Chapter 4 – Thread Concepts

Outline
4.1 Introduction
4.2 Definition of Thread
4.3 Motivation for Threads
4.4 Thread States: Life Cycle of a Thread
4.5 Thread Operations
4.6 Threading Models
  4.6.1 User-Level Threads
  4.6.2 Kernel-Level Threads
  4.6.3 Combining User- and Kernel-Level Threads
4.7 Thread Implementation Considerations
  4.7.1 Thread Signal Delivery
  4.7.2 Thread Termination
4.8 POSIX and Pthreads
4.9 Linux Threads
4.10 Windows XP Threads
4.11 Java Multithreading Case Study, Part 1: Introduction to Java Threads

Objectives

- After reading this chapter, you should understand:
  - the motivation for creating threads.
  - the similarities and differences between processes and threads.
  - the various levels of support for threads.
  - the life cycle of a thread.
  - thread signaling and cancellation.
  - the basics of POSIX, Linux, Windows XP and Java threads.
Thread vs. Process

- Process (Heavy-weight process)
  - Kernel level entity
  - Program counter, virtual memory map, file descriptors, signal table, user ID, current directory info, ...
  - The only way to access data in the process structure is via a system call

- Thread (Light-weight process)
  - User level entity
  - Thread structure is accessed directly with the thread library calls (user level functions)

Threads

- All threads in a process share:
  - Memory space
  - Other resources (file and socket descriptors)

- Each thread has its own:
  - CPU state (registers, program counter)
  - Stack

- Implemented in user space or kernel space
- Threads are efficient, but lack protection from each other
**History**

- The concept of multithreaded programming has existed in R&D labs for several decades.
- 1991: No major commercial OS contained a robust user-level thread library.
- 1996: Commercial OS began to have one (not yet completely).

**Why Multithreading?**
- General good sense and the recognition of a technology.
- The emergence of shared memory symmetric multiprocessors (SMPs).

**Examples**

- **Web browser**
  - thread1: Displaying images and text
  - thread2: Receiving the input from the users
  - thread3: Retrieving data from the network

- **Web servers**
  - With one thread (sequential)
    - Recv request → service the request → send the results
  - Improvement (concurrent)
    - Create another process for each request?
    - Create a thread for each request?
4.1 Introduction

• General-purpose languages such as Java, C#, Visual C++ .NET, Visual Basic .NET and Python have made concurrency primitives available to applications programmer

• Multithreading
  – Programmer specifies applications that contain threads of execution
  – Each thread designate a portion of a program that may execute concurrently with other threads

4.2 Definition of Thread

• Thread
  – Lightweight process (LWP)
  – Threads of instructions or thread of control
  – Shares address space and other global information with its process
  – Registers, stack, signal masks and other thread-specific data are local to each thread

• Threads may be managed by the operating system or by a user application

• Examples: Win32 threads, C-threads, Pthreads

Figure 4.1 Thread Relationship to Processes.
4.3 Motivation for Threads

- Threads have become prominent due to trends in
  - Software design
    - More naturally expresses inherently parallel tasks
  - Networking/distributed applications
  - Performance
    - Scales better to multiprocessor systems
  - Cooperation
    - Shared address space incurs less overhead than IPC

- Each thread transitions among a series of discrete thread states
- Threads and processes have many operations in common (e.g. create, exit, resume, and suspend)
- Thread creation does not require the operating system to initialize resources that are shared between parent processes and its threads
  - Reduces overhead of thread creation and termination compared to process creation and termination

4.4 Thread States: Life Cycle of a Thread

- Thread states (largely based on JAVA thread model)
  - Born state
    - Threads are not started until they are explicitly started
  - Ready state (runnable state)
  - Running state
  - Dead state
  - Blocked state
  - Waiting state
  - Sleeping state
    - Sleep interval specifies for how long a thread will sleep
4.4 Thread States: Life Cycle of a Thread

Figure 4.2 Thread life cycle.

