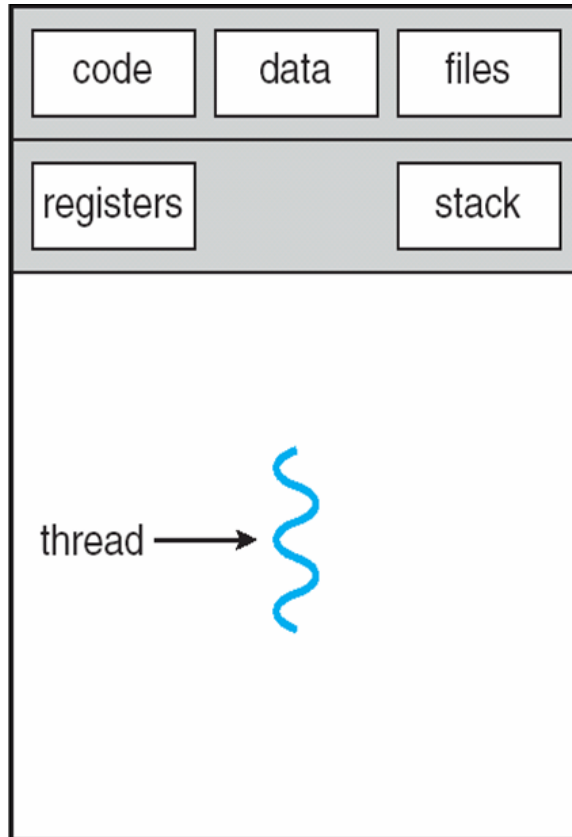


COP 4610 — Chapter 4

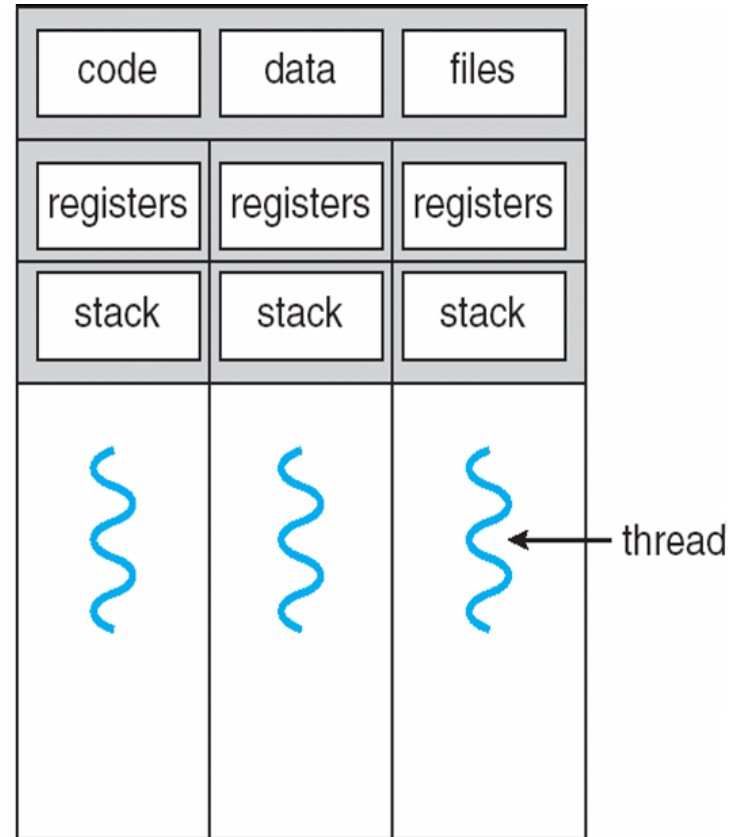
Threads

Ming Zhao, PhD

Single and Multithreaded Processes



single-threaded process

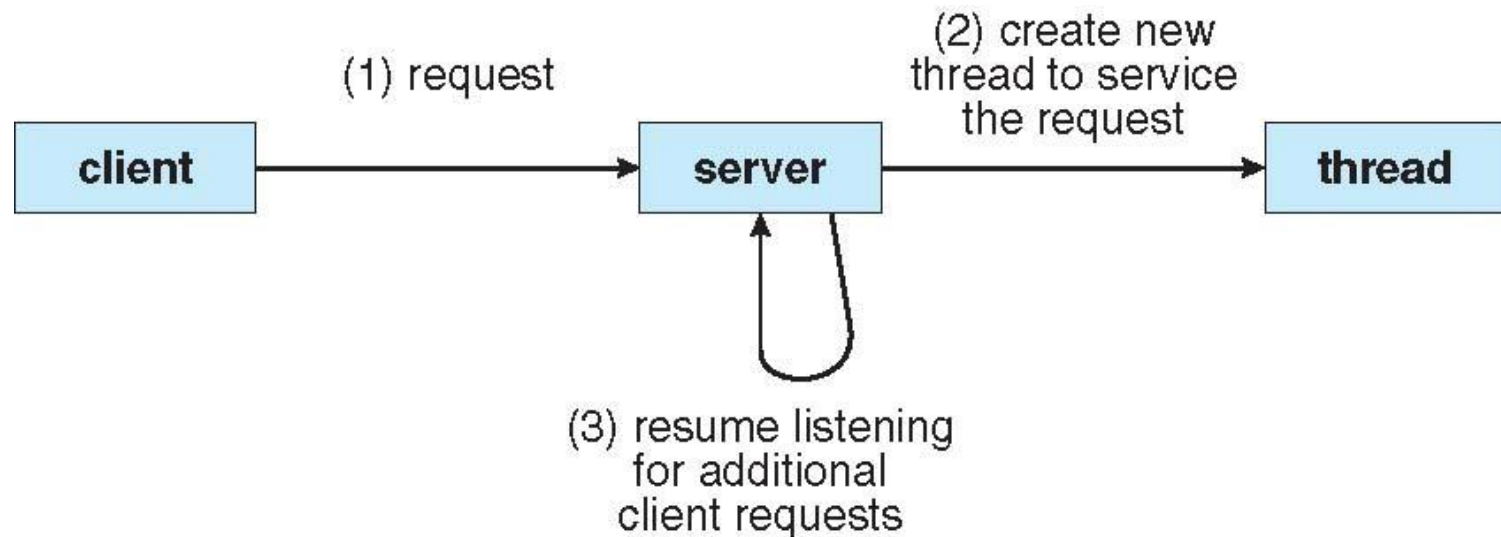


multithreaded process

Benefits

- Responsiveness
 - Allow a program to run even if part of it is blocked
- Resource sharing
 - Threads share memory and resources of the process to which they belong by default
- Economy
 - More economical to create and context-switch threads
- Scalability
 - Multithreading on multiprocessor increases parallelism

