CS 360 Internet Programming
Concurrent Programming
*Programming Threads in C*

Daniel Zappala
Computer Science
Brigham Young University
1. **Threads**
   - Processes vs Threads
   - Using Threads
   - Types of Threads

2. **Pthreads Library**
   - Introduction
   - Creating and Terminating Threads
   - A Threaded Web Server
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
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
Using Threads

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

- asynchronous processing
  - a thread that periodically backs up the word processor’s current file while another handles the user’s actions

- modular program structure
  - give logically separate functionality to different threads
  - easily program with blocking system calls
User-Level and Kernel-Level Threads

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

User-level thread  Kernel-level thread  Process

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
**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

**http://www.llnl.gov/computing/tutorials/pthreads/**
Creating a Thread

- when a program starts, it runs in a single thread called the main thread
- create threads with `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
1 #include <pthread.h>
2
3 void pthread_exit(void *retval);
```

- returned value can be any object that is not local to the thread
for(;;) {
    int item = numconnects % QUEUELEN;
    int len = sizeof(svr_info);
    int hSocket = accept(listen_socket,(sockaddr*)&svr_info,
                           (socklen_t*) &len);
    socketqueue[item] = hSocket;
    if (hSocket == -1) {
        printf("socket is invalid.\n");
        continue;
    }
}
pthread_t listenerThread;
int status = pthread_create(&listenerThread, NULL, &connectionHandler,
                           &socketqueue[item]);
numconnects++;
}
Connection Handler

```
void* connectionHandler(void* inargs) {
    int hSocket = *((int*)inargs);
    int hcSocket = hSocket;
    Webserver w(hcSocket);
    w.launchServer();
    shutdown(hcSocket, SHUT_RDWR);
}
```

- creates one thread per incoming connection
- race condition: threads could be created faster than they consume the socketqueue items
  - relies on thread starting fast enough to copy socket identifier to its local variable before it is overwritten
  - could allocate a new item and then ask thread to delete it
  - could use shared memory to synchronize access to the queue