Operating System Concepts

Che-Wei Chang
chewei@mail.cgu.edu.tw
Department of Computer Science and Information Engineering, Chang Gung University

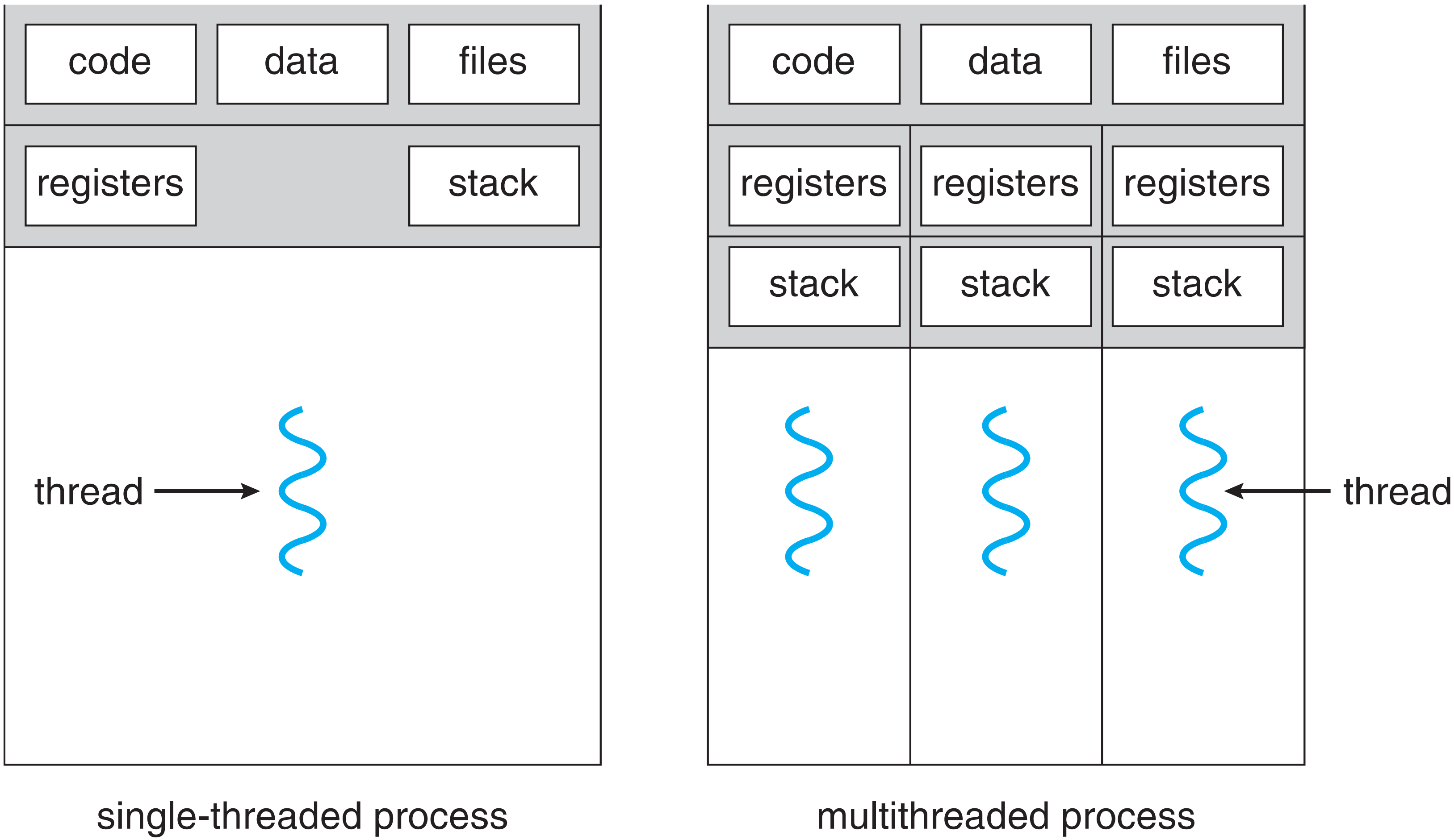
Contents

Chapter 4. Multithreaded Programming

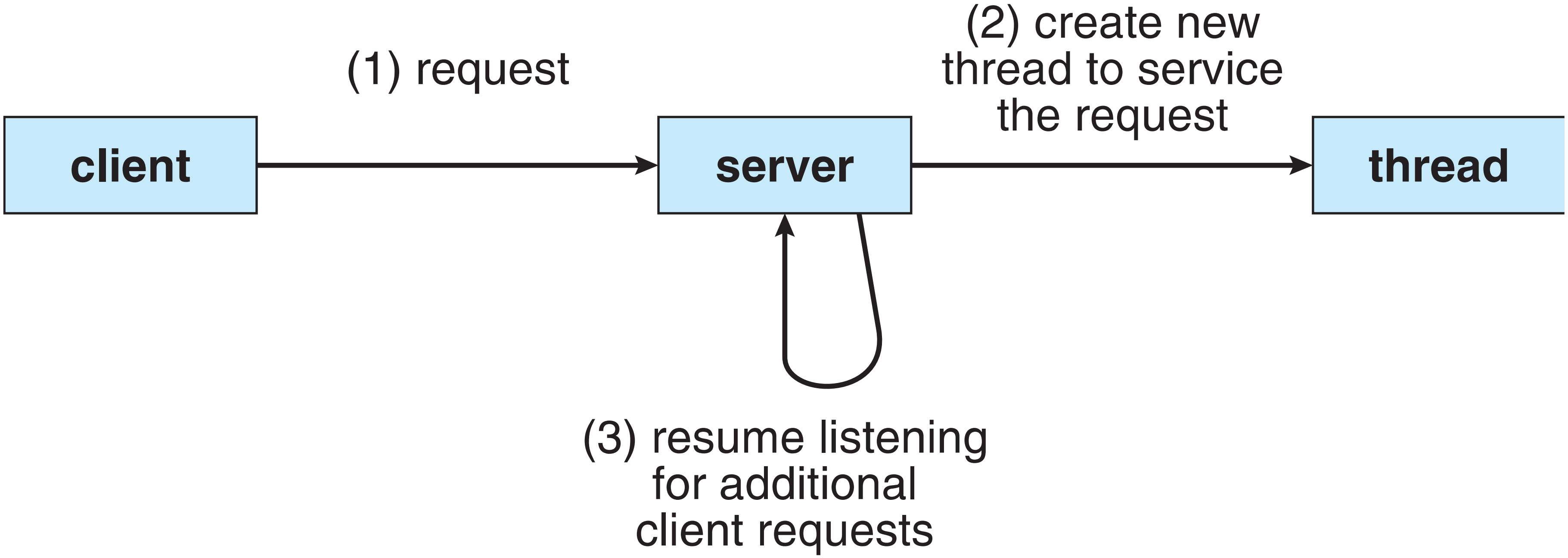
Objectives

- To introduce the notion of a thread
- To discuss the APIs for the Pthreads, Windows, and Java thread libraries
- To explore several strategies that provide implicit threading
- To examine issues related to multithreaded programming

Single and Multithreaded Processes



Multithreaded Server Architecture



Motivation

- Most modern applications are multithreaded
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
- Process creation is heavy-weight while thread creation is light-weight
- Kernels are generally multithreaded

Benefits

- Responsiveness
  - It allows a program to continue running even if part of it is blocked or is performing a lengthy operation
- Resource Sharing
  - Threads share resources of process, easier than shared memory or message passing
- Economy
  - Thread creation is cheaper than process creation
  - Thread switching overhead is lower than context switching
- Scalability
  - Threads can efficiently use multiprocessor architectures

Multicore Programming

- Motivation: the popularity of multiple computing cores per system
  - Multithreaded Programming
- Challenges in Programming
  - Dividing Activities
  - Load Balancing
  - Data Splitting
  - Data Dependency
  - Testing and Debugging

