Chapter 4: Threads
Chapter 4: Threads

- Overview
- Multithreading Models
- Thread Libraries
- Threading Issues
- Operating System Examples
- Windows XP Threads
- Linux Threads
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, Win32, and Java thread libraries
- To examine issues related to multithreaded programming
Single and Multithreaded Processes

- **Single-threaded process**
  - Contains a single thread of execution.
  - Resources (code, data, files, registers, stack) are shared.

- **Multithreaded process**
  - Contains multiple threads of execution.
  - Each thread has its own set of resources (code, data, files, registers, stack), ensuring isolated execution.

This diagram illustrates how resources are managed in single-threaded versus multithreaded processes, emphasizing the isolation and sharing of resources in each scenario.
Benefits

- Responsiveness
- Resource Sharing
- Economy
- Scalability
Multicore Programming

- Multicore systems putting pressure on programmers, challenges include:
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging
Multithreaded Server Architecture

1. Request

2. Create new thread to service the request

3. Resume listening for additional client requests
Concurrent Execution on a Single-core System

- A single core processes tasks in a sequence: T₁, T₂, T₃, T₄, T₁, T₂, T₃, T₄, T₁, ...
- Time progresses as the tasks are executed.
Parallel Execution on a Multicore System

core 1

| T₁ | T₃ | T₁ | T₃ | T₁ | ...
|

core 2

| T₂ | T₄ | T₂ | T₄ | T₂ | ...
|

time
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

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

- Many user-level threads mapped to single kernel thread

- Examples:
  - Solaris Green Threads
  - GNU Portable Threads
Many-to-One Model
One-to-One

- Each user-level thread maps to kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later
One-to-one Model

user thread

kernel thread
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
Many-to-Many Model
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
Two-level Model
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
- 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
- Java threads may be created by:
  - Implementing the Runnable interface

```java
public interface Runnable
{
    public abstract void run();
}
```
class MutableInteger
{
    private int value;
    public int getValue() {
        return value;
    }
    public void setValue(int value) {
        this.value = value;
    }
}

class Summation implements Runnable
{
    private int upper;
    private MutableInteger sumValue;
    public Summation(int upper, MutableInteger sumValue) {
        this.upper = upper;
        this.sumValue = sumValue;
    }
    public void run() {
        int sum = 0;
        for (int i = 0; i <= upper; i++)
            sum += i;
        sumValue.setValue(sum);
    }
}
Java Threads - Example Program

```java
public class Driver {
    public static void main(String[] args) {
        if (args.length > 0) {
            if (Integer.parseInt(args[0]) < 0)
                System.err.println(args[0] + " must be >= 0.");
            else {
                // create the object to be shared
                MutableInteger sum = new MutableInteger();
                int upper = Integer.parseInt(args[0]);
                Thread thrd = new Thread(new Summation(upper, sum));
                thrd.start();
                try {
                    thrd.join();
                    System.out.println("The sum of " + upper + " is " + sum.getValue());
                } catch (InterruptedException ie) {
                }
            }
        } else
            System.err.println("Usage: Summation <integer value>");
    }
}
```
Java Thread States

- **NEW**
  - Transition to **RUNNABLE** via `start()` method
- **RUNNABLE**
  - Transitions to **TERMINATED** if `run()` method exits
  - Transitions to **BLOCKED**, **WAITING**, or **TIMED_WAITING** via `join()` or `join(time)` methods
  - Transition to **BLOCKED** via `locking` method
import java.util.Date;

public class Factory
{
    public static void main(String args[]) {
        // create the message queue
        Channel<Date> queue = new MessageQueue<Date>();

        // Create the producer and consumer threads and pass
        // each thread a reference to the MessageQueue object.
        Thread producer = new Thread(new Producer(queue));
        Thread consumer = new Thread(new Consumer(queue));

        // start the threads
        producer.start();
        consumer.start();
    }
}

import java.util.Date;

class Producer implements Runnable
{
    private Channel<Date> queue;

    public Producer(Channel<Date> queue) {
        this.queue = queue;
    }

    public void run() {
        Date message;

        while (true) {
            // nap for awhile
            SleepUtilities.nap();

            // produce an item and enter it into the buffer
            message = new Date();
            System.out.println("Producer produced "+message);
            queue.send(message);
        }
    }
}
import java.util.Date;

class Consumer implements Runnable
{
    private Channel<Date> queue;

    public Consumer(Channel<Date> queue) {
        this.queue = queue;
    }

    public void run() {
        Date message;

        while (true) {
            // nap for awhile
            SleepUtilities.nap();

            // consume an item from the buffer
            message = queue.receive();

            if (message != null)
                System.out.println("Consumer consumed " + message);
        }
    }
}
Threading Issues

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

- **Thread cancellation** of target thread
  - Asynchronous or deferred

- **Signal** handling

- Thread pools

- Thread-specific data

- Scheduler activations
Semantics of fork() and exec()

- Does `fork()` duplicate only the calling thread or all threads?
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
Signals are used in UNIX systems to notify a process that a particular event has occurred.

A signal handler is used to process signals.

- Signal is generated by particular event
- Signal is delivered to a process
- 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
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.
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)
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.
Operating System Examples

- Windows XP Threads
- Linux Thread
Windows XP Threads

![Diagram of Windows XP Threads]

- **ETHREAD**
  - thread start address
  - pointer to parent process

- **KTHREAD**
  - scheduling and synchronization information
  - kernel stack

- **TEB**
  - thread identifier
  - user stack
  - thread-local storage

---

Operating System Concepts with Java – 8th Edition

Silberschatz, Galvin and Gagne ©2009
## Linux Threads

<table>
<thead>
<tr>
<th>flag</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLONE_FS</td>
<td>File-system information is shared.</td>
</tr>
<tr>
<td>CLONE_VM</td>
<td>The same memory space is shared.</td>
</tr>
<tr>
<td>CLONE_SIGHAND</td>
<td>Signal handlers are shared.</td>
</tr>
<tr>
<td>CLONE_FILES</td>
<td>The set of open files is shared.</td>
</tr>
</tbody>
</table>
Windows XP Threads

- Implements the one-to-one mapping, kernel-level

- 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 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)
End of Chapter 14