Department of Electrical Engineering, Feng-Chia University

Chapter 5: Threads

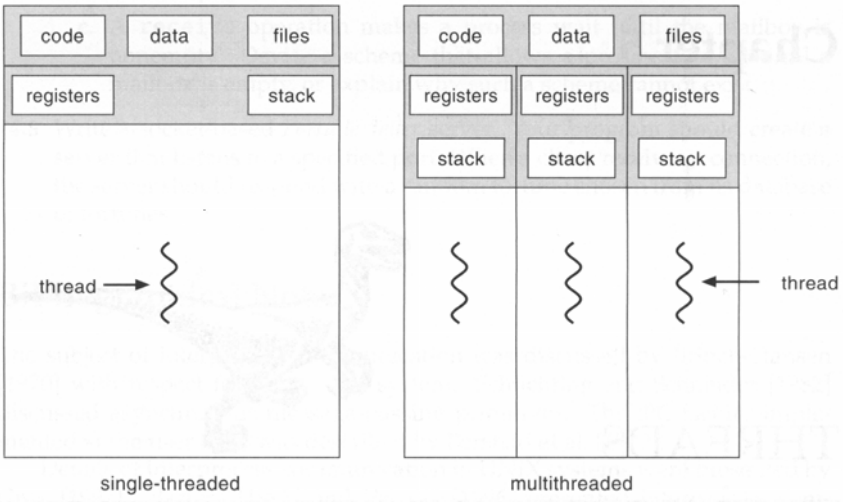
王振傑 (Chen-Chieh Wang)
ccwang@mail.ee.ncku.edu.tw

Department of Electrical Engineering, Feng-Chia University

Outline

- Overview
- Multithreading Models
- Threading Issues

Single and Multithreaded Processes



- Benefits
  - Responsiveness
  - Resource Sharing
  - Economy
  - Utilization of MP Architectures
- Thread vs. Process
  - Thread : Light-weight Process (LWP)
  - Process : Heavy-weight Process (HWP)

Department of Electrical Engineering, Feng-Chia University

Modern "Lightweight" Process with Threads

- Thread: a sequential execution stream within process (Sometimes called a "Lightweight process")
  - Process still contains a single Address Space
  - No protection between threads
- Multithreading: a single program made up of a number of different concurrent activities
  - Sometimes called multitasking, as in Ada…
- Why separate the concept of a thread from that of a process?
  - Discuss the "thread" part of a process (concurrency)
  - Separate from the "address space" (Protection)

Concurrency

- "Thread" of execution
  - Independent Fetch/Decode/Execute loop
  - Operating in some Address space
- Uni-programming: *one thread at a time*
  - MS/DOS, early Macintosh, Batch processing
  - Easier for operating system builder
  - Does this make sense for personal computers?
- Multi-programming: *more than one thread at a time*
  - Multics, UNIX/Linux, OS/2, Windows NT/2000/XP, Mac OS X

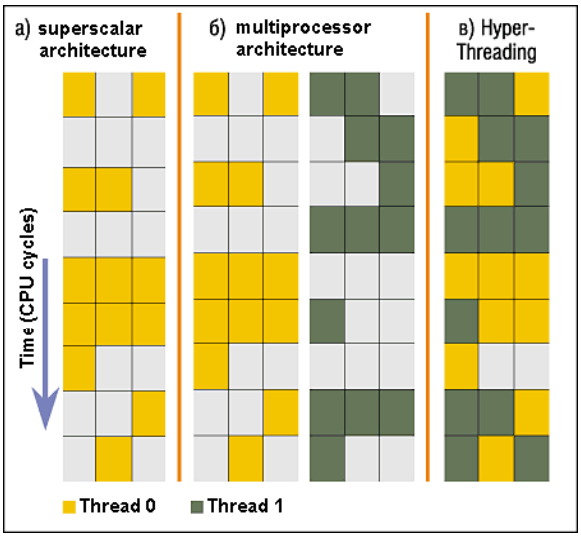
Department of Electrical Engineering, Feng-Chia University

