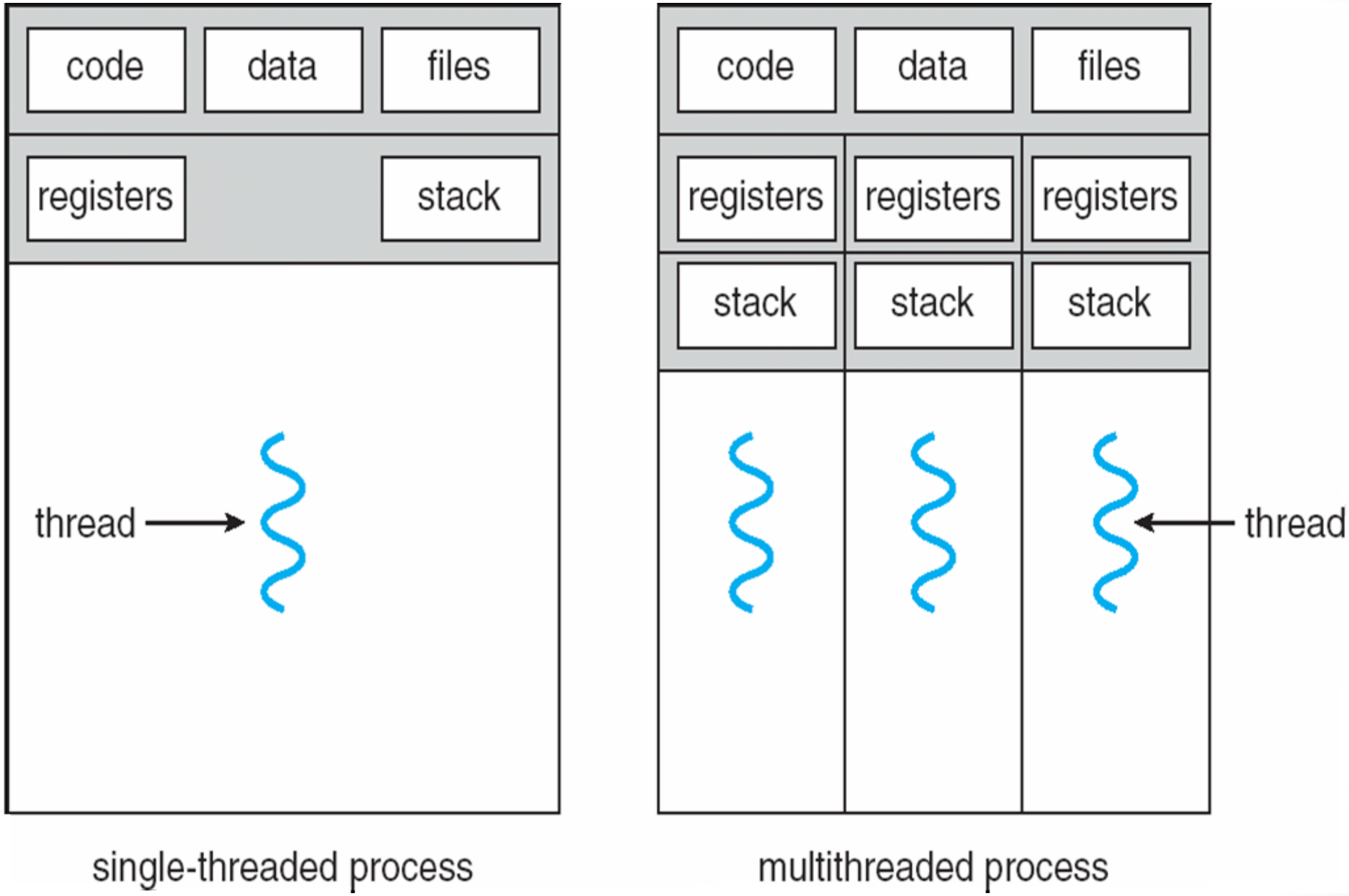
Chapter 4: Threads

adapted from Silberschatz, Galvin, Gagne

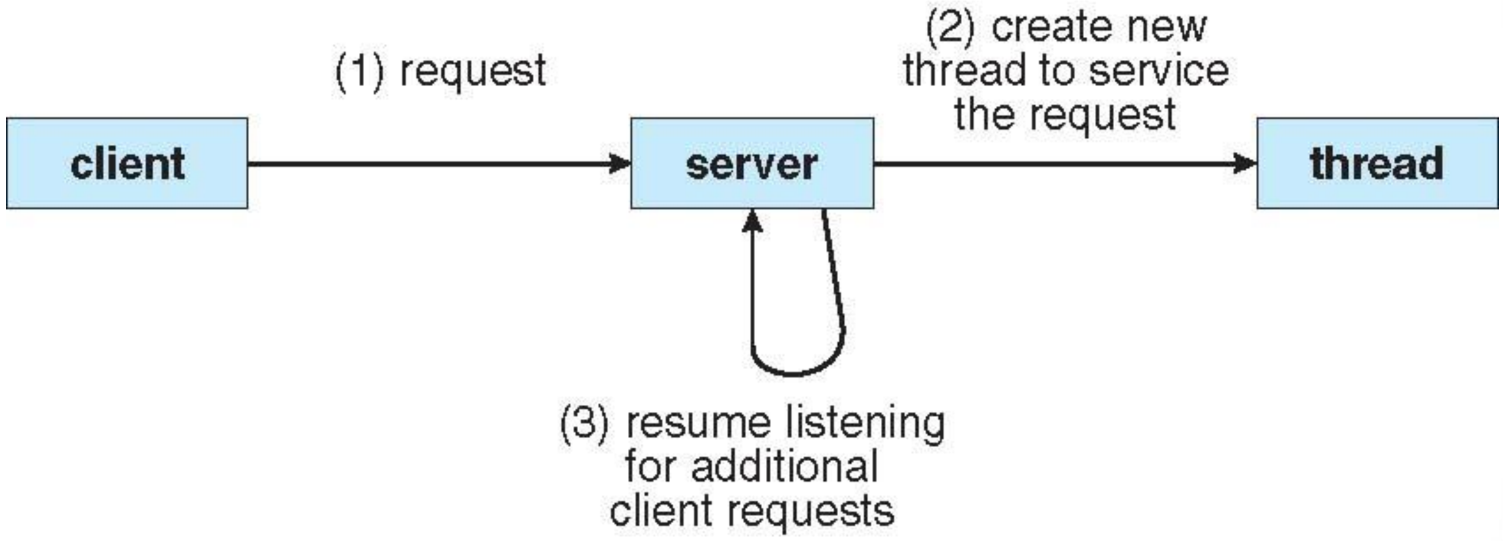
Chapter 4: Threads

- Overview
- Multithreading Models
- Thread Libraries
- Threading Issues

Single and Multithreaded Processes



Multithreaded Server Architecture



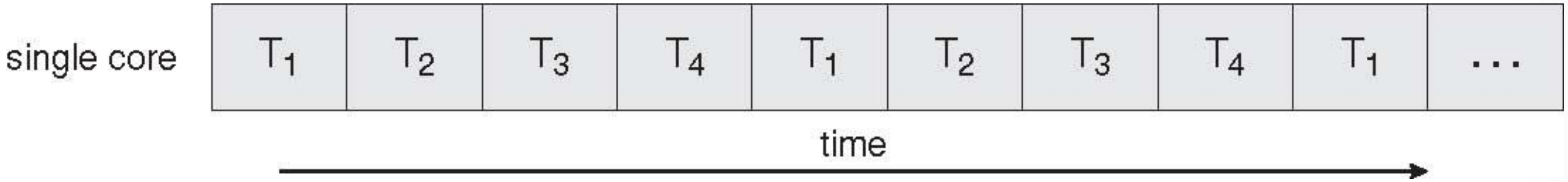
Benefits

- Responsiveness
  - Single-threaded application will be unresponsive to user until a lengthy operations is completed
- Resource Sharing
  - Threads share memory and resources
- Economy
  - Thread management is less time consuming than process management
- Scalability
  - Multi-threaded process run on multi-core CPU