User Threads and Kernel Threads

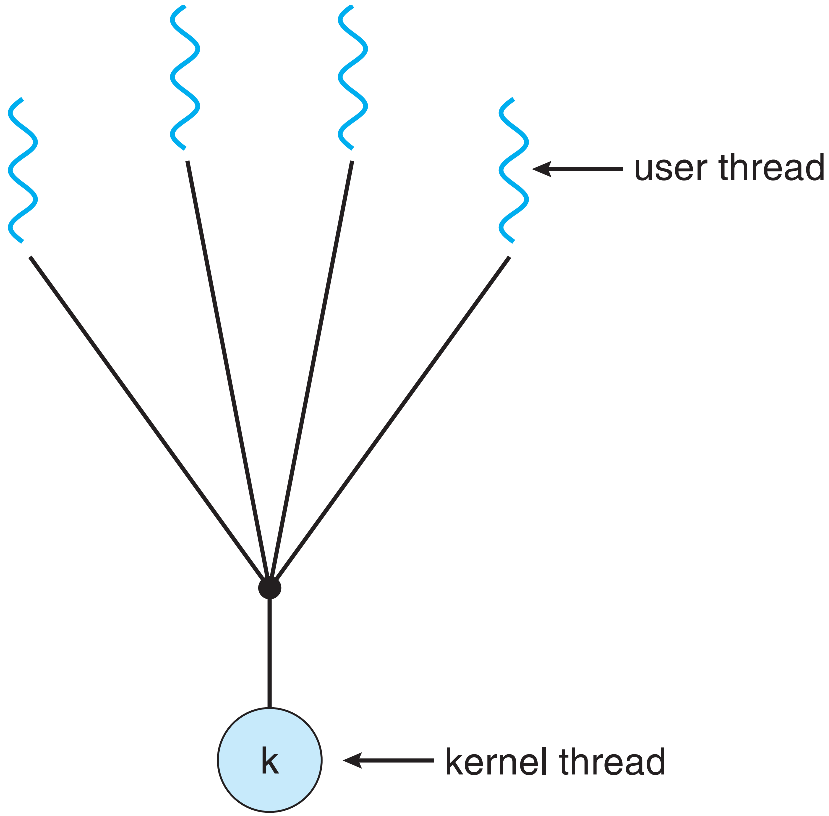
- User threads
  - Management done by user-level thread library
  - Three primary thread libraries:
    - POSIX Pthreads
    - Win32 threads
    - Java threads
- Kernel threads
  - Supported by the Kernel
  - Examples – virtually all general purpose operating systems, including:
    - Windows, Solaris, Linux, Tru64 UNIX, Mac OS X

Multithreading Models

- Relationship between user threads and kernel threads
  - Many-to-One
  - One-to-One
  - Many-to-Many

