Introduction to Threading

CST 357/457 – Systems Programming
Michael Ruth, Ph.D.
Associate Professor
Computer Science & I.T.
mruth@roosevelt.edu

Objectives

- Discuss thread concepts including terminology, benefits, and costs
- Explain threading models and patterns
- Discuss concurrency and parallelism
- Explain race conditions and synchronization using mutual exclusion
- Discuss deadlock and its avoidance
- Explain pthreads API including thread implementations and using the API

Programs, Processes, & Threads

- A ***binary*** is compiled, executable code that is dormant…
  – We colloquially use the term ***program***
- A ***process*** is a running program
  – Includes the binary, instance of virtualized memory, kernel resources, security context, and one or more threads
- A ***thread*** is the unit of activity inside a process
  – Has its own virtualized processor which includes a stack, processor state such as registers and an IP
  – Smallest unit of execution schedulable by an operating system's process scheduler

Introduction

- Each process contains one or more threads
  - If you only have one thread, there is only a single unit of execution in the process and only one thing going on at once
    - Single-threaded processes (what you're used to)
  - If you have more than one thread, then there is more than one thing going at once
    - Multi-threaded process

OS Abstractions

- Modern OS provide two fundamental virtualized abstractions to userspace:
  - Virtualized memory
    - Associated with each process
  - Virtualized processor
    - Associated with each thread
- Together they give the illusion that each process is the only running process

Multithreading Benefits

- Programming Abstraction
  - Natural approach to some problems
- Parallelism
  - Improves throughput
- Improving Responsiveness
  - Long running processes block UI events
- Blocking I/O
- Context Switching
  - T-T switching is cheaper than P-P switching
- Memory Savings
  - Shared memory

Multithreading Costs

- Programming with threads is considerably more difficult than without
  - The general problem?
    - Concurrency with shared memory
  - How do we solve?
    - Not just ***synchronization*** alone!
    - Your threading model and synchronization strategies MUST be a part of your design

Threading Models

- Approaches to implement threads depend on where the functionality is provided: user or kernel space
  - **1:1 threading or Kernel-level threading**
    - There is a 1 to 1 relationship between what the kernel provides and what is consumed
  - N:1 threading or user-level threading
    - User space is where threading is implementing
    - Requires little or NO support from kernel but requires significant user-space code…
    - Benefit is that context switching is cheaper…
  - N:M threading or hybrid threading
    - Kernel and user threads are used and mapped N to M

Threading Patterns

- The first step in developing a threading strategy is to consider threading patterns
  - There are a myriad of abstractions and implementation details to consider but there really are only two core patterns:
    - ***Thread-Per-Connection***
      - Unit of work is assigned to a thread and that thread is responsible only for that unit of work during its execution
    - ***Event-Driven***
      - Since so much of threading is simply waiting on IO, we're going to decouple that waiting from the threads
      - Instead, we'll issue all IO asynchronously and use multiplexed IO to manage the flow of control
      - After callback, the multiplexed IO event loops hands the callback to a waiting thread

Concurrency & Parallelism

- ***Concurrency*** is the ability of two or more threads to execute in overlapping time periods
  - Can occur without parallelism
  - Programming pattern (a way to approach problems)
- ***Parallelism*** is the ability to execute two or more threads simultaneously
  - Parallelism is a specific form of concurrency requiring hardware that allows for two threads to operate at the same time
  - Hardware feature achievable through concurrency

Race Conditions

- It is concurrency that introduces most of the ***pain*** having to do with threading
- Since threads can overlap their execution, it becomes ***nondeterministic***
  - Threads share resources so accessing memory is like a "race" where what happens depends on who ***gets there first***
  - Formally, a ***race condition*** is a situation in which the unsynchronized access of a shared resource by two or more threads leads to ***erroneous program behavior***
    - the shared resource can be anything: data, hardware, kernel resources, etc
    - Data is the most common form and called a data race

Race Conditions (cont)

- The window in which a race can occur is referred to as a ***critical region***
  - This is the area that needs ***synchronization***
- Races are eliminated by ***synchronizing threads*** access to critical regions

Critical Region Example

```
double withdraw(double amt) {
   if (balance > amt) {
       balance = balance - amt;
       return amt;
   } else {
       return -1;
   }
}
```


More Critical Region Examples

```
x++;

++x;

x = x + 4;
```


Synchronization

- The fundamental source of races is the critical regions are a window during which correct program behavior requires that threads ***do not interleave execution***
  - We must synchronize to ensure that each thread has ***mutually exclusive access*** to the critical region
  - We refer to operations as ***atomic*** if it is indivisible
    - To the rest of the program atomic operations appear to occur instantaneously

Mutexes

- The most common mechanism for synchronization is the lock
  - Since we are enforcing mutual exclusion, we often refer to these locks as ***mutexes***
  - ***NOTE:***
    - Nothing is magical about locks, nothing physically enforces the mutual exclusion
    - Locks are basically a ***gentleman's agreement***

Critical Region Example P2

```
double withdraw(double amt) {
   lock();
   if (balance > amt) {
       balance = balance – amt;
       unlock();
       return amt;
   } else {
       unlock();
       return -1;
   }
}
```


Lock Data, Not Code

- As we move further in, don't forget that race conditions are based on **data** and it's manipulation by many threads
- Although we lock critical regions of code, we need to ensure that we focus on locking the **data**
  - Doing so ensures that the data won't be manipulated somewhere without proper consideration for concurrency