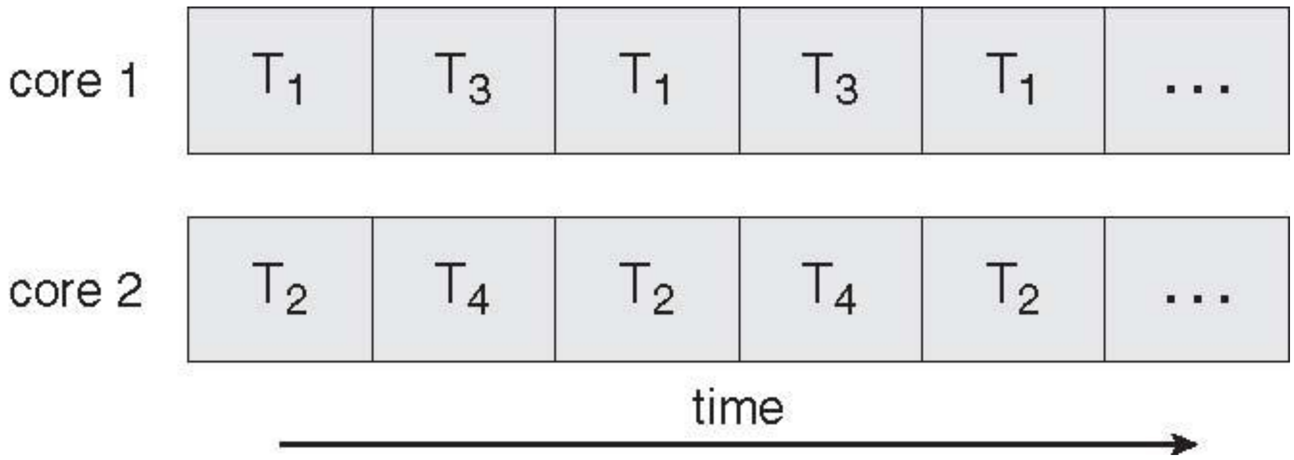
Multicore Programming

- Multicore systems putting pressure on programmers, challenges include
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**

Concurrent Execution on a Single-core System



Parallel Execution on a Multicore System

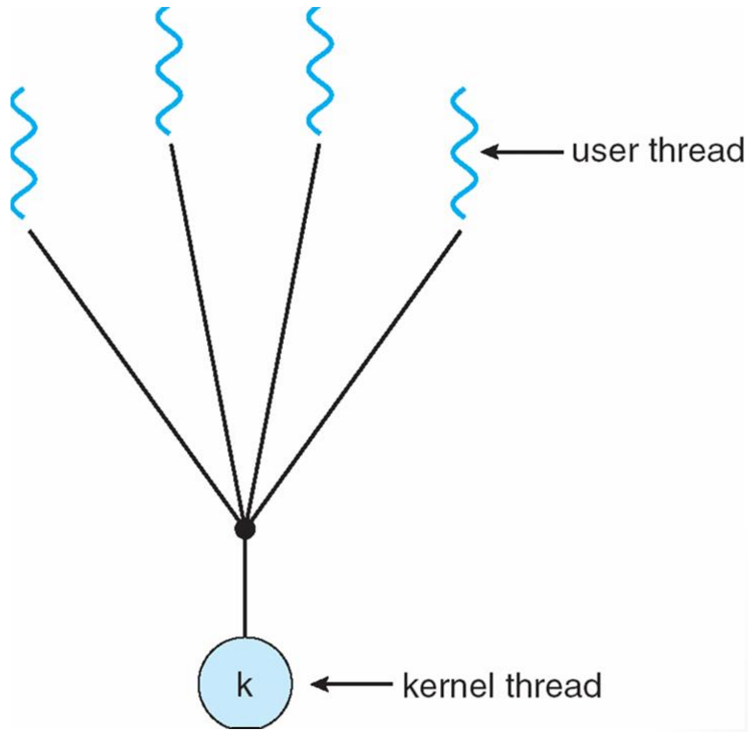


Multi-Threading Model

- User Thread
  - Thread management done by user-level threads library
  - Example: POSIX thread
- Kernel Thread
  - Supported by the Kernel
  - Examples: Windows, Linux, Mac OS X, etc.
- Multi-threading models
  - Many-to-One
  - One-to-One
  - Many-to-Many

