Threads

Eike Ritter[1]

Modified: October 16, 2012

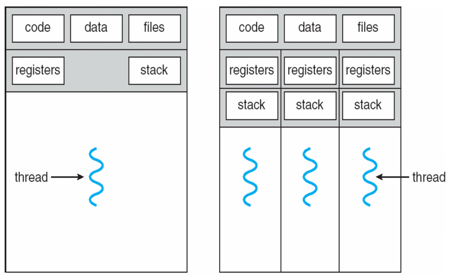
Lecture 8: Operating Systems with C/C++
School of Computer Science, University of Birmingham, UK

[1]Based on material by Matt Smart and Nick Blundell

Outline

Single and Multithreaded Processes

- A thread is an execution state of a process (*e.g.* the next instruction to execute, the values of CPU registers, the stack to hold local variables, *etc.*)
  - Thread state is separate from global process state, such as the code, open files, global variables (on the heap), *etc.*

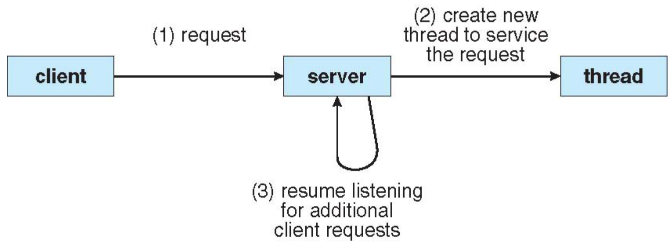


single-threaded process multithreaded process

Benefits of Threads

- **Responsiveness** - user interaction in a GUI can be responded to by a separate thread from that, say, doing long running computation (*e.g.* saving a file, running some algorithm, *etc.*)
- **Resource Sharing** - Threads within a certain process share its address space and can therefore use shared variables to communicate, which is more efficient than passing messages.
- **Economy** - Threads are often referred to as light-weight processes, since running a system with multiple threads has a smaller memory footprint than the equivalent with multiple processes.
- **Scalability** - For multi-threaded processes it is much easier to make use of parallel processing (*e.g.* multi-core processors, and distributed systems)
- **Reduce programming complexity** - Since problems can be broken down into parallel tasks, rather than more complex state machines.

Multithreaded Server Architecture

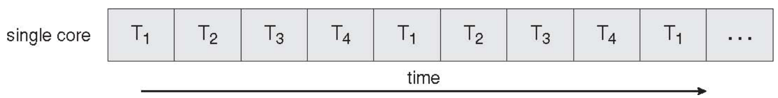


Multicore Programming

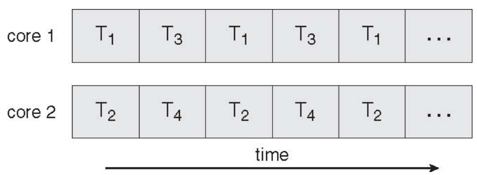
- Multicore systems are putting pressure on programmers, with challenges that include:
  - Dividing activities - How can we make better use of parallel processing?
  - Balance - How should we balance the parallel tasks on the available cores to get maximum efficiency?
  - Data splitting - How can data sets be split for processing in parallel and then rejoined (*e.g.* SETI@home)
  - Data dependency - Some processing must be performed in a certain order, so synchronisation of tasks will be necessary.
  - How to test and debug such systems?

Concurrent and Parallel Execution

Single-core Concurrent Thread Execution



Multicore Parallel Thread Execution



User Threads

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

Kernel Threads

- Threading is supported by modern OS Kernels
- Examples:
  - Windows XP/2000
  - Solaris
  - Linux
  - Mac OS X

Threading Models

- A particular kernel (*e.g.* on an embedded device, or an older operating system) may not support multi-threaded processes, though it is still possible to implement threading in the user process.
- Therefore many threading models exist for mapping user threads to kernel threads:
  - Many-to-One
  - One-to-One
  - Many-to-Many

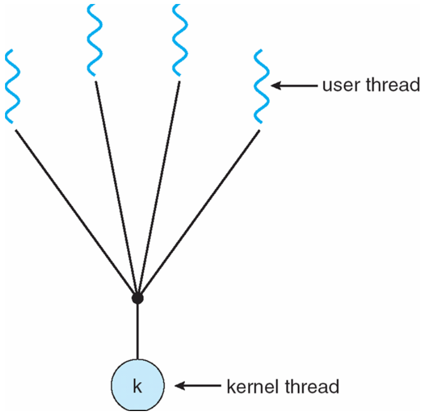
Many-to-One

- Many user-level threads mapped to single kernel thread/process
- Useful if the kernel does not support threads
- But what if one user thread calls a blocking kernel function?
  - This will block the whole process (*i.e.* all the other user threads)
  - Complex solutions exist where the user-mode thread package intercepts blocking calls, changes them to non-blocking and then implements a user-mode blocking mechanism.