Deadlocks

- The cruel irony of programming with threads is that:
  - we want concurrency, so we add threads, but then we get race conditions
  - We don't want race conditions, so we add mutexes, but then we get deadlocks
- ***Deadlocks*** are a situation in which two threads are waiting for the other to finish and thus neither does
  - Programs don't crash (or even appear to)
  - They just hang…

Deadlock Avoidance

- Avoiding deadlocks is important and the only consistent, safe means to do so is by designing locking into your threaded application ***carefully***
- For example, one particular kind of deadlock is the ***deadly embrace***
  - One thread acquires mutex A followed by mutex B and another thread acquires mutex B followed by mutex A
    - Under a timing scenario where they both acquire their first locks before their second locks, neither will ever acquire the second lock
  - Fixing this requires clear rules to assure that this never happens
    - In this case, make it so that all locks are acquired in the same order

POSIX Threads (pthreads)

- The Linux kernel only provides the underlying primitives that enable threading, the bulk of threading libraries is in user space
  - POSIX standardized a threading library and developers call this standard POSIX Threads or pthreads

Linux Threading Implementations

- Linux Threads
  - Original implementation providing 1:1 threading
  - Designed for a kernel with little support for threading (just clone())
  - Implements pthreads using existing UNIX interfaces
  - Scaled poorly and wasn't in compliance
- **Native POSIX Thread Library (NPTL)**
  - Provides 1:1 threading based around the clone() system call and the kernel's model that threads are just like processes except they share resources
  - Capitalizes on system calls new to newer kernels
  - Improves scalability and conformance issues of Linux Threads

Pthread API

- The pthread library is very large but contains everything you need to build a multithreaded application
  - Everything is in <pthread.h>
  - Every function begins with pthread_
- All functions can be broken into two main categories:
  - Thread management
  - Synchronization
- If we're going to use pthreads, we need to link it in since it is in a separate library
  - Automated with **–pthread** flag with **gcc**
    - **EX: gcc –o runme –pthread p1.c**

Creating Threads

- `#include <pthread.h>`
- `int pthread_create (pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);`

- Upon success, a new thread is created and begins executing the function provided by start_routine passed the sole argument arg
- The function will store the thread ID in the pthread_t if not NULL
- The pthread_attr_t is used to change the default behavior of newly created thread
  - Usually just NULL (we'll skip)
- start_routine must have the following signature:
  - `void * start_thread (void *arg)`

Thread IDs (TID)

- Thread ID is the thread analogue to PID
  - It does not have to be an arithmetic type
- Retrieving your TID:
  - `#include <pthread.h>`
  - `pthread_t pthread_self(void)`
- Comparing:
  - `#include <pthread.h>`
  - `int pthread_equal(pthread_t a, pthread_t b)`
  - If equal, returns a nonzero value

Terminating Threads

- Threads may terminate under several circumstances:
  - If a thread returns from start_routine
  - If a thread invokes pthread_exit()
  - If a thread is cancelled by another thread via the pthread_cancel() function
- All of the threads terminate in the following:
  - The process returns from the main function
  - The process terminates via exit
  - The process executes a new binary via exec

Terminating Threads (cont)

- Terminating yourself:
  - `#include <pthread.h>`
  - `void pthread_exit(void *retval)`
- Terminating others:
  - `#include <pthread.h>`
  - `int pthread_cancel(pthread_t thread)`

Cancellable?

- Whether and when a thread is cancellable depends on its state & type
  - Cancellable state:
    - Either enabled (default) or disabled (puts cancel requests on hold)
      - `#include <pthread.h>`
      - `int pthread_setcancelstate(int state, int *oldstate)`
  - Cancellable type:
    - Either asynchronous or deferred (default)
      - In asynchronous, thread can be killed at any point after cancellation request
      - In deferred, thread can only be killed at cancellation points which represent safe points in pthreads library (critical regions)
    - We can change it:
      - `#include <pthread.h>`
      - `int pthread_setcanceltype(int type, int *oldtype)`

Joining Threads

- ***Joining*** allows one thread to wait for another thread to finish (terminate)
  - `#include <pthread.h>`
  - `int pthread_join(pthread_t thread, void **retval)`
  - Upon call, the calling thread waits for the indicated thread
    - Once it terminates, the waiting thread is woken up and if retval is not null, provided the return from pthread_exit
      - We say that the threads are joined!

Detaching Threads

- By default, threads are created ***joinable,*** but we may have them ***detach***
  - Which renders them no longer joinable
- Since threads consume resources until joined, threads you do not intend to join should be detached
  - Good practice to explicitly detach or join all threads
- Actual call:
  - `#include <pthread.h>`
  - `int pthread_detach(pthread_t thread)`

Pthread mutexes

- For all their power, they are relatively easy to use:
  - Initializing mutexes:
    - `pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`
  - Locking Mutexes:
    - `#include <pthread.h>`
    - `int pthread_mutex_lock(pthread_mutex_t *mutex)`
  - Unlocking Mutexes:
    - `#include <pthread.h>`
    - `int pthread_mutex_unlock(pthread_mutex_t *mutex)`

Summary

- Discussed thread concepts including terminology, benefits, and costs
- Explained threading models and patterns
- Discussed concurrency and parallelism
- Explained race conditions and synchronization using mutual exclusion
- Discussed deadlock and its avoidance
- Explained pthreads API including thread implementations and using the API

Questions?