4.5 Thread Operations

- Threads and processes have common operations
  - Create
  - Exit (terminate)
  - Suspend
  - Resume
  - Sleep
  - Wake

- Thread operations do not correspond precisely to process operations
  - Cancel
    - Indicates that a thread should be terminated, but does not guarantee that the thread will be terminated
    - Threads can mask the cancellation signal
  - Join
    - A primary thread (Windows XP) is associated with each process and wait for all other threads to exit by joining them
      - Cannot be terminated
      - The joining thread blocks until the thread it joined exits
4.6 Threading Models

- Three most popular threading models
  - User-level threads
  - Kernel-level threads
  - Combination of user- and kernel-level threads

4.6.1 User-level Threads

- User-level threads perform threading operations in user space
  - Threads are created by runtime libraries that cannot execute privileged instructions or access kernel primitives directly

User-level thread implementation

- Many-to-one thread mappings
  - Operating system maps all threads in a multithreaded process to single execution context
  - Advantages
    - User-level libraries can schedule its threads to optimize performance
    - Synchronization performed outside kernel, avoids context switches
    - More portable
  - Disadvantage
    - Kernel views a multithreaded process as a single thread of control
      - Can lead to suboptimal performance if a thread issues I/O
      - Cannot be scheduled on multiple processors at once

Figure 4.3 User-level threads.
4.6.2 Kernel-level Threads

- Kernel-level threads attempt to address the limitations of user-level threads by mapping each thread to its own execution context
  - Kernel-level threads provide a one-to-one thread mapping
    - Advantages
      - Increased scalability, interactivity, and throughput
    - Disadvantages
      - Overhead due to context switching and reduced portability due to OS-specific APIs
- Kernel-level threads are not always the optimal solution for multithreaded applications

4.6.3 Combining User- and Kernel-level Threads

- The combination of user- and kernel-level thread implementation
  - Many-to-many thread mapping (m-to-n thread mapping)
  - Number of user and kernel threads need not be equal
  - Can reduce overhead compared to one-to-one thread mappings by implementing thread pooling
- Worker threads (thread pooling)
  - Overcome the disadvantages of kernel level threads
  - Persistent kernel threads that occupy the thread pool
  - Improves performance in environments where threads are frequently created and destroyed
  - Each new thread is executed by a worker thread
4.6.3 Combining User- and Kernel-level Threads

- **Scheduler activation**
  - Overcome the disadvantages of user level threads
  - Unable to schedule the user level threads
  - Technique that enables user-level library to schedule its threads
  - A scheduler activation is a kernel level thread that can notify the a user level thread threading library of events
  - Occurs when the operating system calls a user-level threading library that determines if any of its threads need rescheduling

- **Problems with many-to-many mapping**
  - Complicates the OS designs
  - Not standard implementation
  - Solaris 2.2: many-to-many mapping and a user can specify the #of kernel threads per process
  - Solaris 2.6: scheduler activation
  - Solaris 8: abandoned many-to-many mapping and favor simpler one-to-one mapping
  - Windows XP: dynamically adjusts the number of worker threads in its thread pools in response to system load
4.7.1 Thread Signal Delivery

- Two types of signals
  - Synchronous:
    - Occur as a direct result of program execution
    - Example: illegal memory operation exception
    - Should be delivered to currently executing thread
  - Asynchronous
    - Occur due to an event typically unrelated to the current instruction
    - Threading library must determine each signal's recipient so that
      asynchronous signals are delivered properly

- Issues in signal delivery
  - User level thread: OS cannot determine which thread should receive the signal
  - Sender can specify receiving PID to deliver it to all threads???
    - UNIX and POSIX deliver it to all threads and thread signal masks
      are used

- Each thread is usually associated with a set of pending signals that are delivered when it executes
- Thread can mask all signals except those that it wishes to receive

Figure 4.6 Signal masking.
4.7.2 Thread Termination

- Thread termination (cancellation)
  - Differs between thread implementations
  - Prematurely terminating a thread can cause subtle errors in processes because multiple threads share the same address space
  - Some thread implementations allow a thread to determine when it can be terminated to prevent process from entering inconsistent state

4.8 POSIX and Pthreads

- POSIX (Portable Operating System Interface for Computing Environment)
  - A set of standards for OS interfaces published by IEEE Portable Application Standards Committee (PSAC)
  - Largely based on UNIX System V
- Threads that use the POSIX threading API are called Pthreads
  - POSIX states that processor registers, stack and signal mask are maintained individually for each thread
  - POSIX specifies how operating systems should deliver signals to Pthreads in addition to specifying several thread-cancellation modes

4.9 Linux Threads

- Kernel 1.0.9: user level thread
- Kernel 1.3.56: kernel level thread
- Kernel 2.6: one-to-one mapping
  - Process and threads are schedules at the same level
- Linux allocates the same type of process descriptor to processes and threads (tasks)
- Linux uses the UNIX-based system call fork to spawn child tasks
- To enable threading, Linux provides a modified version named clone
  - Clone accepts arguments that specify which resources to share with the child task
4.9 Linux Threads

Figure 4.7 Linux task state-transition diagram.

