Chapter 4: Multithreaded Programming

Chapter 4: Multithreaded Programming

- Overview
- Multicore Programming
- Multithreading Models
- Threading Issues
- Operating System Examples

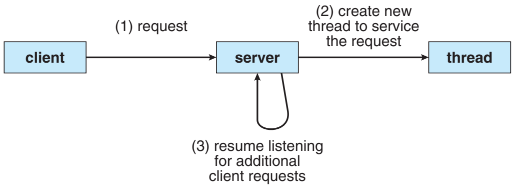
Objectives

- 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, Windows, and Java thread libraries
- To examine issues related to multithreaded programming
- To cover operating system support for threads in Windows and Linux

Motivation

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Multithreaded Server Architecture


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

Benefits

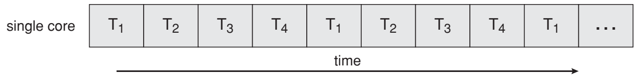
- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads with a process share resources of the process, easier than shared memory or message passing
- **Economy –** thread creation is much cheaper than process creation, thread switching also has much lower overhead than context switching (switching to a different process)
- **Scalability –** A process can take advantage of multiprocessor architectures by running multiple threads of the process simultaneously on different processors (CPUs).

Multicore Programming

- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency
- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation

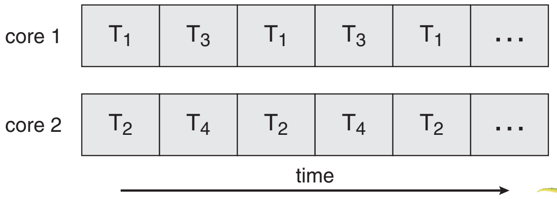
Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**



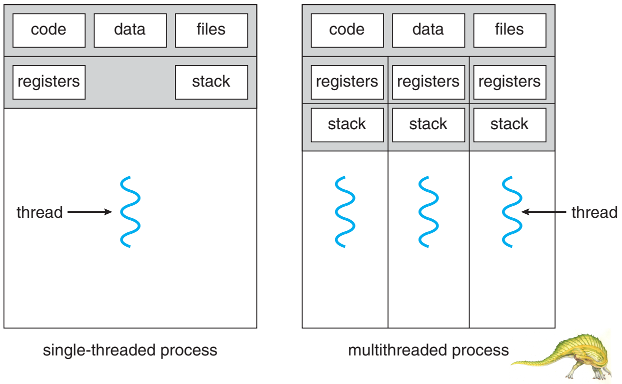
single core	T_1	T_2	T_3	T_4	T_1	T_2	T_3	T_4	T_1	. . .

- **Parallelism on a multi-core system:**



core 1	T_1	T_3	T_1	T_3	T_1	. . .
core 2	T_2	T_4	T_2	T_4	T_2	. . .

Single and Multithreaded Processes



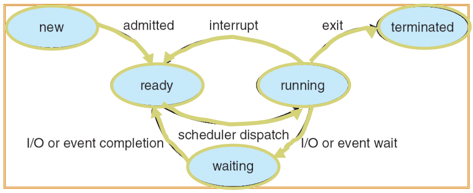
Thread State

- 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)?
- In Nachos: "thread" is a class that includes the TCB
- OS Keeps track of TCBs in protected memory
  - Array, or Linked List, or …

Thread State (Cont.)

- State shared by all threads in process/address space
  - Contents of memory (global variables, heap)
  - I/O state (file system, network connections, etc)
- State "private" to each thread
  - Kept in TCB ≡ Thread Control Block
  - CPU registers (including, program counter)
  - Execution stack – what is this?
- Execution Stack
  - Parameters, temporary variables
  - Keep program counters while called procedures are executing

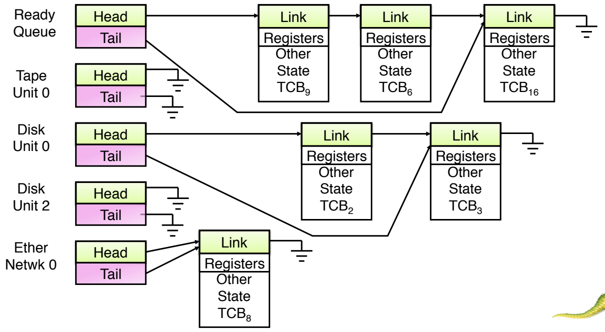
Lifecycle of a Thread



- 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

Ready Queue And Various I/O Device Queues

- Thread not running ⇒ TCB is in some scheduler queue
  - Separate queue for each device/signal/condition
  - Each queue can have a different scheduler policy



Ready Queue: Head, Tail → Link, Registers, Other State, TCB_9 → Link, Registers, Other State, TCB_6 → Link, Registers, Other State, TCB_{16}