The Basic Problem of Concurrency

- The basic problem of concurrency involves resources:
  - Hardware: single CPU, single DRAM, single I/O devices
  - Multiprogramming API: users think they have exclusive access to shared resources
- OS Has to coordinate all activity
  - Multiple users, I/O interrupts, …
  - How can it keep all these things straight?
- Basic Idea: Use Virtual Machine abstraction
  - Decompose hard problem into simpler ones
  - Abstract the notion of an executing program
  - Then, worry about multiplexing these abstract machines

Modern Technique: SMT/Hyperthreading

- Hardware technique
  - Exploit natural properties of superscalar processors to provide illusion of multiple processors
  - Higher utilization of processor resources
- Can schedule each thread as if were separate CPU
  - However, not linear speedup!
  - If have multiprocessor, should schedule each processor first



- Original technique called "Simultaneous Multithreading" (SMT)
  - See http://www.cs.washington.edu/research/smt/
  - Alpha, SPARC, Pentium 4 ("Hyperthreading"), Power 5

Department of Electrical Engineering, Feng-Chia University

Single-Threaded Example

- Imagine the following C program:

```
main() {
   ComputePI("pi.txt");
   PrintClassList("clist.text");
}
```

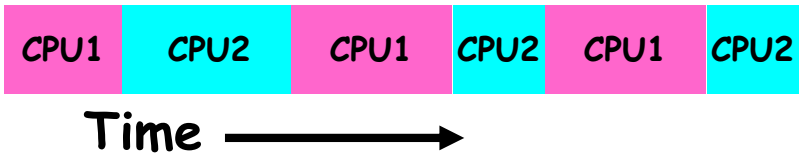
- What is the behavior here?
  - Program would never print out class list
  - Why? ComputePI would never finish

Use of Threads

- Version of program with Threads:

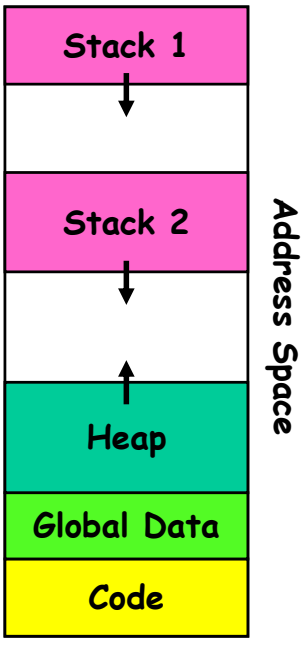
```
main() {
   CreateThread(ComputePI("pi.txt"));
   CreateThread(PrintClassList("clist.text"));
}
```

- What does "CreateThread" do?
  - Start independent thread running given procedure
- What is the behavior here?
  - Now, you would actually see the class list
  - This *should* behave as if there are two separate CPUs



Memory Footprint of Two-Thread Example

- If we stopped this program and examined it with a debugger, we would see
  - Two sets of CPU registers
  - Two sets of Stacks
- Questions:
  - How do we position stacks relative to each other?
  - What maximum size should we choose for the stacks?
  - What happens if threads violate this?
  - How might you catch violations?

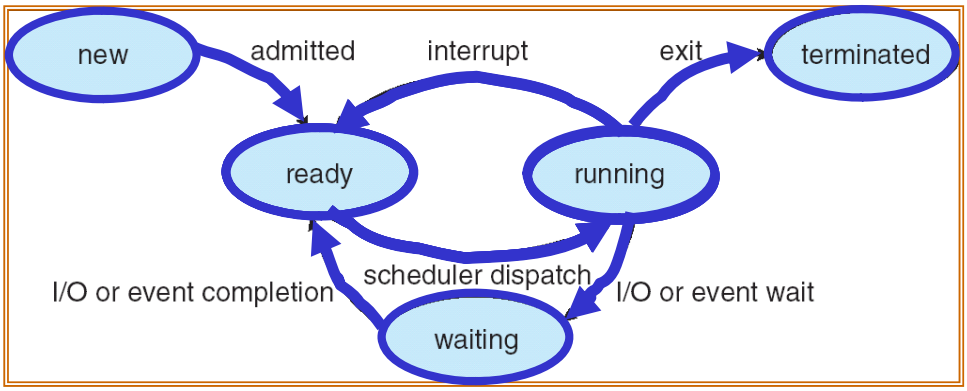


Per Thread State

- Each Thread has a Thread Control Block (TCB)
  - Execution State: CPU registers, program counter, pointer to stack
  - Scheduling info: State (more later), priority, CPU time
  - Accounting Info
  - Various Pointers (for implementing scheduling queues)
  - Pointer to enclosing process? (PCB)?
  - Etc (add stuff as you find a need)
- OS Keeps track of TCBs in protected memory
  - In Array, or Linked List, or …

