CptS 360 (System Programming) Unit 13: Threads

Bob Lewis

School of Engineering and Applied Sciences
Washington State University

Spring, 2022

Motivation

- Threads are in very common use today.
- They can take advantage of multi-core architectures.
- The boss/flunky architecture they present themselves to suits a wide variety of applications.

References

- Stevens & Rago Ch. 11
- Stones & Matthew "Beginning Linux Programming"
- *man* pages (mostly *pthreads*)

Thread Concepts

- A thread, or thread-of-execution, is that part of a process which follows a sequence of instructions, accessing data and requesting system services as needed.
- All processes have at least one thread.
- We are concerned here with processes that have more than one thread.

Thread Benefits

- Threads can deal cleanly with asynchronous events.
- Threads can make use of multiprocessors (but they don't have to).
- Threads are easier to manage than multiple processes (with shared memory).
- Threads can improve response time: one thread can be blocked while others proceeed.
- Example: A Simple Server Architecture
    - One thread synchronizes UI/GUI with internal data structures.
    - One thread dispatches others to fulfill requests.
    - $N - 2$ threads focus on individual requests.

POSIX Threads, a.k.a. "`pthreads`"

- supported almost everywhere
- (run `demos/d`n`_check_posix`)
- can be implemented portably even if multiprocessors not required.
- require some tricky (but portable) assembler language
- require one special instruction (recall it from CptS 260?)

`pthread` Thread Execution

- creation
  - *pthread_create(3)*
  - Note typo in *pthread_create(3)* prototype on bottom of p 357. ("`void`" argument should be "`void *`").
  - Under Ubuntu, you may need to install the `manpages-posix` and `manpages-posix-dev` packages to get the `pthreads` man pages.
- identification
  - *pthread_self(3)*
  - *pthread_equal(3)*
- (run `demos/d`*n*`_thread_id`)
- passing thread-specific data
  - as argument
  - in global `struct` keyed by `thread_id` (but this may be tricky)

pthreads Demos

- (run demos/dn_thread_argument)
- (run demos/dn_exit_status)
- (run demos/dn_bad_exit)
- (run demos/dn_cleanup)

The Bead Analogy

- Think of a single-threaded program as beads on a wire representing clock time, with the beads being individual instructions.
- Beads may not be all together on a multitasking OS, especially if there's OS work to be done.
- Multithreading allows your program to create new wires and specify new beads to go along them, but again, the beads may not be all together. Beads can slide back and forth on any given wire.

The Bead Analogy II

Consider the statement

```
a[++n] = 42:
```

Where `a[]` is a variable in shared memory. In MIPS, this would compile to something like:

```
la   $t0,n
lw   $t1,($t0)
addi $t1,$t1,1
sw   $t1,($t0)
la   $t2, a
add  $t2,$t2,$t1
li   $t3,42
sw   $t3,($t2)
```

Now run two threads of this side-by-side. Remember: Each machine instruction is a "bead."

Case 1: Effectively Serial

If n starts out as 2, what is its final value?

Thread 0:

```
la   $t0,n
lw   $t1,($t0)
addi $t1,$t1,1
sw   $t1,($t0)
la   $t2,a
add  $t2,$t2,$t1
li   $t3,42
sw   $t3,($t2)
```

Thread 1:

```
la   $t0,n
lw   $t1,($t0)
addi $t1,$t1,1
sw   $t1,($t0)
la   $t2, a
add  $t2,$t2,$t1
li   $t3,42
sw   $t3,($t2)
```

Case 2: Another Possible Sequence

```
Thread 0:                  Thread 1:

la   $t0,n
lw   $t1,($t0)
addi $t1,$t1,1
                           la   $t0,n
                           lw   $t1,($t0)
sw   $t1,($t0)
la   $t2, a
add  $t2,$t2,$t1
li   $t3,42
sw   $t3,($t2)
                           addi $t1,$t1,1
                           sw   $t1,($t0)
                           la   $t2, a
                           add  $t2,$t2,$t1
                           li   $t3,42
                           sw   $t3,($t2)
```

Case 3: A Slight Change to Case 2

Thread 0:

```
la   $t0,n
lw   $t1,($t0)
addi $t1,$t1,1
sw   $t1,($t0)


la   $t2, a
add  $t2,$t2,$t1
li   $t3,42
sw   $t3,($t2)
```

Thread 1:

```




la   $t0,n
lw   $t1,($t0)




addi $t1,$t1,1
sw   $t1,($t0)
la   $t2, a
add  $t2,$t2,$t1
li   $t3,42
sw   $t3,($t2)
```

The Synchronization Problem

Lessons learned:

- The *global* order in which the beads are executed may affect the result.
- The result may depend on the order in which the instructions are scheduled by the processor.
- Sometimes or even *most* of the time, it may do the right (serial) thing.