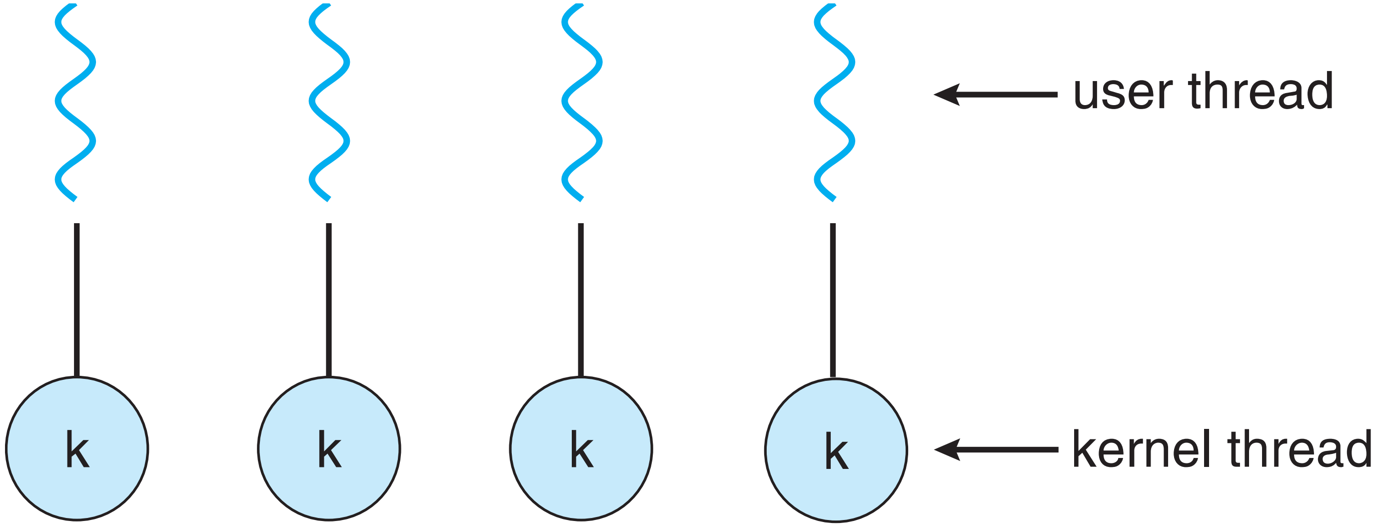
Many-to-One Model

- Many user threads to one kernel thread
- Advantage:
  - Efficiency
- Disadvantage:
  - One blocking system call blocks all
  - No parallelism for multiple processors
- Example:
  - Solaris Green Threads
  - GNU Portable Threads



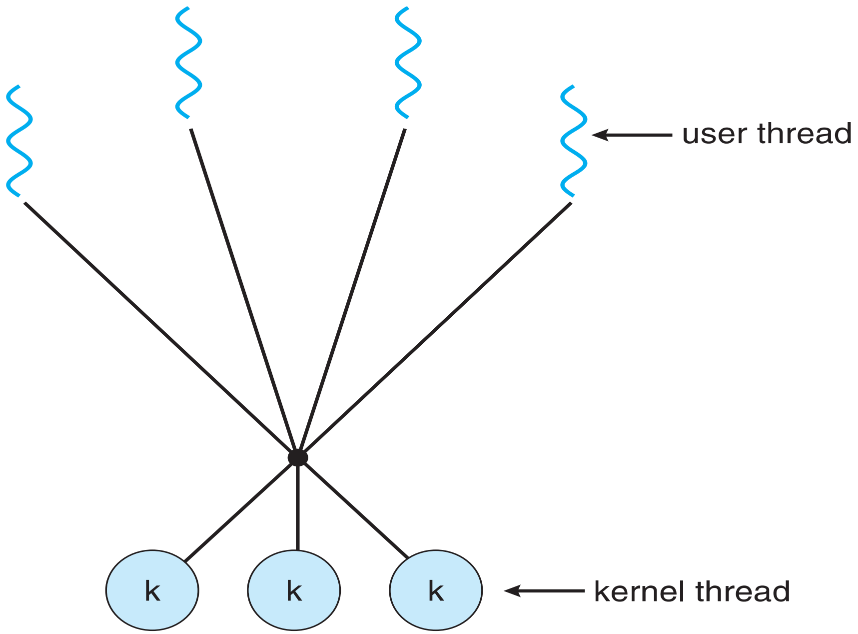
One-to-One Model

- One user-level thread to one kernel thread
- Advantage: One system call blocks one thread
- Disadvantage: Overheads in creating a kernel thread
- Example: Windows NT/2000/XP, Linux, Solaris 9



