Threads

- Overview
- Multithreading Models
- Threading Issues
- Pthreads
- Windows XP Threads
- Linux Threads
- Java Threads
Benefits

- Responsiveness
- Resource Sharing
- Economy
- Utilization of MP Architectures
User Threads

- Thread management done by user-level threads library
- Three primary thread libraries:
  - POSIX Pthreads
  - Java threads
  - Win32 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**

- 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**

- 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

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
Two-level Model

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?

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
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
- 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
**Light Weight Process**

- A user thread
- A lightweight process
- A kernel thread

**Pthreads**

- 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)
**Pthreads**

```c
int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

main(int argc, char *argv[]) {
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of attributes for the thread */
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* create the thread */
    pthread_create(&tid, &attr, runner, argv[1]);
    /* now wait for the thread to exit */
    pthread_join(tid, NULL);
    printf("sum = %d\n", sum);
}

void *runner(void *param) {
    int upper = atoi(param);
    int i;
    sum = 0;
    if (upper > 0) {
        for (i = 1; i <= upper; i++)
            sum += i;
    }
    pthread_exit(0);
}
```

**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

```java
class Worker1 extends Thread {
    public void run() {
        System.out.println("I Am a Worker Thread");
    }
}

public class First {
    public static void main(String args[]) {
        Worker1 runner = new Worker1();
        runner.start();

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

### The Runnable Interface

```java
public interface Runnable {
    public abstract void run();
}
```
### Implementing the Runnable Interface

```java
class Worker2 implements Runnable {
    public void run() {
        System.out.println("I Am a Worker Thread");
    }
}
```

```java
public class Second {
    public static void main(String args[]) {
        Runnable runner = new Worker2();
        Thread thrd = new Thread(runner);
        thrd.start();

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

### Java Thread States

![Java Thread States Diagram]
## Joining Threads

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

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

        try { task.join(); }
        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 class InterruptibleThread implements Runnable {
    public void run() {
        while (true) {
            /**
             * do some work for awhile
             */
            if (Thread.currentThread().isInterrupted()) {
                System.out.println("I'm interrupted!");
                break;
            }
        }
        // clean up and terminate
    }
}

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 Thread

class Producer implements Runnable
{
    private Channel mbox;

    public Producer(Channel mbox) {
        this.mbox = mbox;
    }

    public void run() {
        Date message;

        while (true) {
            SleepUtilities.nap();
            message = new Date();
            System.out.println("Producer produced " + message);

            // produce an item & enter it into the buffer
            mbox.send(message);
        }
    }
}

Consumer Thread

class Consumer implements Runnable
{
    private Channel mbox;

    public Consumer(Channel mbox) {
        this.mbox = mbox;
    }

    public void run() {
        Date message;

        while (true) {
            SleepUtilities.nap();
            // consume an item from the buffer
            System.out.println("Consumer wants to consume.");

            message = (Date)mbox.receive();
            if (message != null)
                System.out.println("Consumer consumed " + message);
        }
    }
}
Windows XP Threads

- Implements the one-to-one mapping
- 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 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

<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>