CS61 Section Notes
Week 9 (Fall 2011)

Outline for this week:

- Stepping Back
- Threads
- Critical Sections, Locks, and Mutual Exclusion
- Concurrent Code in General

Stepping Back (or, a bit of CS philosophy)

Consider what happens when you use your computer. When you sit down to work on something, chances are you have many programs running at the same time. You almost certainly have a web browser open. Maybe you have a word processor open or some other editor you use to write your programs for CS61. Maybe you also have a chat client open (to tell your friends how awesome the CS61 course staff is, of course).

If you sit down and think about this situation for a moment, it is in some sense magical. Your computer has (probably) one to four processors (less than the number of running programs), and it has a fixed number of bytes of physical RAM. That RAM is referred to by integer addresses starting at 0 and sequentially increasing. Yet somehow, all those completely independent programs are running at the same time, and they are not stepping on each other's memory. Even more amazing, the programmers who wrote those programs did not have to explicitly coordinate with the programmers who wrote the other programs to make sure all the programs would work seamlessly. It "just works." (well, ok, most of the time...)

How was this magic accomplished? The answer lies in the design of the operating system. Part of the operating system's job is to abstract the hardware and allow running programs to efficiently share it. You have already seen one example of this: virtual memory. The operating system (with the help of the hardware) creates the illusion that each process gets its own memory "workspace" to play with, whose size is only limited by the number of bits used for a pointer. In reality, of course, the programs must ultimately all share the same, limited physical memory, but nonetheless the operating system is able to create and enforce this (very useful) illusion.

What does this all have to do with threads? What virtual memory is to physical RAM, threads are to the processor. Threads allow different programs to have multiple, totally independent execution environments. In other words, you can pretend you have exclusive access to the processor when writing a program in the same way you can pretend you have exclusive access to RAM. Even a simple C program that involves no process manipulation and no explicit calls to functions that spawn or control threads can be thought of a single, default thread executing your code in a single, default address space that the operating system has created for you.

With that said, let's get some terminology straight:

address space -- an independent copy of memory; think of separate houses as separate address spaces: what I cook in my kitchen is totally independent of what you cook in yours even though we both refer

to the place we manipulate the food as “the kitchen.”

thread -- the processor state associated with a particular execution; think of different people as different threads: I can be cooking one thing while you cook another.

process -- one or more threads operating in an address space; think of the set of people doing stuff in a house, along with the house itself, as being a process.

So, threads are great. But there’s a problem. Once you start using multiple threads, things stop happening one after the other and start happening **concurrently**, i.e., at the same time. But what could go wrong, right?...

Threading Example

The following program is very simple: the first thread immediately spawns two new threads. Each new thread increments a shared counter 2 million times and then exits. The original thread sequentially waits for each thread to finish, and then it prints out the value of the counter. Can you guess what ultimately gets printed out?

```
#include <stdio.h>
#include <pthread.h>
#define NITERS 2000000
void *count(void *arg);

unsigned int counter = 0; /* shared counter variable */

int main() {
    pthread_t tid1, tid2;

    pthread_create(&tid1, NULL, count, NULL);
    pthread_create(&tid2, NULL, count, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    if (counter != (unsigned)NITERS*2)
      printf("BOOM! counter=%d\n", counter);
    else
      printf("OK counter=%d\n", counter);
    exit(0);
}
void *count(void *arg) { /* thread routine */
    int i;
    for (i = 0; i < NITERS; i++)
      counter++;
    return NULL;
}


# Run the program over and over
$ sh test.sh
```

```
OK counter=4000000
BOOM! counter=3490653
BOOM! counter=3471683
BOOM! counter=3404617
…
```

The exciting answer: **You have no idea what will be printed out.** It may be different every time you execute the program.

Why? Because we've created a **race condition** (so named because the two threads are "racing" each other, each trying to access the shared variable counter as fast and as often as they can). The final value of `counter` will be determined by how often and when each thread gets scheduled (i.e., gets time using the processor), and that in turn may depend on a variety of things like what other processes are executing on your system and what they are doing.

