Chapter 4 – Thread Concepts

Outline

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.

4.1 Introduction

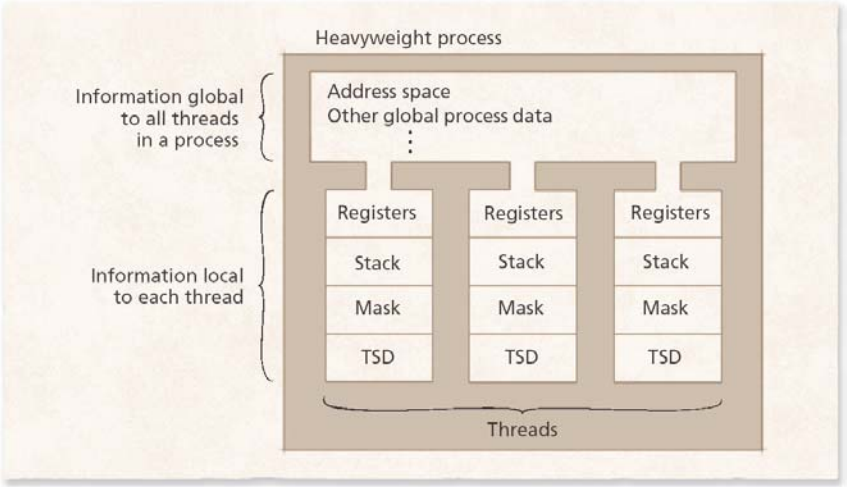
- General-purpose languages such as Java, C#, Visual C++ .NET, Visual Basic .NET and Python have made concurrency primitives available to applications programmer
- Multithreading
  - Programmer specifies applications contain threads of execution
  - Each thread designate a portion of a program that may 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

4.2 Definition of Thread

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
  - Performance
    - Scales better to multiprocessor systems
  - Cooperation
    - Shared address space incurs less overhead than IPC

4.3 Motivation for Threads

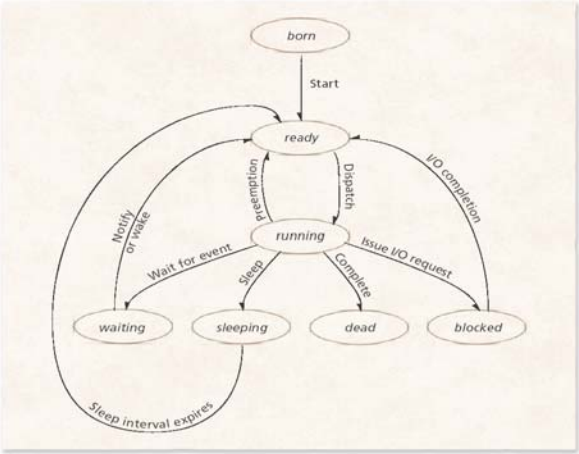
- Each thread transitions among a series of discrete thread states
- Threads and processes have many operations in common (e.g. create, exit, resume, and suspend)
- Thread creation does not require operating system to initialize resources that are shared between parent processes and its threads
  - Reduces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

4.4 Thread States: Life Cycle of a Thread

- Thread states
  - *Born* state
  - *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

4.5 Thread Operations

- 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 can wait for all other threads to exit by joining them
    - The joining thread blocks until the thread it joined exits

