Chapter 4: Threads

Presented By: Dr. El-Sayed M. El-Alfy

Note: Most of the slides are compiled from the textbook and its complementary resources

Recap

- Process Concepts
- Process Scheduling
- Process Creation and Termination
- Inter-process Communication
- Communication in Client-Server Systems

Objectives/Outline

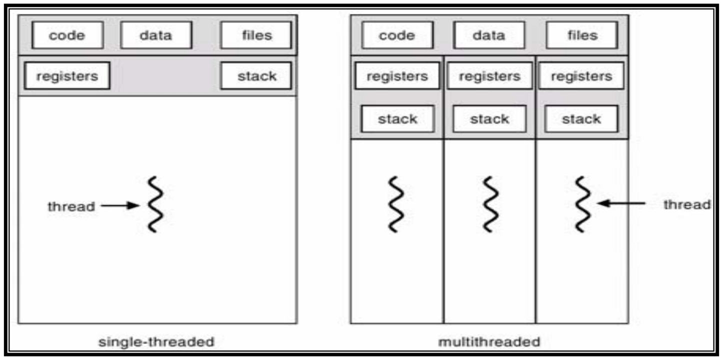
Objectives
- To introduce the concept of threads and multithreading models
- To discuss the APIs for Pthreads, Win32 and Java
- To explore how Windows and Linux OSs support threads at the kernel level

Outline
- Overview
- Multithreading Models
- Thread Libraries
- Threading Issues
- OS Examples

Overview

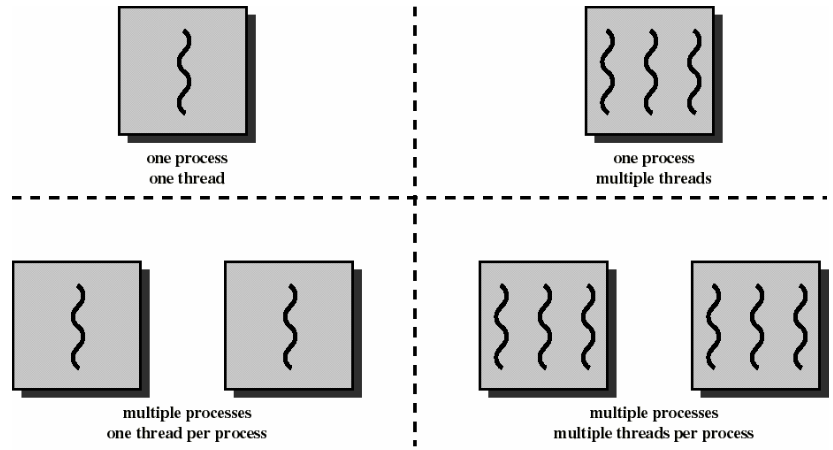
- A thread is a basic unit of CPU utilization and sometimes is called a lightweight process (LWP)
- A traditional (heavyweight) process has a single thread of control
- Each thread has:
  - thread ID, program counter, register set, and stack
- It shares with other threads belonging to the same process:
  - code section, data section, other OS resources such as open files and signals
- If the process has multiple threads of control, it can do more than one task at a time

Overview (Cont.): Single threaded vs. multithreaded



- A simple way to think about a process is as an address space (containing code, data, etc.) in which there is a single thread of execution
- The thread is the active part of a process

Overview (Cont.): Threads Versus Processes



Overview (Cont.): Motivation

- A process can do several things concurrently by running more than a single thread
- Each thread is a different stream of control that can execute its instructions independently
- Examples
  - A web Browser may have:
    - A thread to display images or text
    - A thread to retrieve data from the network
  - Word processor may have
    - A thread to display graphics
    - A thread to respond to keystrokes from the user
    - A thread to perform spelling and grammar checking

Overview (Cont.): Motivation

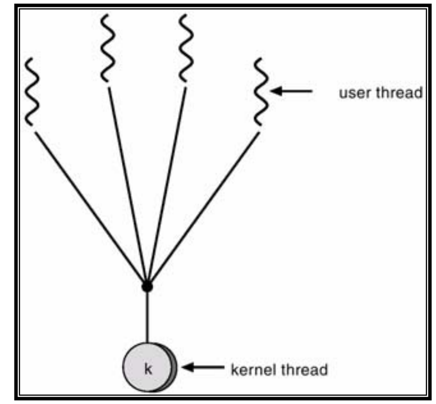
- Benefits
  - Responsiveness: multithreading is useful in an interactive application. E.g., a multithreaded web browser can allow user interaction even though an image is still downloading
  - Resource sharing: threads share memory and resources allocated to the process to which they belong, e.g. code sharing, where different threads of activity all within the same address space
  - Economy: allocating memory and resources for process creation is costly but it is more economical to create and context-switch threads
  - Utilization of multiprocessor architectures: threads may run in parallel on different processors; hence increase the utilization

