Threads and Mutexes

UNX511 Week 11 Class 2

John Sellens

July 22, 2025

Seneca Polytechnic

Outline

POSIX Threads

POSIX Threads

- A thread is a separate execution path in a process
    - Which can take advantage of multiple cores/processors
- Provides a way to have multiple tasks in one program, sharing data
    - Without needed to write and manage a scheduling loop
- We have seen threads used for reading from a message queue etc
    - e.g. the week 10 examples
- Threads share heap memory (basically globals) but not stack memory (basically function local variables)
- Good overview from Backblaze: What's the Diff: Programs, Processes, and Threads
- Recall that POSIX is the international UNIX standard

Threads Overview

- Create a thread by calling `pthread_create(3)`
    - Pass it the name of a function and an argument
    - Thread starts running that function
- Thread continues until it returns from the function, or calls `pthread_exit(3)`
- Some other thread (often the original thread) calls `pthread_join(3)` to join (merge) the exiting thread with this one (sort of like `wait(2)`)
    - Gets return value from function, or `pthread_exit()` argument
- Also `pthread_cancel(3)` (ask thread to exit), `pthread_detach(3)` (no need to re-join)
- Review sample code: `week11_1/1_threads`

POSIX Mutexes

POSIX Mutexes

- mutex – short for Mutual Exclusion
- A locking mechanism within threaded code
    - Most commonly used to isolate updates/uses of shared variables
    - Turns a block of code into (effectively) an atomic operation
    - i.e. Nothing else can make similar changes at the same time
- It’s a cooperative / voluntary mechanism
    - Nothing will prevent changing global variables if you don’t use a mutex
    - But it’s within a single program, so you can set rules for yourself
- There can be multiple mutexes in a program
    - Typically use global `pthread_mutex_t` variables
- Review sample code: `week11_1/2_mutex`

Mutexes Overview

```
// global variable
pthread_mutex_t lock_x;
// typically in original thread
pthread_mutex_init( &lock_x, NULL );
// in cooperating threads
pthread_mutex_lock( &lock_x );      // acquire lock
pthread_mutex_unlock( &lock_x);     // release lock
// typically in original thread
pthread_mutex_destroy( &lock_x );
```

Deadlocks

- Any time you’ve got multiple processes/threads using locks, there could be a possibility of “deadlock”
  - When two things (locks, resources) are needed at the same time and you don’t get both
  - e.g. Thread 1 gets lock 1, thread 2 gets lock 2 and then tries to get lock 1, and thread 1 tries to get lock 2
- Deadlock can also happen in multi-step database updates
- In our simple examples, deadlock is unlikely (impossible?)
- Simply having to wait for another thread to release a lock is not deadlock
  - Unless the other thread is unable to proceed due to a different lock
- Review sample code: `week11_1/3_deadlock`
  - I think this example is a little contrived

More Mutex Code Samples

- Let's have a look in unx511_samples
    - `https://github.com/jsellens/unx511_samples`
- `week11_1/4_clientServer` – UNIX domain sockets with queue
- `week11_1/5_msgServer` – INET domain sockets with queue
- `week11_1/6_msgPump` – message broker / proxy
    - See PPT file in `week11_1/0_documents`

References

- On Ubuntu, for the posix man pages:
  `sudo apt install manpages-posix-dev`
- `https://github-pages.senecapolytechnic.ca/unx511/Week11/Week11.html`
- The Linux Programming Interface book, chapter 30 “Threads: Thread Synchronization”

Summary

- Threads can be handy when:
    - When you have multiple disjoint “tasks” in one program
    - When your processing can take advantage of parallelization across cores/processors
- Mutexes can help keep threads from colliding with each other
    - Whether it’s modifying shared data
    - Or providing output that you don’t want mixed together
    - Or ...