Department of Electrical Engineering, Feng-Chia University

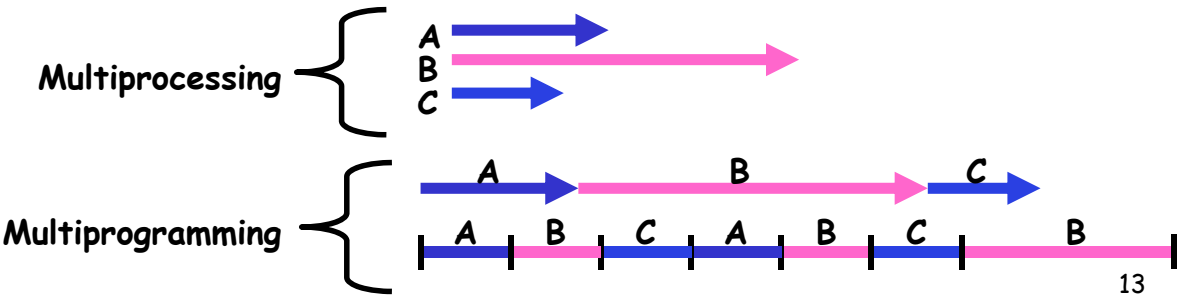
Lifecycle of a Thread (or Process)



- As a thread executes, it changes state:
  - new: The thread is being created
  - ready: The thread is waiting to run
  - running: Instructions are being executed
  - waiting: Thread waiting for some event to occur
  - terminated: The thread has finished execution
- "Active" threads are represented by their TCBs
  - TCBs organized into queues based on their state

Multiprocessing vs Multiprogramming

- Remember Definitions:
  - Multiprocessing ≡ Multiple CPUs
  - Multiprogramming ≡ Multiple Jobs or Processes
  - Multithreading ≡ Multiple threads per Process
- What does it mean to run two threads "concurrently"?
  - Scheduler is free to run threads in any order and interleaving: FIFO, Random, …
  - Dispatcher can choose to run each thread to completion or time-slice in big chunks or small chunks



Outline

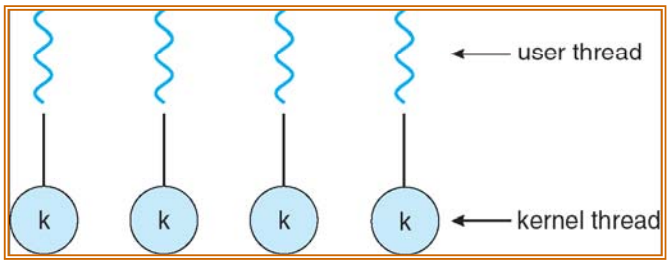
- Overview
- Multithreading Models
- Threading Issues

