Chapter 4. Threads

1. Overview
2. Multithreading Models
3. Pthread
4. Java Threads
5. Threading Issues

1. Overview

- A traditional or ***heavyweight process*** is equal to a task with one *thread.*

- ☺ *How to implement FireFox?*
  - Displaying
  - User text input
  - Weather

- ☺ *One process?*
  - Impossible – E.g., how to display while the user is typing

- ☺ *Three or four processes?*
  - Very hard because they don't share data sections

Thread

- A ***thread*** (or ***lightweight process***) is a basic unit of CPU utilization; it consists of:
  - program counter
  - register set
  - stack space
- A thread shares with its peer threads in the same process its:
  - code section
  - data section
  - operating-system resources

  collectively known as a *task*.
- Example:
  - *FireFox*
    - Displaying
    - User text input
    - Weather

Thread - continued

- Another example:
  - Ball
  - User input – Z and / keys

Thread - continued

- In a multiple threaded task, while one server thread is blocked and waiting, a second thread in the same task can run.
  - Cooperation of multiple threads in same job confers higher throughput and improved performance.
  - Applications that require sharing a common buffer (i.e., producer-consumer) benefit from thread utilization.
- Threads provide a mechanism that allows sequential processes to make blocking system calls while also achieving **parallelism**.

- See Figure 4.1, “Single-threaded and multithreaded processes”.
- See Figure 4.2, “Multithreaded server architecture”.

Benefits

- Responsiveness
- Resource sharing
- Economy
- Utilization of multiprocessor architectures
  - For example, multicore programming these days
    - See Figure 4.3 – 4.4.

User and Kernel Threads

- Multithreading levels

1. User threads
   - Thread management done <u>by user-level threads library</u>
   - Three primary thread libraries:
     - POSIX Pthreads
     - Java threads
     - Win32 threads
2. Kernel threads
   - Supported <u>by the Kernel</u>
   - Context switching here
   - Examples:
     - Windows XP/2000
     - Solaris
     - Linux
     - Tru64 UNIX
     - Mac OS X

2. Multithreading Models

1. Many-to-One
2. One-to-One
3. Many-to-Many

Many-to-One

- See Figure 4.5.
- Many user-level threads are mapped to a single kernel thread.
- The entire process will block if a thread makes a blocking system call.
- Examples:
  - Solaris green threads
  - GNU portable threads

One-to-One

- See Figure 4.6.
- Each user-level thread maps to kernel thread.
- More concurrency by allowing another thread to run when a thread makes a blocking system call.
- Even on multiprocessors
- But system overhead
- Examples:
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

Many-to-Many Model

- Figure 4.7.
- Allows many user level threads to be mapped to many kernel threads.
- Allows the operating system to create a sufficient number of kernel threads.

Two-level Model

- See Figure 4.8.
- Similar to M:N, except that it allows a user thread to be **bound** to kernel thread.
- Examples:
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier

3. Pthreads

- Thread library
- 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)
- See Figure 4.9.

4. Java Threads

- Java threads are managed by the JVM.
- Java threads may be created by:
  - Extending `Thread` class
  - Implementing the `Runnable` interface

Extending the Thread Class

```
class Worker extends Thread
{
   public void run() {
      System.out.println("Worker Thread");
   }
}

public class ThreadExample
{
   public static void main(String args[]) {
      Worker runner = new Worker();
      runner.start();

      System.out.println("Main Thread");
   }
}
```

The Runnable Interface

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

- Another approach
  - Can extend other class while implementing multithreads
  - Sort of multiple inheritance
  - E.g., public class ThreadApplet extends Applet implements Runnable

Implementing the Runnable Interface

```
class Worker implements Runnable
{
   public void run() {
      System.out.println("Worker Thread ");
   }
}
public class ThreadExample
{
   public static void main(String args[]) {
      Runnable runner = new Worker();
      Thread thrd = new Thread(runner);
      thrd.start();

      System.out.println("Main Thread");
   }
}
```

Joining Threads

```
class Worker implements Runnable
{
   public void run() {
      System.out.println("Worker Thread");
   }
}

public class ThreadExample
{
   public static void main(String[] args) {
      Thread task = new Thread(new Worker());
      task.start();

      System.out.println("Main Thread");

      try { task.join(); }  // waiting until task is done
      catch (InterruptedException ie) { }

      System.out.println("Worker Done");
   }
}
```

Thread Cancellation

```java
Thread thrd = new Thread (new InterruptibleThread());
thrd.start();

. . .

// now interrupt it
thrd.interrupt();
```

Thread Cancellation

```
public Worker implements Runnable
{
   public void run()
   {
      while (true) {
         ...
         if (Thread.currentThread().isInterrupted()) {
            System.out.println("Interrupted!");
            break;
         }
      }

      ...
   }
}
```

Thread Specific Data

```
class Service
{
   private static ThreadLocal errorCode = new ThreadLocal();

   public static void transaction() {
      try {
         /**
          * some operation where an error may occur
          */
      catch (Exception e) {
         errorCode.set(e);
      }
   }

   /**
    * get the error code for this transaction
    */
   public static Object getErrorCode() {
      return errorCode.get();
   }
}
```

Thread Specific Data

```
class Worker implements Runnable
{
   private static Service provider;

   public void run() {
      provider.transaction();
      System.out.println(provider.getErrorCode());
   }
}
```

Producer-Consumer Problem

- See Figure 4.13, “The `Factory` class”.
- See Figure 4.14, “`Producer` thread”.
- See Figure 4.15, “`Consumer` thread”.

5. Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Thread cancellation
- Signal handling
- Thread pools
- Thread specific data
- Scheduler activations

Semantics of fork() and exec()

- ☺ *Does **fork()** duplicate only the calling thread or all threads?*
- **exec**() – the entire process including all threads

Thread Cancellation

- Terminating a thread before it has finished
  - Example – stop button while a web page on a *FireFox* window is loading
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately.
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled.

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 a default signal handler or a user-defined signal handler.
- 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.

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 to create a new thread.
  - Allows the number of threads in the application(s) to be bound to the size of the pool.
- Example: Web server

Thread Specific Data

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

Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application.
- ***Scheduler activations*** provide **upcalls** - a communication mechanism from the kernel to the thread library.
- This communication allows an application to maintain the correct number kernel threads.