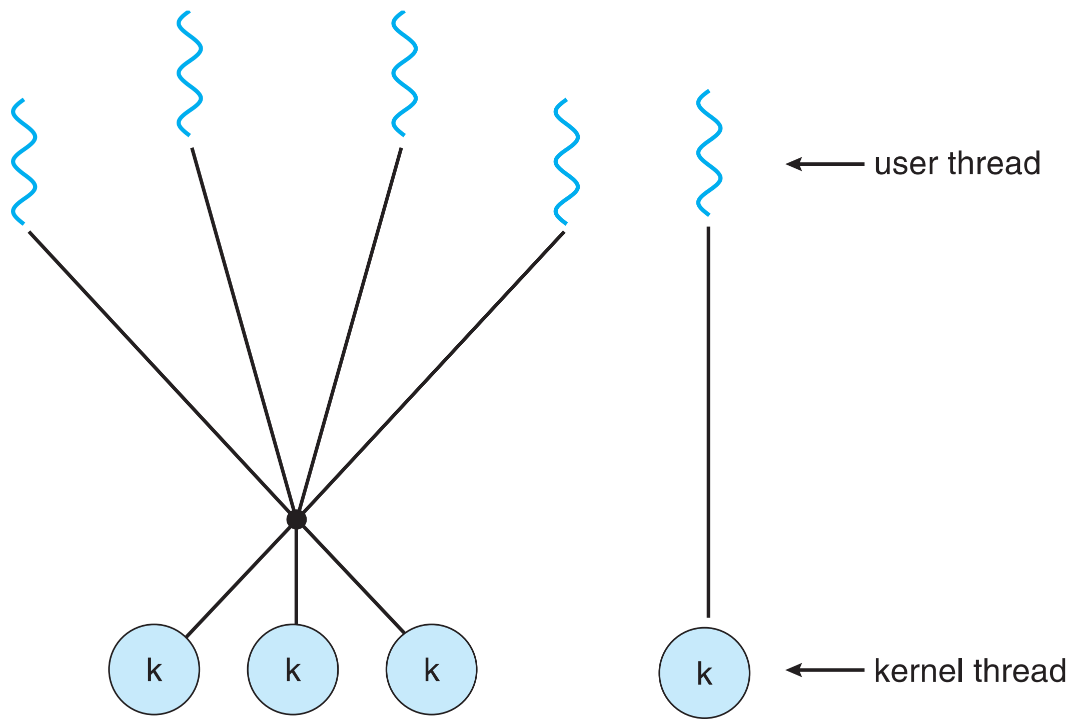
Many-to-Many Model

- Many-to-Many Model
  - Many user-level threads to many kernel threads
  - Advantage: A combination of parallelism and efficiency
  - Example: Solaris prior to version 9



Two-Level Model

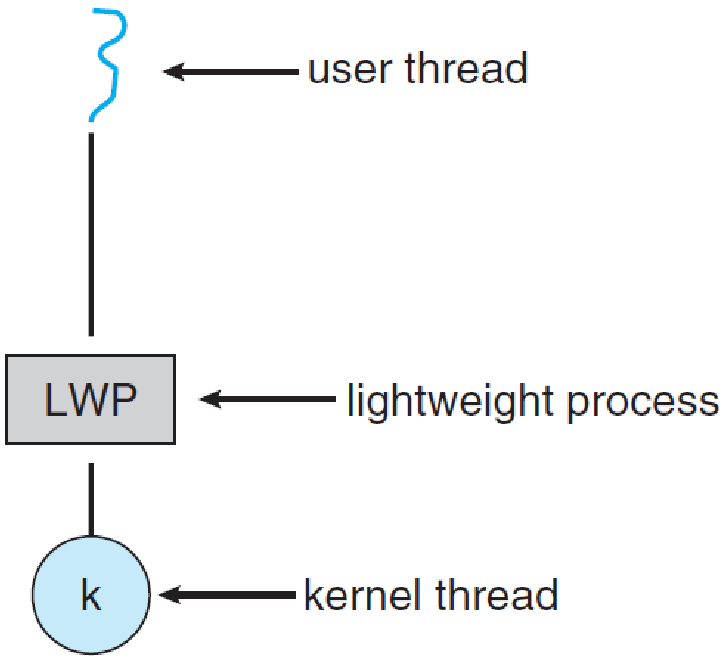
- Similar to the many-to-many model, except that it allows a user thread to be bound to a kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



Department of Computer Science and Information Engineering, Chang Gung University

Scheduler Activations

- Definition: A scheme for the communication between the user-thread library and the kernel
  - The kernel provides a set of virtual processors, i.e., light weight processes (LWP)
  - User threads on a LWP are blocked if any of the user threads is blocked!