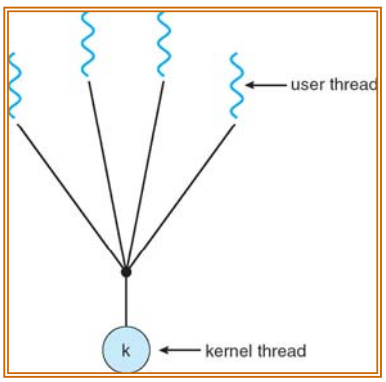
User and Kernel Threads

- **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

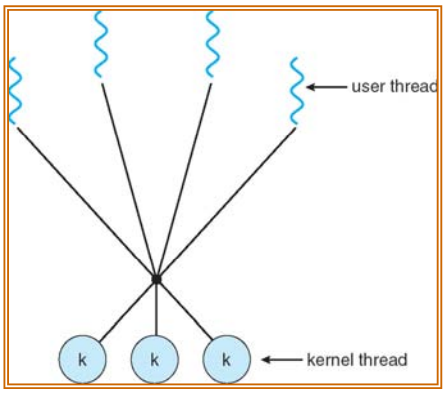
Multithreading Models

Simple One-to-One Threading Model





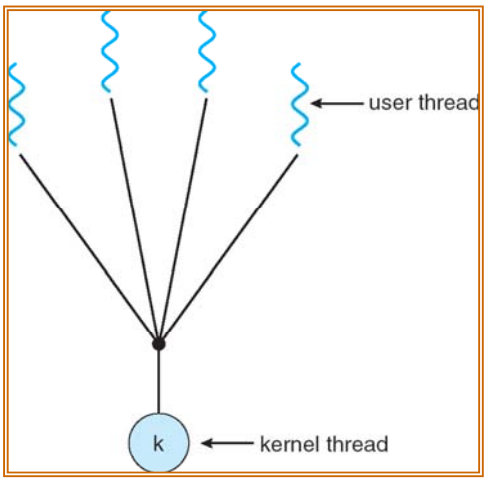
Many-to-One



Many-to-Many

Many-to-One Model

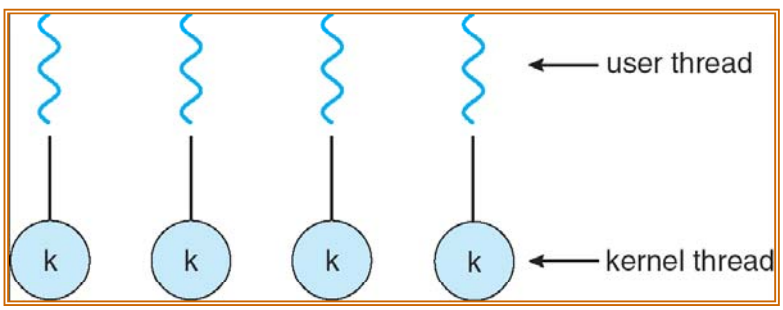
- Many user-level threads mapped to single kernel thread
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads



Department of Electrical Engineering, Feng-Chia University

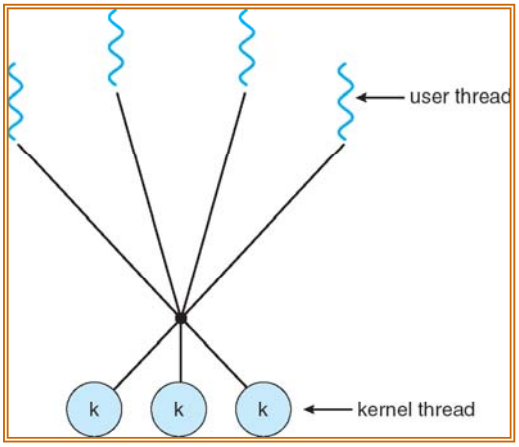
One-to-one Model

- Each user-level thread maps to kernel thread
- 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



Outline

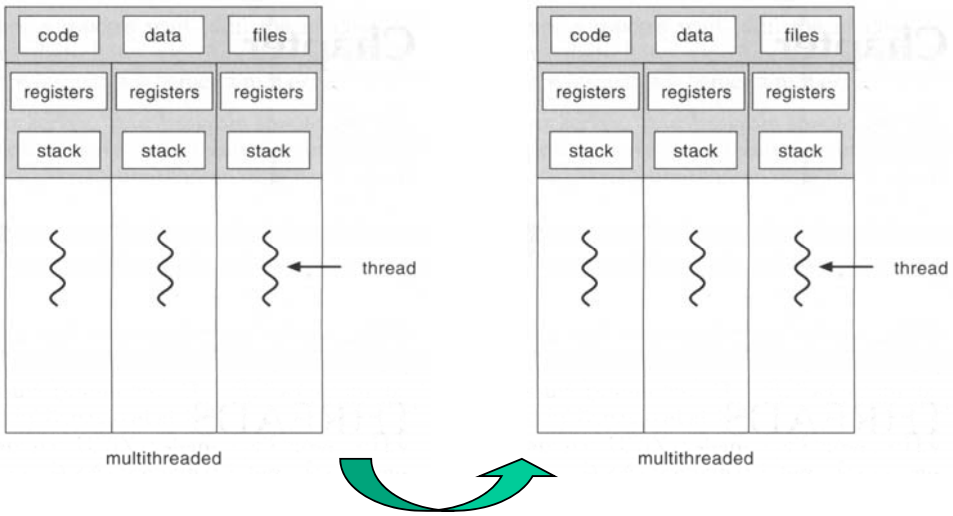
- Overview
- Multithreading Models
- Threading Issues