4.6 Threading Models

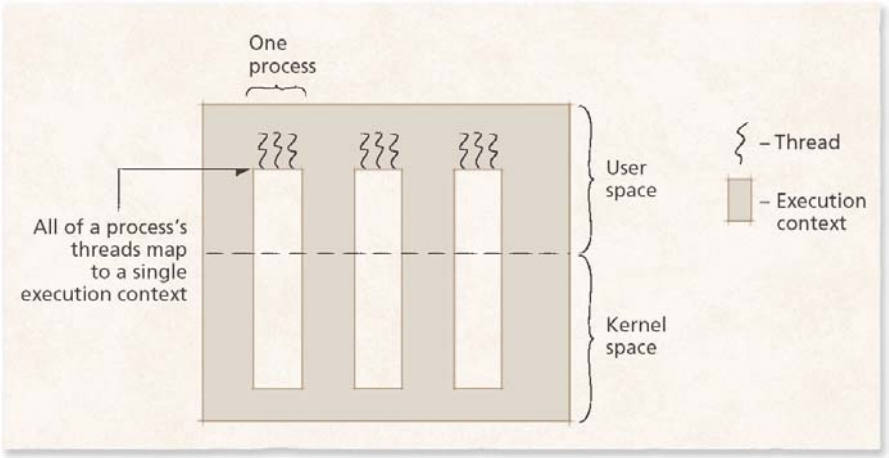
- Three most popular threading models
  - User-level threads
  - Kernel-level threads
  - Combination of user- and kernel-level threads

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

4.6.1 User-level Threads

Figure 4.3 User-level threads.

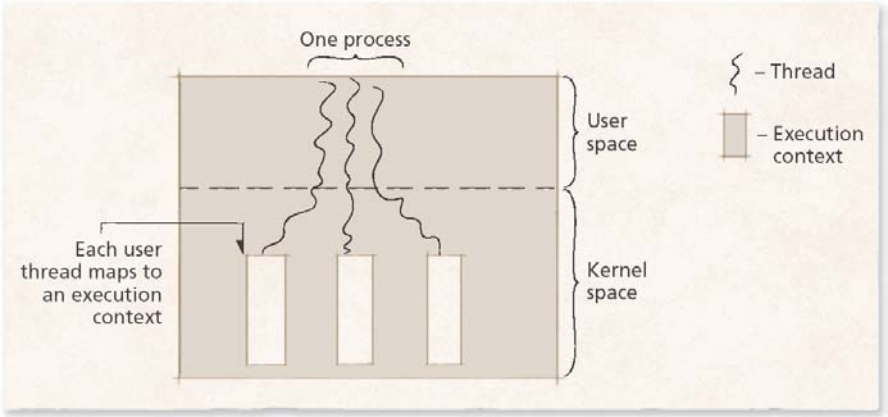


4.6.2 Kernel-level Threads

- 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

4.6.2 Kernel-level Threads

Figure 4.4 Kernel-level threads.

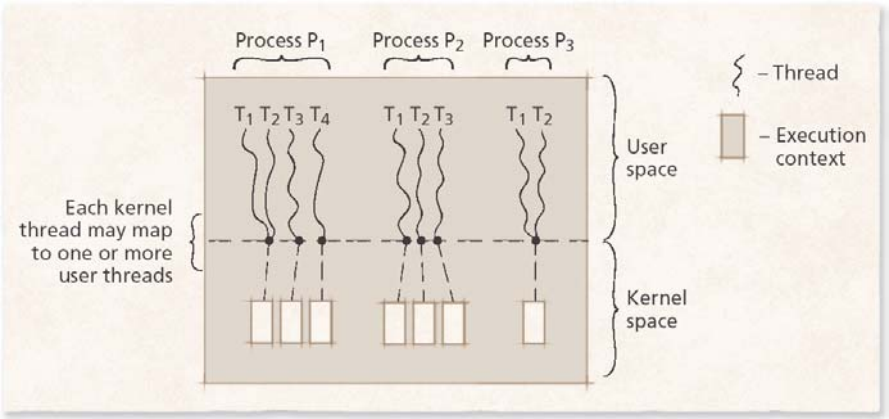


4.6.3 Combining User- and Kernel-level Threads

- 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
  - 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
- Scheduler activation
  - Technique that enables user-level library to schedule its threads
  - Occurs when the operating system calls a user-level threading library that determines if any of its threads need rescheduling

4.6.3 Combining User- and Kernel-level Threads

Figure 4.5 Hybrid threading model.