Thread Libraries

- The goal thread libraries is to provide an API for creating and managing threads
- Two Approaches
  - User Thread Library
  - Kernel-Level Thread Library
- Well-Known Examples
  - POSIX Pthread – User or Kernel Level
  - Win32 thread – Kernel Level
  - Java thread – Level Depending on the Thread Library on the Host System

A Pthread Example (1/3)

- The specification of the example program
  - Read an input integer N
  - Create a thread to calculate the summation from 1 to N
  - Wait for the completion of the thread
  - Print the result from the thread
- Now, let’s use the Pthread library to implement the program

A Pthread Example (2/3)

```
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
  pthread_t tid; /* the thread identifier */
  pthread_attr_t attr; /* set of thread attributes */

  if (argc != 2) {
    fprintf(stderr,"usage: a.out <integer value>\n");
    return -1;
  }
  if (atoi(argv[1]) < 0) {
    fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
    return -1;
  }
```

Include the header file of pthread

Declare the function to be executed by the thread

Create the data-structure to be used by the thread

A Pthread Example (3/3)

```
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* create the thread */
    pthread_create(&tid,&attr,runner,argv[1]);
    /* wait for the thread to exit */
    pthread_join(tid,NULL);

    printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```

Initialize the data-structure to be used by the thread

Create the thread

Wait for the completion of the thread

Define the function to be executed by the thread

Compiling POSIX-Thread Programs

Compiler / Platform	Compiler Command	Description
INTEL Linux	**icc -pthread**	C
	icpc -pthread	C++
PGI Linux	**pgcc -lpthread**	C
	pgCC -lpthread	C++
GNU Linux, Blue Gene	**gcc -lpthread**	GNU C
	g++ -lpthread	GNU C++
IBM Blue Gene	**bgxlc_r / bgcc_r**	C (ANSI / non-ANSI)
	bgxlC_r, bgxlc++_r	C++

Implicit Threading

- Implicit threading is growing in popularity as numbers of threads increase
- Program correctness is more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Examples
  - OpenMP on Linux, Windows and Mac OS X
  - Grand Central Dispatch on Mac OS X

Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Signal handling
  - Synchronous and asynchronous
- Thread cancellation of target thread
  - Asynchronous or deferred
- Thread-local storage
- Scheduler activations

Fork and Exec System Calls

- When a process consists of multiple threads, does **`fork()`** duplicate only the calling thread or all threads?
  - Some UNIX systems have two versions of fork()
- **`exec()`** usually works as normal– replaces the running process including all threads

Signal Handling

- Two Types of Signals
  - Synchronous signal– should be delivered to the same process that performed the operation causing the signal
    - e.g., illegal memory access or division by zero
  - Asynchronous signal– can happen at any time point
    - e.g., ^C or timer expiration
- Delivery of a Signal
  - To the thread to which the signal applies
    - e.g., division-by-zero
  - To every threads in the process
    - e.g., ^C
  - To certain threads in the process
  - Assign a specific thread to receive all signals for the process

Thread Cancellation

- A cancellation signal is sent to the target thread
- Two scenarios for the cancellation:
  - Asynchronous cancellation
    - Immediate cancel the thread
  - Deferred cancellation
    - Wait until some special point of the thread, e.g., cancellation points in Pthread
- Difficulty
  - Resources have been allocated to a cancelled thread
  - A thread is cancelled while it is updating data

Thread-Local Storage

- Thread-local storage (TLS) allows each thread to have its own copy of data
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations
- Similar to **`static`** data
  - TLS is unique to each thread

Windows Threads

- Windows implements the Windows API– primary API for Win 98, Win NT, Win 2000, Win XP, Win 7, Win 8, and Win 10
- It implements the one-to-one mapping
- Each thread contains
  - A thread id
  - Register set representing state of processor
  - Separate user and kernel stacks for when thread runs in user mode or kernel mode
  - Private data storage area used by run-time libraries and dynamic link libraries (DLLs)
- The register set, stacks, and private storage area are known as the **context** of a thread

Linux Threads

- The concepts of threads was introduced in version 2.2
- In Linux
  - Processes and threads are called tasks
  - Any task has a PID (process identifier)
  - If two tasks do not share any data-structure, they are two processes
  - If two tasks share some data-structure, they just like two threads in the same process
  - fork() is used to create a new process
  - clone() is used to create a new thread
    - Flag setting in clone() invocation: CLONE_FS, CLONE_VM, CLONE_SIGHAND, CLONE_FILES