Here's part of disassembly of the `for` loop code of function count():

C:

```
for (i = 0; i < NITERS; i++)
  counter++;
```

Assembly:

```
.L12:
    movl counter, %eax      #L: Load counter
    leal 1(%eax), %eax      #U: Update counter
    movl %eax, counter      #S: Store counter
```

Interleaving 1:

Step	Thread	Instruction	%eax1	%eax2	counter
1	1	L1	0	-	0
2	1	U1	1	-	0
3	1	S1	1	-	1
4	2	L2	-	1	1
5	2	U2	-	2	1
6	2	S2	-	2	2
7	1	L1	2	-	2
8	1	U1	3	-	2
9	1	S1	3	-	3

Interleaving 2:

Step	Thread	Instruction	%eax1	%eax2	counter
1	1	L1	0	-	0
2	1	U1	1	-	0
3	**2**	**L2**	**-**	**0**	**0**
4	1	S1	1	-	1
5	**2**	**U2**	**-**	**1**	**1**

6	2	S2	-	1	1
7	1	L1	1	-	1
8	1	U1	2	-	1
9	1	S1	2	-	2

Note that there is an important lesson here: **just because something is one instruction in C code does not mean it is one instruction in assembly** (and even if it were, we still might not be safe).

So what, oh what, is a lowly programmer to do???

Critical Sections, Locks, and Mutual Exclusion

We need to enforce mutual exclusion in the critical section where the counter variable is updated using a lock.

Erm...what does that mean?

mutual exclusion -- access is limited to be "one at a time"
critical section -- a piece of code (a sequence of instructions) requiring mutual exclusion
lock -- a means of enforcing mutual exclusion

Think of an intersection of two streets. If any arbitrary access were allowed to the intersection, cars might easily crash into each other. The intersection here is the critical section, and we need to enforce mutual exclusion on it. So we put a stoplight in. Only one road has a green light at a time. You can think of having the green light as acquiring a lock on the intersection. If someone else has the green light (the lock), you have to wait until they release it before you can acquire it and go through the intersection.

So what about our code? We need to make sure that reading, updating, and storing back the variable `counter` to memory all happens atomically, i.e., all at once from the perspective of the outside world. So, we need a lock (technically, a mutex...we'll get to this distinction in the next couple of lectures).

```
-----
#include <stdio.h>
#include <pthread.h>
#define NITERS 2000000
void *count(void *arg);

unsigned int counter = 0; /* shared counter variable */
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;

int main() {
    pthread_t tid1, tid2;
```

```
    pthread_create(&tid1, NULL, count, NULL);
    pthread_create(&tid2, NULL, count, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    if (counter != (unsigned)NITERS*2)
      printf("BOOM! counter=%d\n", counter);
    else
      printf("OK counter=%d\n", counter);
    exit(0);
}
void *count(void *arg) { /* thread routine */
    int i;
    for (i = 0; i < NITERS; i++) {
      pthread_mutex_lock(&mutex1);
      counter++;
      pthread_mutex_unlock(&mutex1);
    }
    return NULL;
}


...
$ sh test.sh
OK counter=4000000
OK counter=4000000
OK counter=4000000
-----
```

Ahh...much better! And, unfortunately, much slower (why?). But slow and working is always better than fast and broken!

Here's another example. Can you spot the race condition?

```
#include <stdio.h>
#include <pthread.h>
#define N 4

void *thread(void *vargp);

int main() {
      pthread_t tid[N];
      int i;


      for (i = 0; i < N; i++)
            pthread_create(&tid[i], NULL, thread, &i);
      for (i = 0; i < N; i++)
            pthread_join(tid[i], NULL);

      return 0;
}


/* thread routine */
void *thread(void *vargp) {
      int myid = *((int *)vargp);
      printf("Hello from thread %d\n", myid);
      return NULL;
}
```

