Threads

Bhuvana J
Associate Professor

Department of CSE, SSNCE

Agenda

Objectives

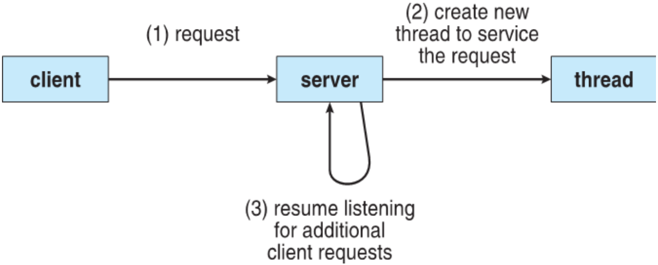
- To introduce the notion of a threada fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To discuss the APIs for the Pthreads, Windows, and Java thread libraries
- To explore several strategies that provide implicit threading
- To examine issues related to multithreaded programming
- To cover operating system support for threads in Windows and Linux

Presentation Outline

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



Benefits

- **Responsiveness** may allow continued execution if part of process is blocked, especially important for user interfaces **Resource Sharing** threads share resources of process, easier than shared memory or message passing **Economy** cheaper than process creation, thread switching lower overhead than context switching **Scalability** process can take advantage of multiprocessor architectures

Presentation Outline

Multicore Programming

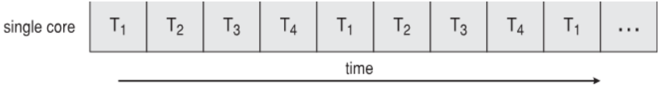
- **Multicore or multiprocessor** systems putting pressure on programmers, challenges include:
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging
- Parallelism implies a system can perform more than one task simultaneously
- Concurrency supports more than one task making progress
- Single processor / core, scheduler providing concurrency

Multicore Programming

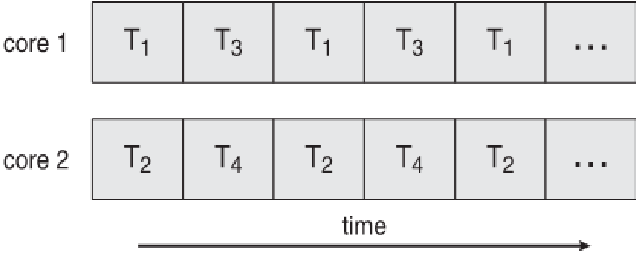
- Types of parallelism
  - Data parallelism distributes subsets of the same data across multiple cores, same operation on each
  - Task parallelism distributing threads across cores, each thread performing unique operation
- As # of threads grows, so does architectural support for threading
- CPUs have cores as well as hardware threads
- Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core

Concurrency vs. Parallelism

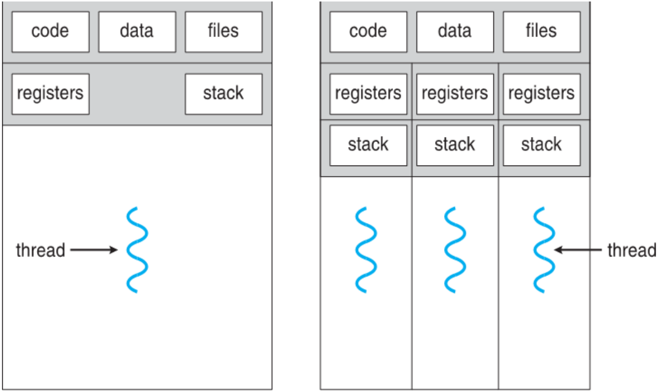
- Concurrent execution on single-core system:



- Parallelism on a multi-core system:



Single and Multithreaded Processes



single-threaded process

multithreaded process

Amdahls 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

$$Speedup \leq \frac{1}{S + \frac{1-S}{N}} \tag{1}$$

- That is, 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

User Threads and Kernel Threads

- User threads - management done by user-level threads library
- Three primary thread libraries:
  - POSIX Pthreads
  - Windows 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

