15-213
“The Class That Gives CMU Its Zip!”

Introduction to Computer Systems

Taerim Kim
April 25, 2011

Topics:
- Threads
- Synchronization
- Final exam review
News

Malloclab
- They're graded. I have them. Remind me to return them at the end of class.

Proxylab
- Due Thursday, April 28. (My birthday!)
- Remember to sign up for a demo. If you want a grade for proxylab, you must sign up and show up for a demo.
- We are not (officially) accepting bribes.

Final exam
- Tuesday, May 3, at 1:00pm. Check the final exam schedule for the location.
- It's cumulative. D:
Threads

What are threads?

- A thread of execution—a “schedulable” entity.
- Typically contained within a process—there can be many threads per process.
- Share certain resources—perhaps most importantly, memory.

What constitutes a thread?

- A stack.
- Registers.
Single-threaded process

(Stolen from 15-410 lecture slide.)

Stack

Heap

Data

Code

Registers

stdin

stdout

timer
Multi-threaded process

(Stolen from 15-410 lecture slide.)
POSIX Threads (Pthreads)

POSIX standard for threads

- Thread management
  - pthread_create
  - pthread_join
  - pthread_self
  - pthread_cancel
  - pthread_exit

- Synchronization (More on this later.)
  - pthread_mutex_*
  - pthread_cond_*
  - pthread_sem_*
  - pthread_rwlock_*

#include <pthread.h>
#include <pthread.h>
void *thread(void *arg) {
    printf("Hello, world!\n"); return NULL;
}

int main() {
    pthread_t t[8]; int i;
    for(i = 0; i < 8; i++)
        pthread_create(t + i, NULL, thread, NULL);
    for(i = 0; i < 8; i++)
        pthread_join(t[i]);
    return 0;
}
Joinable v. detached threads

**Joinable threads**
- Exit when return from start routine or call `pthread_exit`.
- Must be reaped after exiting (using `pthread_join`).
- `pthread_join` blocks until target thread exits.

**Detached threads**
- Exit under same conditions.
- Automatically reaped after termination.

**Default state is joinable**
- Use `pthread_detach(pthread_self())` to detach a thread.

**For proxylab**
- You probably want detached threads.
**pthread_exit v. exit**

**pthread_exit**
- Terminates the calling thread.
- Does not terminate the entire process/other threads.

**exit**
- Terminates the entire process.
- All threads are gone—poof!

**Why should I care?**
- Think carefully about which is appropriate in the face of error.
- Different error conditions merit different responses.
Race conditions

What is a race condition?
- You had some exposure in tshlab.
- Nondeterminism in the behavior of a program.
- A program has a race condition if its result/correctness depends on the sequence/timing of events; i.e., how things end up being scheduled.
- They suck. *Hard.*

How do they come about?
- Shared variables/data structures.
- Threads dependent on a condition.
Race condition example

What is the output?

```c
static int foo = 0;
void *thread(void *arg) {
    printf("%d\n", foo++); return NULL;
}
int main() {
    pthread_t p, q;
    pthread_create(&p, NULL, thread, NULL);
    pthread_create(&q, NULL, thread, NULL);
    pthread_join(p, NULL);
    pthread_join(q, NULL);
    return 0;
}
```
Murphy's law of threading

(Stolen from 15-410 lecture slide.)

The world may *arbitrarily interleave* execution.

- **Multiprocessor**
  - N threads executing instructions *at the same time*.
  - Of course effects are interleaved!

- **Uniprocessor**
  - Only one thread running at a time...
  - But N threads runnable, timer counting down toward zero...

The world will choose the *most painful* interleaving.

- “One chance in a million” happens every minute.
Resolving race conditions

Carefully analyze operation sequences

Find subsequences which must be uninterrupted
- “Critical section”

Use a synchronization mechanism
- Mutex
- Condition variable (not in this course)
- Semaphore
- Read-write lock

Alternatively, avoid critical sections altogether
- Minimize shared variables, memory
Synchronization primitives

Semaphore
- Restricts number of threads that can concurrently access a resource.

Mutex
- Special case of semaphore where at most one thread can access a resource at a time.
- (The difference is actually slightly more nuanced, but not important for this course.)

Read-write lock
- Multiple readers allowed.
- Single writer allowed.
- No readers allowed when writer is present, and vice versa.
Fixing our previous example

```c
static int foo = 0; static sem_t sem;

void *thread(void *arg) {
    sem_wait(&sem);
    printf("%d\n", foo++);
    sem_post(&sem);
    return NULL;
}

int main() {
    pthread_t p, q;
    sem_init(&sem, 0, 1);
    pthread_create(&p, NULL, thread, NULL);
    pthread_create(&q, NULL, thread, NULL);
    pthread_join(p, NULL); pthread_join(q, NULL);
    return 0;
}
```
Tour of 15-213 (not exhaustive)

Data representation
- Integers, floating point, structs

Assembly
- IA-32, x86-64
- Stack discipline

Caching
- Motivation for caching
- Counting hits, misses, evictions
- Analyzing programs' cache performance
- Optimizing for cache performance
Tour of 15-213 (not exhaustive)

Process control
- Exceptional control flow (fork, wait, exec, exit)
- Signals, handling signals, blocking signals

System-level I/O
- open, read, write, close, dup, dup2
- File descriptor table v. file table v. v-node table

Virtual memory
- Motivation for virtual memory
- Design considerations, especially number of page levels
- Virtual-to-physical translation, including TLB
Tour of 15-213 (not exhaustive)

Dynamic memory allocation
- Motivation for DMA
- Design considerations

Threading
- Motivation for multithreading
- Identifying simple race conditions
- Synchronization techniques

Networking
- Sockets
- socket, bind, connect, accept, send, recv, close
The end

Good luck on proxylab, final exam.