/* cc thisfile.c -lpthread */

#include <pthread.h>    /* POSIX thread library */
#define NUM_THREADS 5
#define SLEEP_TIME 10

void *sleeping(void *);    /* thread routine */
int i;
thread_t tid[NUM_THREADS]; /* array of thread IDs */

int main(int argc, char *argv[])
{
    /* Implementation of POSIX Threads */

    /* Thread creation */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i], NULL, sleeping, (void *)SLEEP_TIME);

    /* Main thread waiting for other threads to complete execution */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);
    break;

    /* Main thread reporting that all is done */
    printf("main() reporting that all %d threads have terminated\n", i);
    return (0);

} /* main */

/* Function called in each pthread_create() above, which is
   going to be executed by all the threads concurrently */
void *sleeping(void *arg)
{
    int sleep_time = (int)arg;
    printf("thread %d sleeping %d seconds \n", thr_self(), sleep_time);
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 1:**
- Starting threads
- Threads started, main ends
- thread1 going to sleep for 1217 milliseconds
- thread2 going to sleep for 3989 milliseconds
- thread3 going to sleep for 662 milliseconds
- thread3 done sleeping
- thread1 done sleeping
- thread2 done sleeping

**Sample Output 2:**
- Starting threads
- thread1 going to sleep for 314 milliseconds
- thread2 going to sleep for 1990 milliseconds
- Threads started, main ends
- thread3 going to sleep for 3016 milliseconds
- thread1 done sleeping
- thread2 done sleeping
- thread3 done sleeping
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.)

```java
46       Thread.sleep( sleepTime );
47   } // end try
48
49         // if thread interrupted during sleep, print stack trace
50   catch ( InterruptedException exception ) {
51         exception.printStackTrace();
52       } // end catch
53
54        // print thread name
55   System.err.println( getName() + " done sleeping" );
56
57 } // end method run
58
59 } // end class PrintThread
```
Figure 4.9 Java threads being created, starting, sleeping and printing. (Part 2 of 4.)

```java
// class PrintThread controls thread execution
class PrintThread extends Thread {
    private int sleepTime;

    // assign name to thread by calling superclass constructor
    public PrintThread( String name )
    {
        super( name );

        // pick random sleep time between 0 and 5 seconds
        sleepTime = ( int ) ( Math.random() * 5001 );
    } // end PrintThread constructor

    // method run is the code to be executed by new thread
    public void run()
    {
        // put thread to sleep for sleepTime amount of time
        try {
            System.err.println( getName() + " going to sleep for " + sleepTime + " milliseconds" );
        }
    }
} // end PrintThread class
```
4.11 Java Multithreading Case Study, Part I: Introduction to Java Threads

Figure 4.9 Java threads being created, starting, sleeping and printing. (Part 1 of 4.)

```java
// Fig. 4.9: ThreadTester.java
// Multiple threads printing at different intervals.

public class ThreadTester {

    public static void main( String [] args )
    {
        // create and name each thread
        PrintThread thread1 = new PrintThread( "thread1" );
        PrintThread thread2 = new PrintThread( "thread2" );
        PrintThread thread3 = new PrintThread( "thread3" );

        System.err.println( "Starting threads" );

        thread1.start(); // start thread1; place it in ready state
        thread2.start(); // start thread2; place it in ready state
        thread3.start(); // start thread3; place it in ready state

        System.err.println( "Threads started, main ends\n" );

    } // end main

} // end class ThreadTester
```
4.9 Linux Threads

Figure 4.7 Linux task state-transition diagram.
4.7.1 Thread Signal Delivery

Figure 4.6 Signal masking.
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
4.6.1 User-level Threads

Figure 4.3 User-level threads.

All of a process’s threads map to a single execution context.
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
4.5 Thread Operations

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

\[ \land C \quad \text{Quit} \]
\[ \land Z \quad \text{Suspend} \]
\[ \% \text{more} \quad \log.out \quad (\text{resume normal background}) \]
\[ \% \text{kill} -9 \quad 34700 \quad \text{foreground} \]
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
4.2 Definition of Thread

Figure 4.1 Thread Relationship to Processes.
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
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
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

© 2004 Delmar & Associates, Inc. All rights reserved.