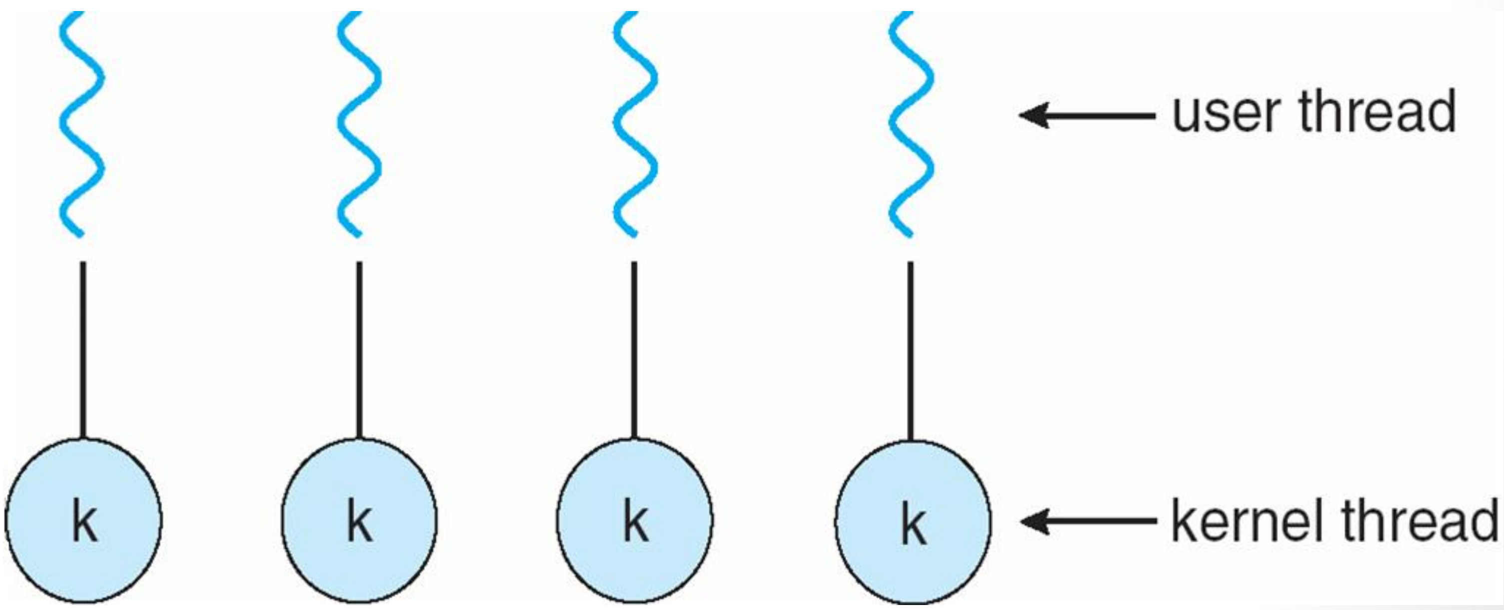
Many-to-One Model

- Many user-level threads mapped to single kernel thread
- Discontinued because of inability to use multi-core
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads