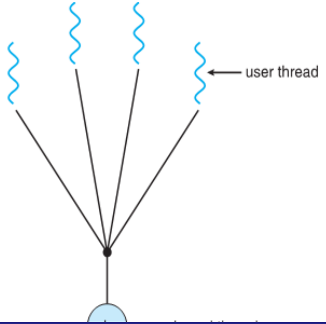
Presentation Outline

Multithreading Models

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

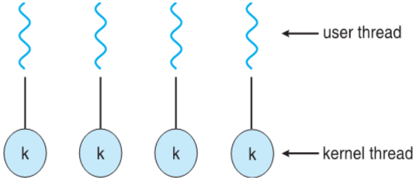
Many-to-One

- 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
- Examples: Solaris Green Threads and GNU Portable Threads



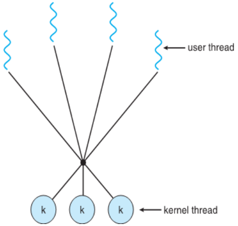
One-to-One

- 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, Linux, Solaris 9 and later



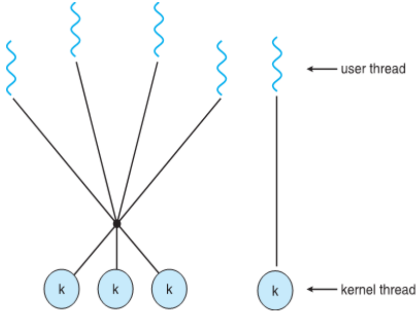
Many-to-Many Model

- 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 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



Presentation Outline

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
- Specification, not implementation
- 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)

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:

```
public interface Runnable
{
    public abstract void run();
}
```

- Extending Thread class
- Implementing the Runnable interface

Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Three methods explored
  - Thread Pools
  - OpenMP
  - Grand Central Dispatch
- Other methods include Microsoft Threading Building Blocks (TBB), java.util.concurrent package

Thread Pools

- 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
- Separating task to be performed from mechanics of creating task allows different strategies for running task i.e.Tasks could be scheduled to run periodically
- Windows API supports thread pools:

OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies parallel regions blocks of code that can run in parallel
  # *pragma omp parallel*
- Create as many threads as there are cores
  #*pragma omp parallel* for
  for(i=0;i<N;i++) {
  c[i] = a[i] + b[i];
  }
- Run for loop in parallel

Grand Central Dispatch

- Apple technology for Mac OS X and iOS operating systems
- Extensions to C, C++ languages, API, and run-time library
- Allows identification of parallel sections
- Manages most of the details of threading
- Block is in $\hat{\{\}}$ $\hat{\{}$ printf("I am a block"); }
- Blocks placed in dispatch queue
- Assigned to available thread in thread pool when removed from queue

Grand Central Dispatch

- Two types of dispatch queues:
- **serial** blocks removed in FIFO order, queue is per process, called **main queue**
  Programmers can create additional serial queues within program
- **concurrent** removed in FIFO order but several may be removed at a time
  Three system wide queues with priorities low, default, high

Presentation Outline

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
- Scheduler Activations

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:
     - default
     - 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

Signal Handling

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 Cancellation

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
- Cancellation only occurs when thread reaches cancellation point I.e. *pthread_testcancel()*
- Then cleanup handler is invoked
- On Linux systems, thread cancellation is handled through signals

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

Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads **lightweight process (LWP)**
- Appears to be a virtual processor on which process can schedule user thread to run
- Each LWP attached to kernel thread
- How many LWPs to create?
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the **upcall handler** in the thread library
- This communication allows an application to maintain the **correct number kernel threads**

Summary

- A thread is a flow of control within a process
- The benefits include increased responsiveness to the user, resource sharing within the process, economy, and scalability issues
- User-level threads are threads that are visible to the programmer and are unknown to the kernel
- Operating-system kernel supports and manages kernel-level threads.
- Three different types: The many-to-one model, one-to-one, and many-to-many model threads.

Test your Understanding

1. What are two differences between user-level threads and kernel-level threads? Under what circumstances is one type better than the other?
2. What resources are used when a thread is created? How do they differ from those used when a process is created?
3. Which of the following components of program state are shared across threads in a multithreaded process?
   a. Register values
   b. Heap memory
   c. Global variables
   d. Stack memory