Threading Issues

1) Semantics of **fork()** and **exec()** system calls
2) Thread cancellation
3) Signal handling
4) Thread pools
5) Thread specific data

1. Semantics of fork() and exec()

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



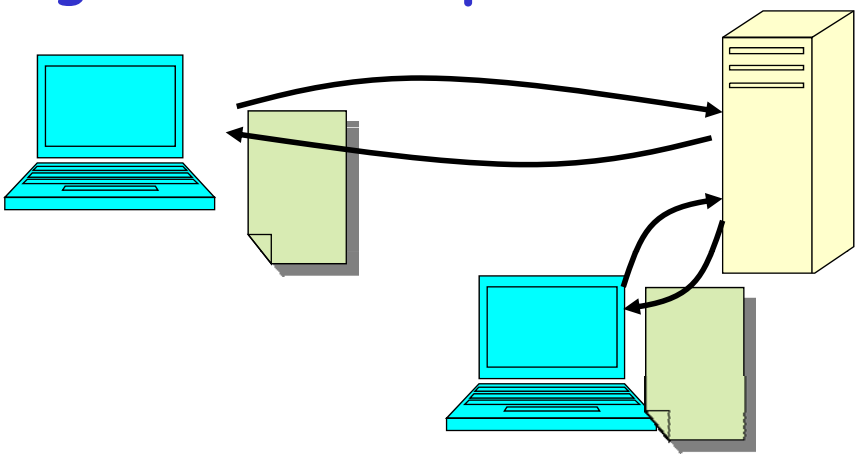
2. Thread Cancellation

- Terminating a thread before it has finished
- Two general approaches:
  - Asynchronous cancellation terminates the target thread immediately
  - Deferred cancellation allows the target thread to periodically check if it should be cancelled

3. 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 thread to receive all signals for the process

High-level Example: Web Server



- Server must handle many requests
- Non-cooperating version:

```
serverLoop() {
    con = AcceptCon();
    ProcessFork(ServiceWebPage(),con);
}
```

- What are some disadvantages of this technique?

Department of Electrical Engineering, Feng-Chia University

Threaded Web Server

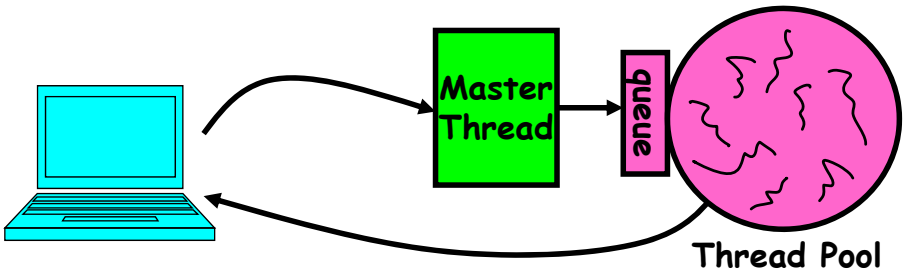
- Now, use a single process
- Multithreaded (cooperating) version:

```
serverLoop() {
    connection = AcceptCon();
    ThreadFork(ServiceWebPage(),connection);
}
```

- Looks almost the same, but has many advantages:
  - Can share file caches kept in memory, results of CGI scripts, other things
  - Threads are *much* cheaper to create than processes, so this has a lower per-request overhead

4. Thread Pools

- Create a number of threads in a pool where they await work
- Problem with previous version: Unbounded Threads
  - When web-site becomes too popular – throughput sinks
- Instead, allocate a bounded "pool" of worker threads, representing the maximum level of multiprogramming



5. Thread Specific Data

- Allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

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)

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)

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