One-to-one Model

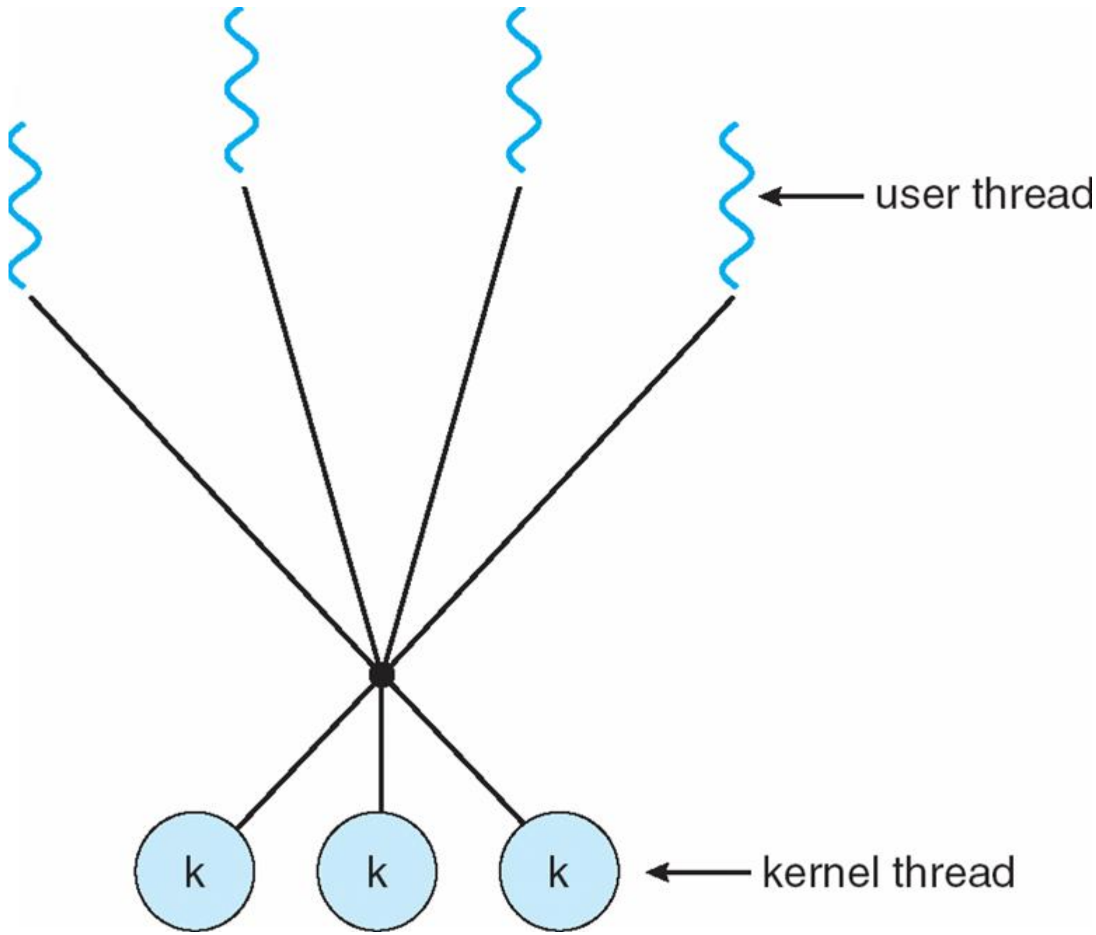
- Each user-level thread maps to kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later



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



Thread Libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS
- Example: POSIX thread - Pthread

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard API for thread creation and synchronization
  - The Portable Operating System Interface (POSIX) is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems.
  - POSIX defines the application programming interface (API), along with command line shells and utility interfaces, for software compatibility with variants of Unix and other operating systems.
- 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)

Pthread Programming

```
/* thread.c */
#include <pthread.h>
#include <stdio.h>

int sum = 0; //shared by all threads
void *runner(void *param) {
	int i, upper = atoi(param);
	for (i=1;i<=upper;i++)
		sum += i;
	pthread_exit(0);
}
int main() {
	pthread_t tid; // thread identifier
	pthread_attr_t attr; //set of thread attributes
	pthread_attr_init(&attr); //get the default attributes
	/* create the thread */
	pthread_create(&tid, &attr, runner, "10");
	/* wait for the thread to exit */
	pthread_join(tid, NULL);
	printf("sum = %d\n", sum);
}

/* gcc –o test thread.c –lpthread */
```