Multithreaded Server Architecture



User-perceived Threads

- Threads perceived by users
 - Created and managed through user-level threads library
- Three primary thread libraries:
 - POSIX Pthreads
 - Win32 threads
 - Java threads

Kernel-supported Threads

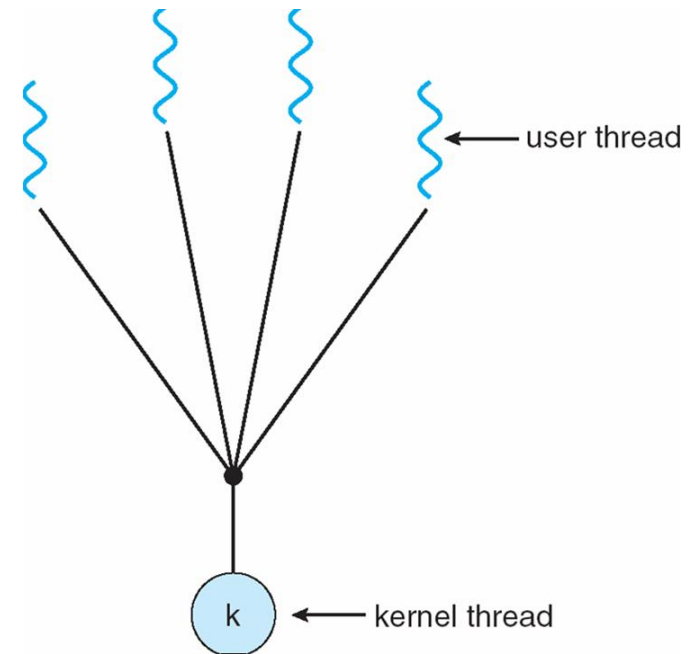
- Threads supported by kernel
 - Executed and managed by kernel
- Examples
 - Windows XP/2000
 - Solaris
 - Linux
 - Mac OS X

Multithreading Models

- The relationship between *user-perceived* threads and *kernel-supported* threads
 - Many-to-One
 - One-to-One
 - Many-to-Many

