Programming Threads in C
CS 360 Internet Programming

Daniel Zappala

Brigham Young University
Computer Science Department
Process and Thread Models

Process vs. Thread

- **Process**: unit of resource ownership
  - Process is allocated a virtual address space: process image
  - Assigned main memory, I/O devices, files
  - Protection from other processes

- **Thread**: unit of dispatch/execution
  - Instruction trace through a program
  - Thread execution state
  - Context: stack, storage for local variables
  - Access to memory and resources of the process – shared with all other threads for the same process
Benefits of Threads

- faster to create a new thread than a process
- faster to terminate a thread than a process
- faster to switch between two threads within the same process
- more efficient communication between threads
  - process communication requires protection and communication provided by kernel
  - threads can avoid the kernel
- parallel processing
Using Threads

- foreground and background work
  - a thread for the GUI and another to execute tasks
  - faster response time for the user

- networking communication
  - separate threads to handle blocking system calls so multiple connections can be handled simultaneously

- modular program structure
  - give logically separate functionality to different threads
  - easily program with blocking system calls
Thread Support in Operating Systems

- MS-DOS: one process, one thread
- old Windows, UNIX: multiple user processes, but only one thread per process
- JVM: one process, multiple threads
- modern operating systems (Linux, Windows 2000+, Solaris, Mach): multiple threads per process
User-Level and Kernel-Level Threads

(a) Pure user-level
(b) Pure kernel-level
(c) Combined

User-Level Threads

- all thread management done by the application
  - creating and destroying threads
  - thread communication
  - thread synchronization
  - thread scheduling
- runs in a single process, no kernel involvement
- advantages
  - efficient: no kernel mode switch to handle a different thread
  - application-specific scheduling
  - O/S independent
- disadvantages
  - thread system call blocks entire process
  - no multiprocessing: threads of the same process cannot run on different processors
Kernel-Level Threads

- thread management handled by kernel
- kernel schedules threads, not processes

Advantages
- multiprocessing support
- blocked thread doesn’t block entire process
- kernel can be multithreaded

Disadvantages
- thread switching more expensive: requires mode switch
Introduction

- **Pthreads**: POSIX threads library
  - POSIX is the collective name of a family of related standards specified by the IEEE to define the application programming interface (API) for software compatible with variants of the Unix OS. – Wikipedia
  - threads standardized in 1995

- **Linux**
  - 1:1 mapping to kernel level threads
  - compile application with `gcc/g++ -pthread`
  - Native POSIX Thread Library (NPTL): In tests, NPTL succeeded in running 100,000 threads simultaneously on a IA-32 which were started in two seconds. In comparison, this test under a kernel without NPTL would have taken around 15 minutes. – Wikipedia

Creating a Thread

- when a program starts, it runs in a single thread called the main thread
- create threads with \texttt{pthread\_create()}

```c
#include <pthread.h>

int pthread_create(pthread_t *thread, pthread_attr_t *attr,
void *(*start_routine)(void *), void *arg);
```

- the thread identifier is returned through the thread pointer
- the new thread runs the given start routine with the given arguments, terminates by finishing this routine
- attributes include priority, stack size, etc. - leave as default by passing a null pointer
- return value is normally zero, return positive error value otherwise
Joining a Thread

- wait for threads to terminate with `pthread_join()`

```c
#include <pthread.h>

int pthread_join(pthread_t th, void **thread_return);
```

- specify thread identifier of thread to wait for
- return value of thread in given in returned pointer if non-null
- must call join to reclaim thread memory and thus avoid memory leaks
Getting a Thread ID

- get your own thread ID with `pthread_self()`

```c
#include <pthread.h>

pthread_t pthread_self(void);
```

- returns the thread’s thread identifier
Exiting a Thread

- exit a thread with `pthread_exit()`

```c
#include <pthread.h>

void pthread_exit(void *retval);
```

- returned value can be any object that is not local to the thread
// initialize data to pass to thread 1
data1.number = 1;
strcpy(data1.message, "Hello!");

// initialize data to pass to thread 2
data2.number = 2;
strcpy(data2.message, "Goodbye!");

// create threads
pthread_create(&thread1, NULL, &print, (void *) &data1);
pthread_create(&thread2, NULL, &print, (void *) &data2);

- see threading example code on web site
Threaded Server

```c
while ((c = accept(s, (struct sockaddr *)&client, &clientlen)) > 0) {
    // create a thread to handle the client
    pthread_t tid;
    int *arg = new int;
    *arg = c;
    pthread_create(&tid, NULL, &worker, arg);
}
```

- be sure to allocate memory for arguments and free it in thread
- creating one thread per client is not scalable
- see threaded server example code on web site