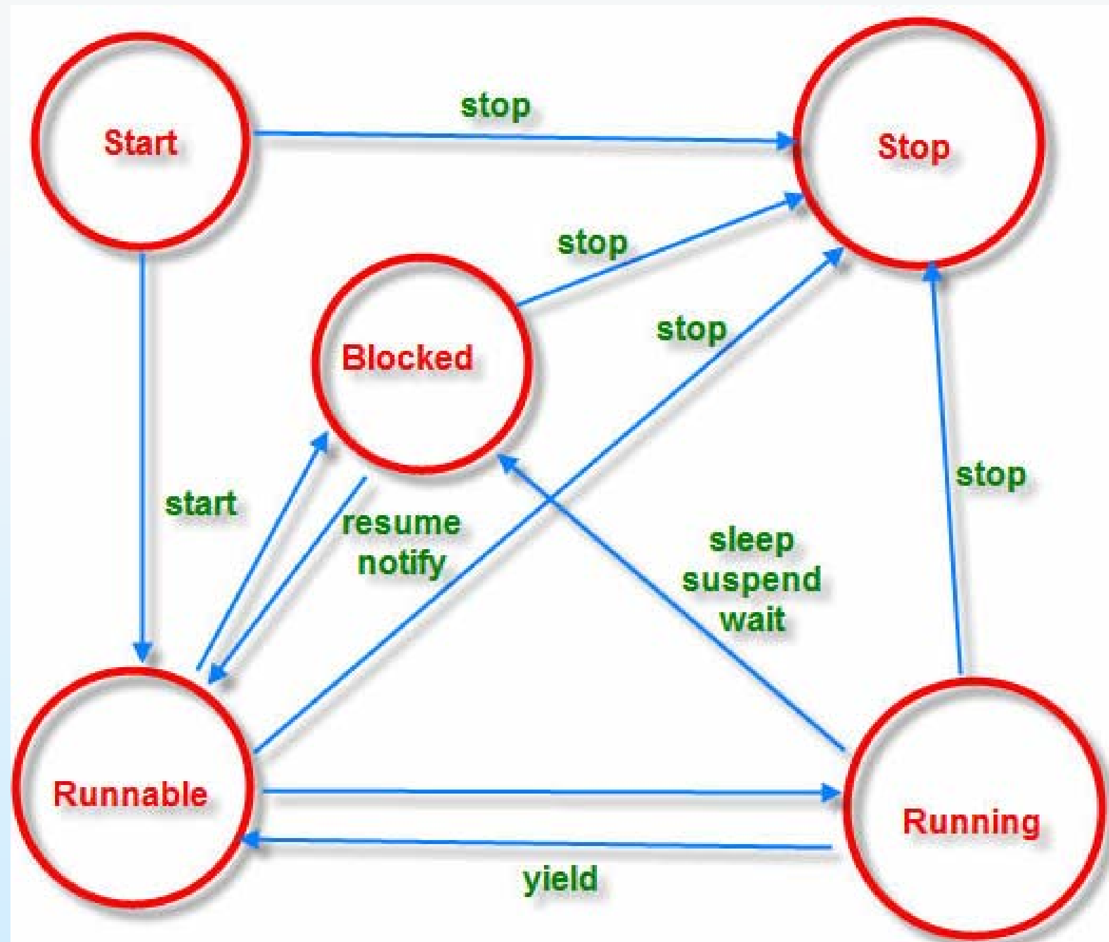
exits
run()
method

I/O
is
available

dead

blocked

Java Thread Life



Readings

- Readings
  - Silberschatz, Chapter 4

End of Chapter 4

Just for Fun ☺