To solve this problem, we need some way to synchronize access to shared memory.

Mutexes

Mutual exclusion "devices" (an abstraction), a.k.a *mutexes* declare a certain sequence of (instruction) beads to be accessed by only one thread at a time. They can lock both data and code.

- *pthread_mutex_init(3)*
- *pthread_mutex_lock(3)*
- *pthread_mutex_unlock(3)*
- *pthread_mutex_trylock(3)*
- *pthread_mutex_destroy(3)*

Mutex Demos

These demos show the need for and use of mutexes:

- (see demos/dn_mutex_conceptual)
- (run demos/dn_stdout_buffered)
- (run demos/dn_stdout_unbuffered)
- (run demos/dn_clock_unthreaded)
- (run demos/dn_clock_threaded)
- (run demos/dn_stopwatch_without_mutex)
- (run demos/dn_stopwatch_with_mutex)

Restricting Access With Mutexes

- to data structures ...
  - If you want to locking access to a data structure, consider adding a mutex ("lock") to the structure. (in constructor?)
  - To access the data structure, a function must first acquire the mutex and must release it when done.
- to code ...
  - A block of code protected by a mutex is called a *critical section*.
    1. create mutex
    2. acquire mutex
    3. run critical section code
    4. release mutex
    5. delete mutex
  - Be careful of any `setjmp`/`longjmp`s in or below #3. (Think "`finally`" in an exception.)
- Remember that mutexes take time to create, acquire, release, and delete, so these may not always be the most efficient approaches.

Mutex Lock Demos

- (see demos/dn_two_mutexes_conceptual)
- (see demos/dn_simpler_locking_conceptual)

Deadlock Avoidance

- What is deadlock?
  1. multiple mutexes to acquire
  2. multiple threads running
  3. Thread 0 holds a mutex Thread 1 wants, but needs a mutex B holds and vice versa.
  4. Lock ordering not always possible.
- (see `demos/d`n`_two_mutexes`)
- (see `demos/d`n`_simpler_locking`)
- The second is simpler because it uses the hash list lock to protect the list traversal as well.

Reader-Writer Locks

Mutexes have two states: locked and unlocked. Reader-writer locks have three: read-locked, write-locked, and unlocked. Read-locking permits any number of readers. Write-locking permits only one writer.

- *pthread_rwlock_init(3)*
- *pthread_rwlock_destroy(3)*
- *pthread_rwlock_rdlock(3)*
- *pthread_rwlock_tryrdlock(3)*
- *pthread_rwlock_wrlock(3)*
- *pthread_rwlock_trywrlock(3)*
- *pthread_rwlock_unlock(3)*
- (see `demos/d`n`_rwlock_conceptual`)

Condition Variables

These wait for a "condition variable" to become ready. These can implement "barriers".

- *pthread_cond_init(3)*
  - or set condition variable to `PTHREAD_COND_INITIALIZER`
  - `cond_attr` arg is ignored on Linux
- *pthread_cond_signal(3)*
  - wakes up one thread.
- *pthread_cond_broadcast(3)*
  - wakes up all of them.
- *pthread_cond_wait(3)*
- *pthread_cond_timedwait(3)*
- *pthread_cond_destroy(3)*

Semaphores I

- Don't confuse these with IPC semaphores (later).
  - they're intended for threads and only work within one process (for now).
- *sem_init(3)*
  - creates a semaphore
- *sem_post(3)*
  - always increases the value of the semaphore by 1.
  - It does this *atomically*, so if two processes call *sem_post()* at the same time, the semaphore is guaranteed to be incremented by two.

Semaphores II

- *sem_wait(3)*
  - atomically decreases the value of the semaphore by 1, but waits until the semaphore is nonzero before doing so.
  - If the semaphore is 0, the calling thread waits.
- *sem_trywait(3)*
  - Like *sem_wait()*, but doesn't wait.
- *sem_destroy()*
  - gets rid of a semaphore.
- (run `demos/d`*n*`_semaphore`)