Many-to-One

- You could implement something like this yourself, by having a process respond to timer events that cause it to perform a context switch in user space (*e.g.* store current registers, CPU flags, instruction pointer, then load previously stored ones)
  - Since most high-level languages cannot manipulate registers directly, you would have to write a small amount of assembler code to make the switch.
- Examples:
  - Solaris Green Threads: http://bit.ly/qYnKAQ
  - GNU Portable Threads: http://www.gnu.org/software/pth/

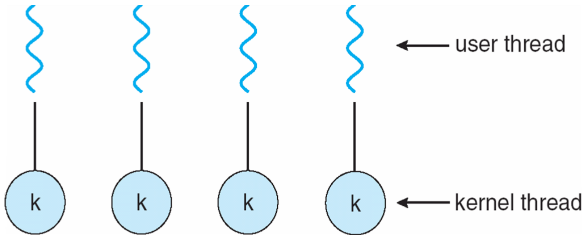
Many-to-One



One-to-One

- Each user-level thread maps to kernel thread
- But, to switch between threads a context switch is required by the kernel.
- Also, the number of kernel threads may be limited by the OS
- Examples:
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

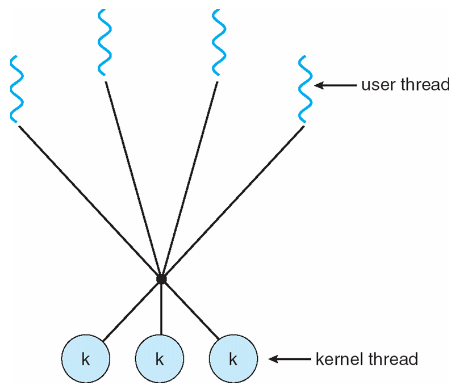
One-to-One



Many-to-Many

- Allows many user level threads to be mapped to many kernel threads
  - Best of both worlds
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the ThreadFiber package

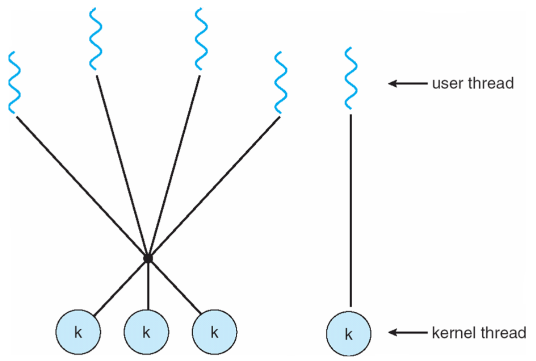
Many-to-Many



Two-Level Model

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

Two-Level Model



Unclear Semantics of UNIX `fork()` system call

- Does `fork()` duplicate only the calling thread or all threads?
- Sometimes we want this, and sometimes we don’t, so some UNIX systems provide alternative fork functions.

Thread Cancellation

- How to terminate a thread before it has finished?
- Two general approaches use by programmers:
  - **Asynchronous cancellation** terminates the target thread immediately
    - Useful as a last resort if a thread will not stop (*e.g.* due to a bug, *etc.*)
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
    - This approach is often considered to be much cleaner, since the thread can perform any clean-up processing (*e.g.* close files, update some state, *etc.*)

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
  - Signal is generated by particular event
  - Signal is delivered to a process
  - Signal is handled by some function
- Not so straightforward for a multi-threaded process. Options are:
  - 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
- In most UNIX systems a thread can be configured to receive or block (*i.e.* not handle) certain signals to help overcome these issues.

Thread Pools

- Under a high request-load, multithreaded servers can waste a lot processing time simply creating and destroying threads.
- Solution:
  - Pre-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, to ensure some level of service for a finite number of clients.

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

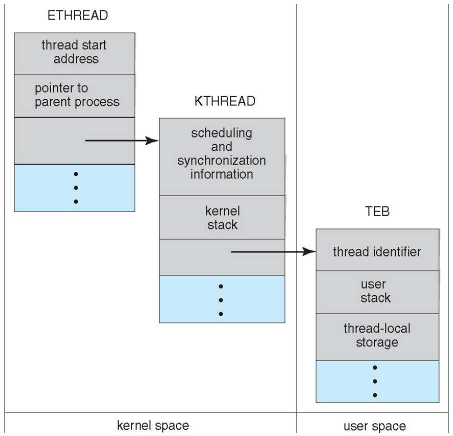
Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behaviour of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)
- Example: `threadtest.c`. Note, this is an implementation of *POSIX* Pthreads, so compiles differently!

Windows XP Threads

- Implements the one-to-one mapping (*i.e.* kernel-level threads)
- 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) - Stores general info about a thread: its parent process, address of the instruction where the thread starts execution.
  - `KTHREAD` (kernel thread block) - Stores kernel-level state of the thread: kernel stack, *etc.*
  - `TEB` (thread environment block) - Stores user-level state of the thread: user stack, thread-local storage.

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) and can be passed flags to control exactly what resources are shared.

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.

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

Summary

We looked at:

1. Multithreaded Processes
   - Overview
   - Concurrent and Parallel Execution
   - User and Kernel Threads
2. Threading Models
   - Threading Models
   - Many-to-One
   - One-to-One
   - One-to-One
   - Many-to-Many
   - Two-Level Model
3. Some Considerations of Threads
   - Semantics of `fork()`
   - Thread Cancellation
   - Signal Handling
   - Thread Pools
4. Threading Implementation
   - Thread Libraries
   - Pthreads
   - Windows XP Threads
   - Linux Threads
   - Java Threads

Next time:
Process synchronisation