Multithreading Models

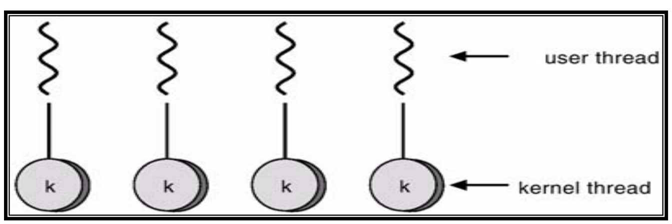
- Support of thread can be either at user level (user threads) or at the kernel level (kernel threads)
- User threads:
  - Supported above the kernel and managed without kernel support
  - Implemented by the thread library at the user level
  - User thread are generally fast to create and manage
  - Example of user thread libraries: Pthreads and Mach C-threads
- Kernel threads:
  - The kernel performs creation, scheduling, and management
  - Supported by the OS such as Windows XP, Linux, Mac OS X, Solaris and True64 Unix
- Relationship between user threads and kernel threads
  - Many-to-one model
  - One-to-one model
  - Many-to-many model

Multithreading Models: Many-to-One

- Maps many user threads to one kernel thread
- Thread management is done by the thread library in user space; Efficient
- Drawbacks
  - Entire process will block if a thread makes a blocking system call
  - Unable to run multiple threads in parallel on multiprocessors
- Example: Green threads library for Solaris



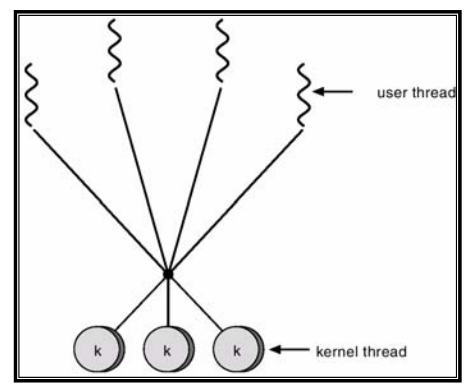
Multithreading Models – One-to-One



- Maps each user thread to a kernel thread
- Provides more concurrency than the many-to-one model (i.e., the kernel allows another thread to run when a thread makes a blocking system call)
- Allows multiple threads to run in parallel on multiprocessors
- Drawback
  - creating a user thread requires creating the corresponding kernel thread which can burden the overall system performance. Therefore, most implementations of this model restrict the number of threads supported by this system

Multithreading Models – Many-to-Many

- multiplex many user-level threads to a smaller or equal number of kernel threads
- The number of kernel threads may be specific to either a particular application or a particular machine
- An application may be allocated more threads on a multiprocessor than a uniprocessor machine
- Tow-level model: a popular variation of many-to-many which allows ULT to bound to KTL
- Example: Solaris 2 OS



Quiz

Advantages and inconveniences of ULT

L4.2

- **Advantages**
  - Thread switching does not involve the kernel: no mode switching
  - Scheduling can be application specific: choose the best algorithm.
  - ULTs can run on any OS. Only needs a thread library
- **Inconveniences**
  - Most system calls are blocking and the kernel blocks processes. So all threads within the process will be blocked
  - The kernel can only assign processes to processors. Two threads within the same process cannot run simultaneously on two processors

Advantages and inconveniences of KLT

- **Advantages**
  - the kernel can simultaneously schedule many threads of the same process on many processors
  - blocking is done on a thread level
  - kernel routines can be multithreaded
- **Inconveniences**
  - Thread switching within the same process involves the kernel. We have 2 mode switches per thread switch: user to kernel and kernel to user.
  - This results in a significant slow down

Thread Libraries

- A thread library provides the programmer with an API for
  - creating and destroying threads
  - passing messages and data between threads
  - scheduling thread execution
  - saving and restoring thread contexts
- Implementing a thread library
  - User-level library (with no kernel support) – the code and data structures for the library exist in the user space; invoking a function results in local function call in the user space
  - Kernel-level library – the code and data structures for the library exist in the kernel space; invoking a function results in a system call to the kernel
- Thread libraries: Pthreads, Win32, Java

Pthreads

- Posix Threads (pthreads) is a standardized programming interface for UNIX systems,
  - the interface is specified by the IEEE 1003.1c standard (1995)
  - this standard specifies behavior of the thread library
  - implementations which adhere to this standard are referred to as POSIX threads, or Pthreads
- Pthreads is:
  - Widely used threads package
  - defined as a set of C language programming types and procedure calls, implemented with a pthread.h header
  - Conforms to the Posix standard
  - Sample Calls: pthread_create(), pthread_exit(), pthread_join(), etc.
  - Typical used in C/C++ applications
- Examples of OS implementing Pthreads: Solaris 2, Linux, Mac OS X, True64 Unix and shareware implementation for Windows

Pthreads (Cont.)