Tape Unit 0: Head, Tail

Disk Unit 0: Head, Tail → Link, Registers, Other State, TCB_2 → Link, Registers, Other State, TCB_3

Disk Unit 2: Head, Tail

Ether Netwk 0: Head, Tail → Link, Registers, Other State, TCB_8

Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- S is serial portion
- N processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- I.e. if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As N approaches infinity, speedup approaches 1 / S

Serial portion of an application has disproportionate effect on performance gained by adding additional cores

Examples of Multithreaded Programs

- Embedded systems
  - Elevators, Planes, Medical systems, Wristwatches
  - Single Program, concurrent operations
- Most modern OS kernels
  - Internally concurrent to deal with concurrent requests by multiple users
  - But no protection needed within kernel
- Database Servers
  - Access to shared data by many concurrent users
  - Also background utility processing must be done
- Network Servers
  - Concurrent requests from network
  - Again, single program, multiple concurrent operations
  - File server, Web server, and airline reservation systems
- Parallel Programming (More than one physical CPU)
  - Split program into multiple threads for parallelism
  - This is called **Multiprocessing**

User Threads and Kernel Threads

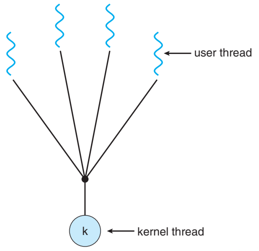
- User threads - management done by user-level threads library
- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads
- Kernel threads - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - 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
- One thread blocking causes all to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads



user thread

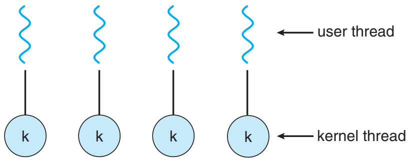
k

kernel thread

One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead

- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later



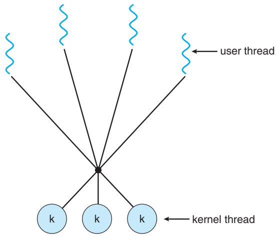
Many-to-Many Model

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

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

- Solaris prior to version 9

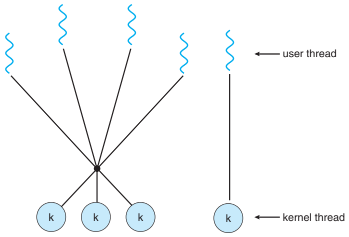
- Windows NT/2000 with the *ThreadFiber* package



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



Threading Issues

- Semantics of **fork()** and **exec()** system calls

- Signal handling
  - Synchronous and asynchronous

- Thread cancellation of target thread
  - Asynchronous or deferred

- Thread-local storage

Semantics of fork() and exec()

- Does `fork()` duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of fork

- `Exec()` usually works as normal – replace the running process including all threads

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 by one of two signal handlers:
     1. default
     2. user-defined

- Every signal has default handler that kernel runs when handling signal
  - User-defined signal handler can override default
  - For single-threaded, signal delivered to process

- Where should a signal be delivered for multi-threaded?
  - 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

Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is **target thread**
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- Pthread code to create and cancel a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```

Thread-Local Storage

- **Thread-local storage** (**TLS**) 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)
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations
- Similar to `static` data
  - TLS is unique to each thread

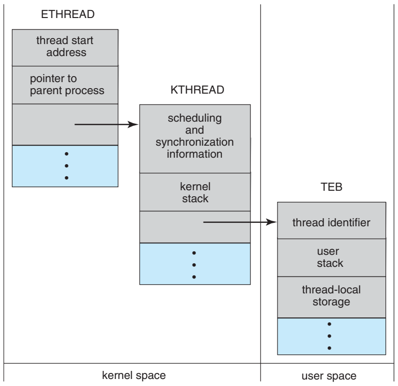
Operating System Examples

- Windows XP Threads
- Linux Thread

Windows Threads

- Windows implements the Windows API – primary API for Win 98, Win NT, Win 2000, Win XP, and Win 7
- Implements the one-to-one mapping, kernel-level
- Each thread contains
  - A thread id
  - Register set representing state of processor
  - Separate user and kernel stacks for when thread runs in user mode or kernel mode
  - Private data storage area used by run-time libraries and dynamic link libraries (DLLs)
- The register set, stacks, and private storage area are known as the **context** of the thread
- The primary data structures of a thread include:
  - ETHREAD (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space
  - KTHREAD (kernel thread block) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space
  - TEB (thread environment block) – thread id, user-mode stack, thread-local storage, in user space

Windows XP Threads Data Structures



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)
  - Flags control behavior

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.

- `struct task_struct` points to process data structures (shared or unique)