4.10 Windows XP Threads

- Threads
  - Actual unit of execution dispatched to a processor
  - Execute a piece of the process’s code in the process’s context, using the process’s resources
  - Execution context contains
    - Runtime stack
    - State of the machine’s registers
    - Several attributes

- Windows XP threads can create fibers
  - Fiber is scheduled for execution by the thread that creates it, rather than the scheduler
  - Easier to port applications that employ a user level thread
- Windows XP provides each process with a thread pool that consists of a number of worker threads, which are kernel threads that execute functions specified by user threads
  - A thread is in the standby state during the context switching
4.10 Windows XP Threads

Figure 4.8 Windows XP thread state-transition diagram.

4.11 Java Multithreading Case Study, Part I: Introduction to Java Threads

- Java allows the application programmer to create threads that can port to many computing platforms
- Threads
  - Created by class Thread
  - Execute code specified in a Runnable object's run method
- Java supports operations such as naming, starting and joining threads

```java
public class ThreadTester {
    public static void main(String[] args) {
        // Create three threads
        Thread thread1 = new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("Thread 1 started");
                try {
                    Thread.sleep(5000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("Thread 1 finished");
            }
        });
        Thread thread2 = new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("Thread 2 started");
                try {
                    Thread.sleep(5000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("Thread 2 finished");
            }
        });
        Thread thread3 = new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("Thread 3 started");
                try {
                    Thread.sleep(5000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("Thread 3 finished");
            }
        });

        thread1.start();
        thread2.start();
        thread3.start();
    }
}
```
4.11 Java Multithreading Case Study, Part I: Introduction to Java Threads

Figure 4.9 Java threads being created, starting, sleeping and printing. (Part 2 of 4.)

```java
25  // Class PrintThread controls thread execution
26  class PrintThread extends Thread {
27      private int sleeptime;
28
29      // sister thread is started by calling superclass constructor
30      public PrintThread(String name) {
31          super(name);
32
33      }
34
35      // pick random sleep time between 0 and 5 seconds
36      sleeptime = (int) (Math.random() * 5000.0);  // end PrintThread constructor
37
38      // method run is the code to be executed by new thread
39      public void run() {
40          try {
41              // sleep for sleeptime amount of time
42              Thread.sleep(sleeptime);
43              System.err.println(getName() + " done sleeping");
44          } catch (InterruptedException exception) {
45              exception.printStackTrace();
46          }
47          // end try
48
49      }
50
51      // print thread name
52      System.err.println(getName() + " begin sleeping");
53
54      // end method run
55
56      // end PrintThread
```

4.11 Java Multithreading Case Study, Part I: Introduction to Java Threads

Figure 4.9 Java threads being created, starting, sleeping and printing. (Part 3 of 4.)

Sample Output 1:

```
Sleeping thread
Sleeping thread
Sleeping thread
Sleeping thread
Sleeping thread
Sleeping thread
Sleeping thread
Sleeping thread
Sleeping thread
```

Sample Output 2:

```
Thread going to sleep for 127 milliseconds
Thread going to sleep for 199 milliseconds
Thread going to sleep for 302 milliseconds
Thread going to sleep for 583 milliseconds
Thread going to sleep for 1108 milliseconds
Thread going to sleep for 59 milliseconds
Thread going to sleep for 189 milliseconds
Thread going to sleep for 274 milliseconds
Thread going to sleep for 274 milliseconds
```

4.11 Java Multithreading Case Study, Part I: Introduction to Java Threads

Figure 4.9 Java threads being created, starting, sleeping and printing. (Part 4 of 4.)

Sample Output 3:

```
Sleeping thread
Sleeping thread
Sleeping thread
Sleeping thread
Sleeping thread
Sleeping thread
Sleeping thread
Sleeping thread
Sleeping thread
```

Sample Output 4:

```
Thread going to sleep for 127 milliseconds
Thread going to sleep for 199 milliseconds
Thread going to sleep for 302 milliseconds
Thread going to sleep for 583 milliseconds
Thread going to sleep for 1108 milliseconds
Thread going to sleep for 59 milliseconds
Thread going to sleep for 189 milliseconds
Thread going to sleep for 274 milliseconds
Thread going to sleep for 274 milliseconds
```