- Multithreaded C program using Pthreads API

```
int sum;
#include <pthread.h>
void main(int argc, char *argv[]) {
    pthread_t tid; pthread_attr_t attr;
    …
    pthread_attr_init(&attr);   /*get default attributes
    pthread_create(&tid, &attr, runner, argv[1]);  /* create thread*/
    pthread_join(tid, NULL);   /* wait for thread to finish*/
    printf("sum = %d", sum)
}

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

Function to run in the thread created

$$sum = \sum_{i=0}^{N} i$$

Win32 Threads

- Similar to pthreads in several ways
- Kernel-level library available on Windows systems

```
#include <windows.h>
……..
void main(int argc, char *argv){
………
ThreadHandle = CreateThread( ……., Summation, …….)
if(ThreadHandle ! = NULL){
    WaitForSingleObject(ThreadHandle, INFINITE);
    CloseHandle(ThreadHandle);
    printf("sum = %d", Sum);
}
……..
```


Java Threads

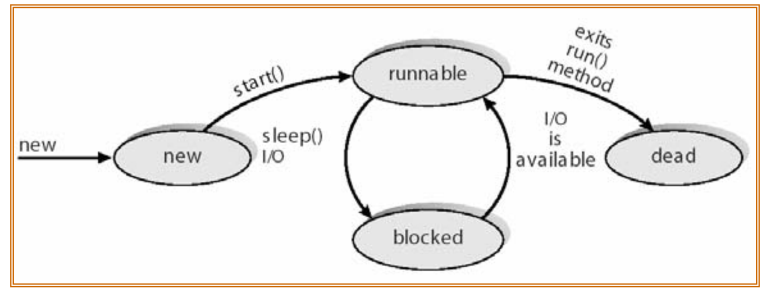
- Java language and its API provide a rich set of features for creating and managing threads
- As JVM is running on the top of host OS, Java thread API is typically implemented using a thread library available on the host system
  - Windows: Java Threads API uses Win32 API
  - Unix and Linux: Java Threads API uses Pthread
- Thread is a fundamental model of program execution in Java
  - All Java programs comprise of at least a single thread
- Java threads are managed by the JVM
- Two approaches to create threads in Java
  - Create a new class that extends Thread class and override the run() method
  - Define a class that implements Runnable interface (and define the run() method)
    - A thread is created by creating an instance of the Thread class and passing a Runnable object; then call the start() method to actually create the thread

Java Threads (Cont.)

```
class Sum{

}

class Summation implements Runnable{
    .....
    public void run(){

    }
}

public class Driver{
    public static void main(String[] args){
      ....