Pthread Programming (Cont)

```
/* thread2.c */
#include <pthread.h>
#include <stdio.h>

int sum = 0; //shared by all threads

int main() {
        int i;
        pthread_t tid[10]; // thread identifier
        pthread_attr_t attr; //set of thread attributes
        pthread_attr_init(&attr); //get the default attributes
        /* create the thread */
        for(i=0; i < 10; i++)
                pthread_create(&tid[i], &attr, runner, "10");
        /* wait for the thread to exit */
        for(i=0; i < 10; i++)
                pthread_join(tid[i], NULL);
        printf("sum = %d\n", sum);
}
/* gcc –o test2 thread2.c –lpthread */
```

Threading Issues

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

Semantics of fork() and exec()

- Does **fork()** duplicate only the calling thread or all threads?
- Some UNIX systems have chosen to have two versions of fork()
  - One version duplicates all threads
  - One version duplicates only the calling thread
- exec() semantics
  - Replace the entire process with the new program
- Which fork() to use is application dependent
  - If exec() is to be called: duplicate only the calling thread
  - If exec() is not to be called: duplicate all thread

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
- Challenges:
  - One thread is canceled while updating shared data
    - Asynchronous cancellation may cause problems
  - Resources have been allocated to a canceled thread
    - OS will reclaim system resource from a canceled thread but not all resources
    - System-wide resource may not be freed in asynchronous cancellation

Signal Handling

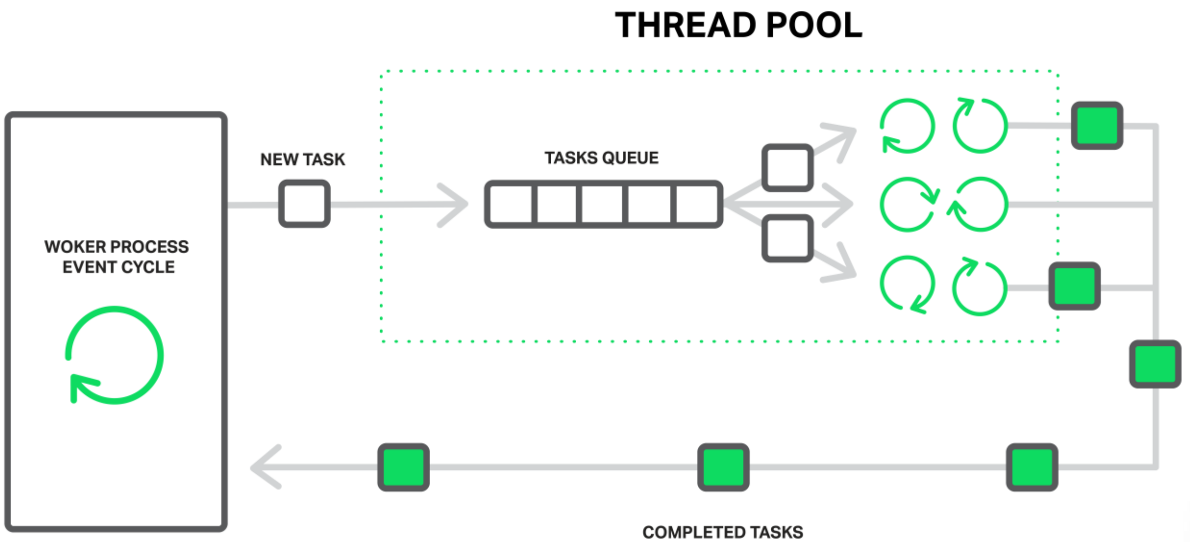
- Signals are used in UNIX systems to notify a process that a particular event has occurred
- A signal handler is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled
- A signal can be handled by
  - A default signal handler
  - A user-defined signal handler

Signal Handling (Cont)

- Synchronous signals and asynchronous signals
  - Synchronous signals: e.g., divided by 0
  - Asynchronous signals: e.g., <control><C>
- Handling signals in multi-threaded programs:
  - 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
- Signal handling:
  - Synchronous signals: deliver to the thread causing the signal
  - Asynchronous signals: tricky
    - <control><C> should be delivered to all

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 Local Storage (TLS)

- Data is shared among threads by default
- TLS allows each thread to have its own copy of data
- Example
  - Thread identifier

End of Chapter 4