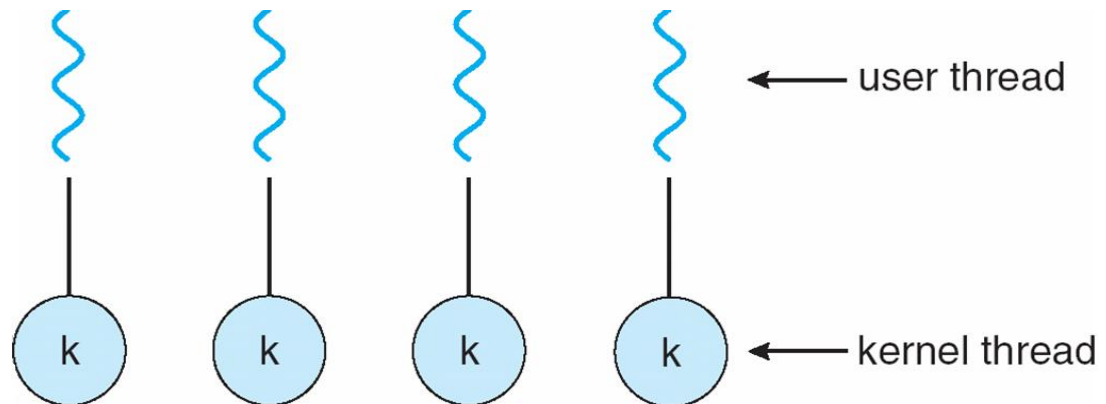
Many-to-One

- Many user-perceived threads mapped to single kernel-supported thread
 - Efficient: thread management done by user-space thread library
 - The entire process will block if any thread makes a blocking system call
 - Unable to run in parallel on multiprocessors
- Examples:
 - Solaris Green Threads
 - GNU Portable Threads



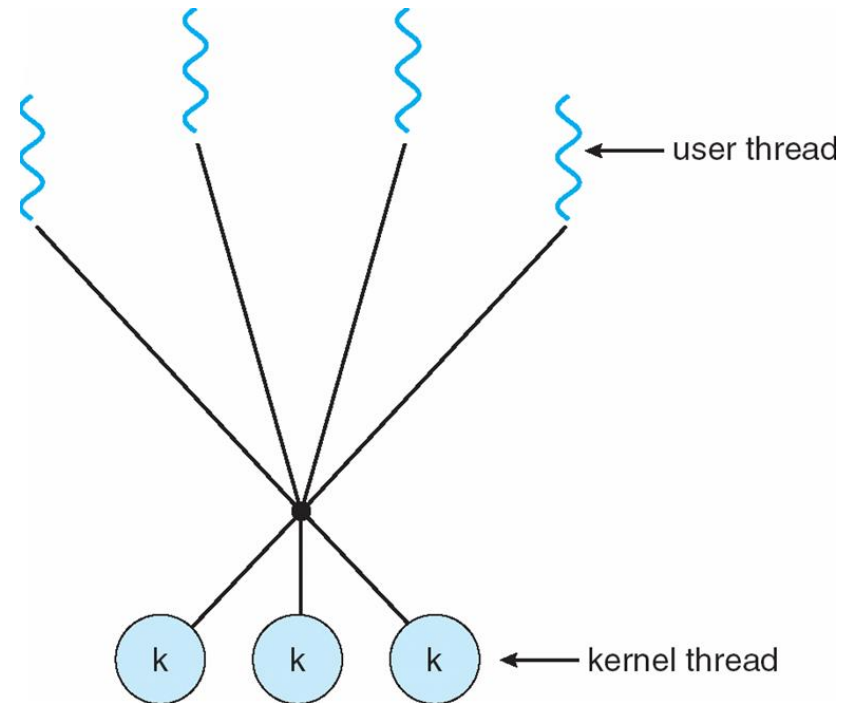
One-to-One

- Each user-perceived thread maps to one kernel-supported thread
 - Allow more concurrency
 - More overhead of creating kernel threads
- Examples
 - Windows NT/XP/2000
 - Linux
 - Solaris 9 and later



Many-to-Many Model

- Allows many user-perceived threads to be mapped to many kernel-supported threads
- Examples
 - Solaris prior to version 9
 - 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

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

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:
 - Extending Thread class
 - Implementing the Runnable interface

