Threads

Threads

A thread is an independent execution sequence within a single process.

- Operating systems and programming languages generally allow processes to run two or more functions simultaneously via threading.
- The stack segment is subdivided into multiple miniature stacks, one for each thread.
- The thread manager time slices and switches between threads in much the same way that the OS scheduler switches between processes.
- In fact, threads are often called lightweight processes.
- Each thread maintains its own stack, but all threads share the same text, data, and heap segments.

Pros and Cons of Threads vs.Processes

- Pro: it's easier to support communication between threads, because they run in the same virtual address space.
- Con: there's no memory protection, since virtual address space is shared. Race conditions and deadlock threats need to be mitigated, and debugging can be difficult. Many bugs are hard to reproduce, since thread scheduling isn't predictable.
- Pro and con: Multiple threads can access the same globals.
- Pro and con: One thread can share its stack space (via pointers) with others.

pthreads

- ANSI C doesn't provide native support for threads.
- But **pthreads**, which comes with all standard UNIX and Linux installations of **gcc**, provides thread support, along with other related concurrency directives.
- The primary **pthreads** data type is the **pthread_t**, which is a type used to manage the execution of a function within its own thread of execution.
- The only **pthreads** functions we'll need are **pthread_create** and **pthread_join**.

Examine introverts!

Key points of introverts

- Introverts declares an array of six **pthread_t** handles.
- The program initializes each **pthread_t** (via **pthread_create**) by installing **recharge** as the thread routine each **pthread_t** should execute.
- All thread routines take a **void *** and return a **void ***. That's the best C can do to support generic programming.
- The second argument to **pthread_create** is used to set a thread priority and other attributes. We can just pass in **NULL** if all threads should have the same priority. That's what we do here.
- The fourth argument is passed verbatim to the thread routine as each thread is launched. In this case, there are no meaningful arguments, so we just pass in **NULL**.
- Each of the six **recharge** threads is eligible for processor time the instant the surrounding **pthread_t** has been initialized.
- The six threads compete for thread manager's attention, and we have very little control over what choices it makes when deciding what thread to run next.

pthread_join waits

- **pthread_join** is to threads what **waitpid** is to processes.
- The main thread of execution blocks until the child threads all exit. The second argument to **pthread_join** can be used to catch a thread routine's return value.
- If we don't care to receive it, we can pass in **NULL** to ignore it.

Deadlocks and Race Conditions

- Our next program will seem harmless but has a fatal flaw.
- Here we all introduce ourselves on the last day of classes.

Examine confused-friends!

What went wrong?

- Note that **meetup** dereferences its incoming parameter and that **pthread_create** accepts the address of the surrounding loop's index variable **i** via its fourth parameter. **pthread_create**'s fourth argument is always passed verbatim as the single argument to the thread routine.
- The problem? The main thread advances **i** without regard for the fact that **i**'s address was shared with many child threads.
- At first glance, it's easy to absentmindedly assume that pthread_create captures not just the address of **i**, but the value of **i** itself. That assumption of course, is incorrect, as it captures the address and nothing else.
- The address of **i** (even after it goes out of scope) is constant, but its contents evolve in parallel with the execution of the **meetup** threads.
- ***(int *)args** in **meetup** takes a snapshot of whatever **i** happens to contain at the time it's evaluated.
- This is an example of a race condition and is typical of the types of problems that come up when multiple threads share access to the same data.

Here the fix is simple!

Instead of passing **i** and getting the names within **meetup,** let's pass the address to each name (i.e., char *) as the argument to the thread routine.

Examine friends!

Sharing data

- Sharing data can be complicated and dangerous in concurrent execution.
- Concurrent programming often makes use of specific tools to control how data is shared between threads
  - Mutexes
  - Semaphores
  - Condition variables
  - Etc.

Examine robberBaronsBroken!

Something is wrong!

- How do we know?
  - Printing is out of order at the end
  - Negative value for the **stash**?
- Multiple threads are modifying the global variable **stash**
- Is it possible for two threads to evaluate **stash > 0** as True with only $10000 left and then both subtract from stash?
  - Yep! Say thread A evaluates **stash > 0** and then the thread manager switches to thread B before thread A subtracts the steal money from the **stash.**
  - Thread B executes fully bringing the stash to $0.
  - Thread A resumes execution and subtracts its $10000 bringing the total to -$10000.
  - Yikes!

Mutexes

- A mutex is a **mut**ual **ex**clusion object.
- It is a locking mechanism to protect shared data or critical regions of code so that only one thread can be permitted access.
- Here we need to protect the stash so that only one robber can modify the stash at a given time.
- We declare a mutex with **pthread_mutex_t**.
- To lock a piece of code, we use **pthread_mutex_lock().**
  - When a thread tries to acquire a lock, it will either take the lock if it is not being currently used or it will wait until the lock becomes available.
- To unlock a piece of code, we use **pthread_mutex_unlock().**
  - Note that the only code that is allowed to unlock the lock is the thread that currently has the lock

Examine robberBarons!