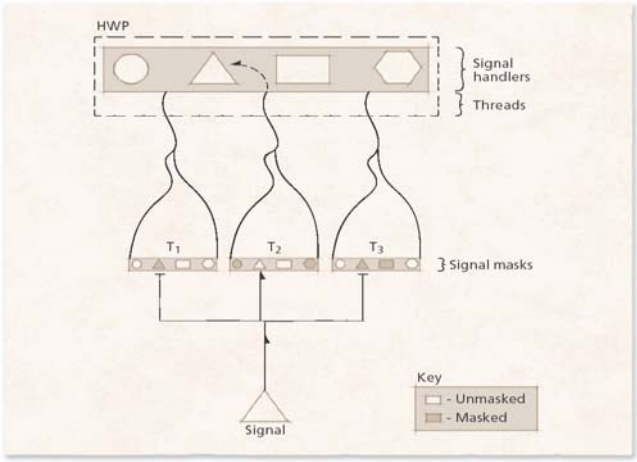


4.7.1 Thread Signal Delivery

- Two types of signals
  - Synchronous:
    - Occur as a direct result of program execution
    - 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
- 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

4.7.1 Thread Signal Delivery

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

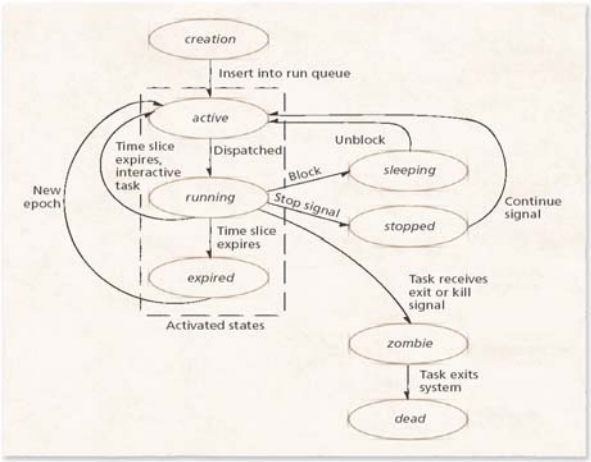
- Threads that use the POSIX threading API are called Pthreads
  - POSIX states that processor registers, stack and signal mask are maintained individually for each thread
  - POSIX specifies how operating systems should deliver signals to Pthreads in addition to specifying several thread-cancellation modes

4.9 Linux Threads

- 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

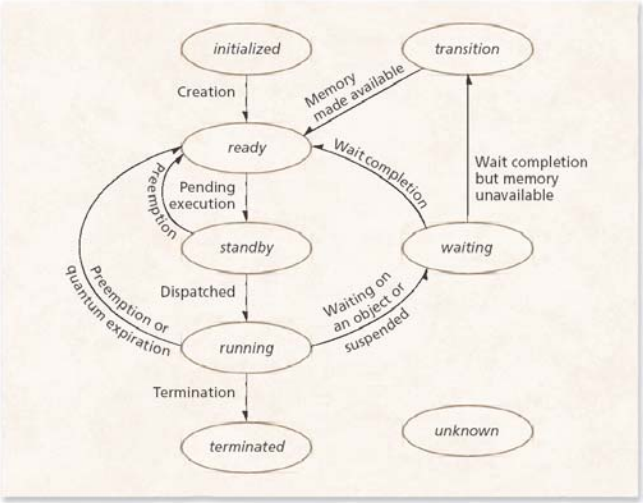
- Threads
  - Actual unit of execution dispatched to a processor
  - Execute a piece of the process's code in the process's context, using the process's resources
  - Execution context contains
    - Runtime stack
    - State of the machine's registers
    - Several attributes

4.10 Windows XP Threads

- Windows XP threads can create fibers
  - Fiber is scheduled for execution by the thread that creates it, rather than the scheduler
- 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

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

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

```


4.11 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
// 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" );

```

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.)

```
         Thread.sleep( sleepTime );
      } // end try

      // if thread interrupted during sleep, print stack trace
      catch ( InterruptedException exception ) {
         exception.printStackTrace();
      } // end catch

      // print thread name
      System.err.println( getName() + " done sleeping" );

   } // end method run

} // end class 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 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
```