      Thread thd = new Thread(new Summation(..., ....);
      thd.start();     /* start the thread, here the thread is actually created */
      ....
      try{
          thd.join();            /* wait for the new thread to finish */
          ..........
      }catch(InterruptedException ie){}
    }
}
```

Define a class that implements Runnable and pass an instance of it to Thread class

Java Thread States



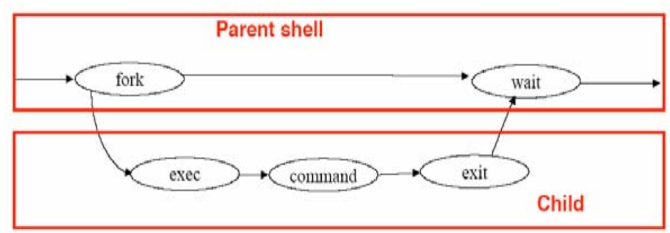
Outline

L4.3

- Threading Issues
  - Semantics of fork() and exec() system calls
  - Thread cancellation
  - Signal handling
  - Thread pools
  - Thread-specific data
  - Scheduler activations
- OS Examples

Review

- How Unix shell runs a program:



- when you type a command, the shell forks a clone of itself
- the child process makes an exec call, which causes it to stop executing the shell and start executing your command
- the parent process, still running the shell, waits for the child to terminate

Threading Issues: fork and exec system calls

- Semantics of fork and exec system calls change in multithreaded program
- When a thread (associated with process A) calls fork, two things can happen:
  - The new process duplicates all threads associated with process A
  - The new process will be single-threaded
- Some Unix operating systems support these two versions of fork
- The exec system call is used after a fork system call and typically work as described in a single-thread program
  - It replaces the entire process (including all threads) with the program specified in the parameter to exec
  - It loads a binary file into memory
  - It destroys the memory image containing the exec system call
  - It starts its execution

Threading Issues – Cancellation

- Thread cancellation is the task of terminating a thread before it is completed
  - For example: assume that multiple threads are searching a database. As soon as one thread returns the search result, we can terminate the remaining threads
- A thread to be cancelled is referred to as a target thread
- Thread cancellation can happen in two ways:
  - Asynchronous cancellation: One thread immediately terminates the target thread
  - Deferred cancellation: the target thread can periodically checks whether it should terminate; allows termination to happen in an orderly manner

Threading Issues – Cancellation (cont.)

- Thread cancellation is not as easy as it appears
  - What about resources allocated to a canceled thread
  - A thread might be cancelled while in the middle of updating a shared variable
- Becomes especially troublesome with asynchronous cancellation
- An OS usually reclaim system resources from a cancelled thread. But often does not reclaim all resources. Why?
- Deferred cancellation provides safer cancellation
  - A thread check whether it should be canceled at points when it can be cancelled safely (cancellation points)
  - The Pthreads API provides cancellation points

Threading Issues – Signal Handling

- A signal is used in Unix to notify a process that a particular event has occurred
  - illegal memory access, division be zero, terminating a process with Ctrl-C, time expire
- All signals follow the same pattern
  - A signal is generated by the occurrence of a particular event
  - A generated signal is delivered to a process
  - Once delivered, the signal must be dealt with
- A signal might occur synchronously and asynchronously

Threading Issues – Signal Handling (cont.)

- Synchronous signals:
  - Generated by events internal to the running process
  - Synchronous signals are delivered to the same process that performed the operation causing the signal
  - Examples include illegal memory access, division be zero, etc
- Asynchronous signals:
  - Generated by events external to the running process
  - Examples include terminating a process by specific keystrokes (e.g., control c), time expire
  - Asynchronous signals are more complicated

Threading Issues – Signal Handling (cont.)

- Every signal, whether synchronous or asynchronous, is handled in two ways
  - A default signal handler
  - A user-defined signal handler
- By default, every signal has a default signal handler that is run by the kernel
- This default signal handler can be overwritten by the user-defined signal handler
- Single-threaded programs: straightforward, signals are always delivered to the process
- Multithreaded programs: more complicated

Threading Issues – Signal Handling (cont.)

- When a signal is delivered to a multithreaded program, the following can happen:
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread
  - Deliver the signal to certain threads in the process
  - Deliver the signal to a specific thread
- Examples:
  - A terminating signal should be sent to all thread in the process
  - Solaris 2 implements the fourth option (i.e., creates a special thread within each process solely for signal handling)

Threading Issues – Thread Pools

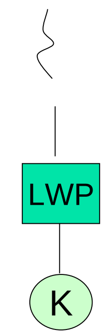
- Creating threads can be time consuming
- Too many threads can bog down the system
- Thread pools help with this problem
- Threads are pre-allocated
- The number of threads available at a given time is fixed
- Some systems may adjust the thread pool size depending on usage

Threading Issues – Thread Specific Data

- Threads belonging to the same process share the process data; benefit multithreaded programs
- However, in some instances, each thread might need its own copy of data.
  - For example, a transaction processing multithreaded application might service each transaction in a separate thread
- Most thread libraries such as Win32, Pthreads, and Java provide support for thread-specific data

Threading Issues – Scheduler Activations

- Both many-to-many and two-level models require communication between the kernel and thread library
- Such coordination allows the appropriate number of kernel threads to be dynamically adjusted to ensure best performance
- Scheduler activation
  - The kernel provides an application with a set of virtual processors (LWPs)
  - The application can schedule user threads onto an available LWP
- Scheduler activations provide **upcalls** –
  - a communication mechanism from the kernel to the thread library
  - Upcall handler must run on a virtual processor



OS Examples: Windows XP Threads

- Win32 API is the primary API for the family of Microsoft OSs (Windows 95, 98, NT, 2000, XP)
- Windows XP application runs as a separate process that may contain one or more threads
- Windows XP uses one-to-one model to map each ULT to an associated KLT
- Each thread contains
  - Thread ID
  - Register set (representing the status of the processor)
  - Separate user and kernel stacks
  - Private data storage area
- The thread context: register set, stacks, private storage area

OS Examples: Linux Threads

- Linux provides fork() system call for duplicating a process
- Linux does not distinguish between a process and a thread
  - Linux uses concept of task rather than thread or process.
- Clone() system call for creating threads and processes
  - Which resources are shared is controlled by a set of flags passed to the system call
  - Using clone is equivalent to creating thread
    - If parent and child tasks share file system information, memory space, signal handlers, set of open files
  - Using clone is equivalent to creating a process using fork()
    - If none is shared

End of Chapter 4

Operating System Concepts, 7th Ed. A. Siblerschatz, P. Galvin, and G. Gagne. Addison Wesley, 2005