When the main thread creates a peer thread it passes a pointer to i. Depending on whether the main thread is run or the newly created thread, the value of i could be modified. How can we fix this?

```
#include <stdio.h>
#include <pthread.h>
#define N 4

void *thread(void *vargp);

int main() {
      pthread_t tid[N];
      int i, *ptr;

      for (i = 0; i < N; i++) {
            ptr = malloc(sizeof(int));
            *ptr = i;
            pthread_create(&tid[i], NULL, thread, ptr);
      }
```

```
    for (i = 0; i < N; i++)
        pthread_join(tid[i], NULL);

    return 0;
}

/* thread routine */
void *thread(void *vargp) {
    int myid = *((int *)vargp);
    free(vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

Suggestions for Programming Concurrent Code

1. Assume that anything can happen in any order, because in general it can. From this basic assumption, figure out the situations where you can, in fact, guarantee an order of operations between threads or processes (because of mutexes, waits, etc.)
2. Figure out which **invariants** your program needs to maintain (i.e., things that are true no matter what), and then use mutual exclusion to enforce these invariants.
   An example of an invariant might be, "the value of the 'total assets' is always the sum of the amount in the checking account and the amount in the savings account." You might ensure this by, for instance, always acquiring a lock on the checking account (or savings account), then acquiring a lock on the "total assets" value, and updating both before releasing both locks so that the two updates occur atomically.
3. Debugging concurrent programs is notoriously hard. By definition, two executions of the same program may behave differently. If you have a race condition, a bug may only show up in one of those executions. If you're truly unlucky, that bug may only show up in, say, one in every million executions. But if the program you're debugging involves transferring millions of dollars every time it executes, that one buggy execution may be a big problem!
   The best way to debug a concurrent program is to have rigorously designed it in the first place so you don't have to debug it. That may sound flippant, but it's true. Avoid reasoning about concurrent programs on the fly, and instead be rigorous about any changes you make to the logic of the program. Changing the behavior of one part of the code will almost certainly have ramifications for other pieces of code.
4. When you start writing concurrent code, and you need to make it run quickly, you will often find that some lock, somewhere becomes a bottleneck. Think about a crowd trying to quickly exit a stadium through a single revolving door. Remember that locking arises from the need to safely share data among concurrent threads. The best way to alleviate this situation is to find a way to avoid sharing data in the first place. Often it becomes worth it to give each thread a separate copy of the same data structure in order to make this possible. If you and your 6 friends are going through a list of potential donors to your political campaign, it's probably worth giving each person their own copy of the list, even though those lists are redundant.
5. Take CS262 and/or CS264. :)

Some Questions to Ponder

1. Normally it is unsafe for one thread to access data on another thread's stack.
   a. Why?
   b. Is there a way to make it safe?
2. When should you use threads, and when should you use processes?
3. What kinds of things could go wrong when programming with mutexes?
4. Let's be a little more concrete: in Assignment 4, you all experienced the sheer joy of implementing malloc. How might you design malloc so that it's **thread-safe** (i.e., so that multiple different threads can call malloc and free at the same time)?
   a. What's the most obvious solution? What are the possible downsides to it?

b. You might be tempted to just use separate free lists for each thread. Why wouldn't this work (at least without some extra mechanisms)?
c. So what kinds of things might you do?

Excercise

Consider the following two functions (and global variable definition)

```
volatile int x = 0;
volatile int y = 0;

void *foo(void *arg) {
    int z = 0;
    y++;
    z = 5;
    printf("Hello from foo: x,y,z is %d,%d,%d.\n", x, y, z); }

void bar() {
    int z = 7;
    x = 2;
    printf("Hello from bar: y is %d, z is %d.\n", y, z); }
```

1) Write a main function that forks a new thread and executes `foo` in the new thread. The main thread should execute `bar`, and then wait for the new thread to finish. (Use the pthreads library.)

2) The two threads access some resources concurrently. Declare, initialize and use locks to prevent race conditions in the code.

3) What are the possible outputs of the program?