Thread Pools

- Create a number of threads in a pool where they await work
 - The multithreaded Web server example
- 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

Implicit Threading

- Shift the burden of thread creation and management from developers to compilers and run-time libraries
 - With the continued growth of multicore processing, programming with many threads is not trivial
- Grand Central Dispatch (GCD) on Mac OS X/iOS
 - GCD identifies **blocks** – self-contained units of work

```
^ { printf("I am a block"); }
```
 - GCD schedules blocks using dispatch queues
 - A scheduled block is assigned a thread from a thread pool
 - Serial queue – blocks executed one by one sequentially
 - Concurrent queue – multiple blocks executed concurrently

Semantics of `fork()` and `exec()`

- Does *fork()* duplicate only the calling thread or all threads?
 - Some UNIX systems have two versions of *fork()*:
 - One duplicate all threads
 - The other duplicates only the thread that invokes *fork()*
 - Application decides which version to use
 - Which one to use if *exec()* is called immediately after forking?
- Does *exec()* replace only the calling thread or all threads?
 - Typically replace the entire process, including all threads

Semantics of `exit()`

- Does *exit()* terminate only the calling thread or all threads?
 - Typically terminate all threads before they complete
- How does a thread terminate itself?
 - E.g., *pthread_exit()*
- Will the termination of the *main* thread also terminate the other threads?

Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred
 - Synchronous signal generated by the process's own operation
 - Illegal memory access, division by zero
 - Asynchronous signal generated by event external to the process
 - Keystrokes, timer
- 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
- Two possible handler implementation
 - Default one run by kernel
 - User-defined one can override the default one

SIGHUP	1	Exit	Hangup
SIGINT	2	Exit	Interrupt
SIGQUIT	3	Core	Quit
SIGILL	4	Core	Illegal Instruction
SIGTRAP	5	Core	Trace/Breakpoint Trap
SIGABRT	6	Core	Abort
SIGEMT	7	Core	Emulation Trap
SIGFPE	8	Core	Arithmetic Exception
SIGKILL	9	Exit	Killed
SIGBUS	10	Core	Bus Error
SIGSEGV	11	Core	Segmentation Fault
SIGSYS	12	Core	Bad System Call
SIGPIPE	13	Exit	Broken Pipe
SIGALRM	14	Exit	Alarm Clock
SIGTERM	15	Exit	Terminated
SIGUSR1	16	Exit	User Signal 1
SIGUSR2	17	Exit	User Signal 2
SIGCHLD	18	Ignore	Child Status
SIGPWR	19	Ignore	Power Fail/Restart
SIGWINCH	20	Ignore	Window Size Change
SIGURG	21	Ignore	Urgent Socket Condition
SIGPOLL	22	Ignore	Socket I/O Possible
SIGSTOP	23	Stop	Stopped (signal)
SIGTSTP	24	Stop	Stopped (user)
SIGCONT	25	Ignore	Continued
SIGTTIN	26	Stop	Stopped (tty input)
SIGTTOU	27	Stop	Stopped (tty output)
SIGVTALRM	28	Exit	Virtual Timer Expired
SIGPROF	29	Exit	Profiling Timer Expired
SIGXCPU	30	Core	CPU time limit exceeded
SIGXFSZ	31	Core	File size limit exceeded
SIGWAITING	32	Ignore	All LWPs blocked
SIGLWP	33	Ignore	Virtual Interprocessor Interrupt for Threads Library
SIGAIO	34	Ignore	Asynchronous I/O

Signal Handling

- Options for multithreaded programs
 - 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
- Synchronous signals?
- Asynchronous signals?
- UNIX allows a thread to specify which signals it will accept and which it will block

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 resources of its parent task

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.

Linux Kernel Threads

- Standard processes exist solely in kernel-space
 - Do not context switch to user-space
 - Schedulable and preemptable
- Useful for kernel to perform background operations
 - Flush dirty data out of memory cache
 - Handle soft interrupts
 - ...
 - Kill zombies (your Lab 2!)

Kernel vs. User Threads: Clarification

- Kernel threads
 - Threads run solely in kernel space
 - The kernel must support threading
- User threads (supported by kernel)
 - Threads run user space but scheduled by kernel
 - The kernel must support threading
- User threads (not supported by kernel)
 - Threads run and scheduled in user space
 - Supported solely by user-space threading library

Exercises

- 4.2, 4.8, 4.15, 4.16