CS61 Scribe Notes – Lecture 24 – 11/29/2012

Assignment 5

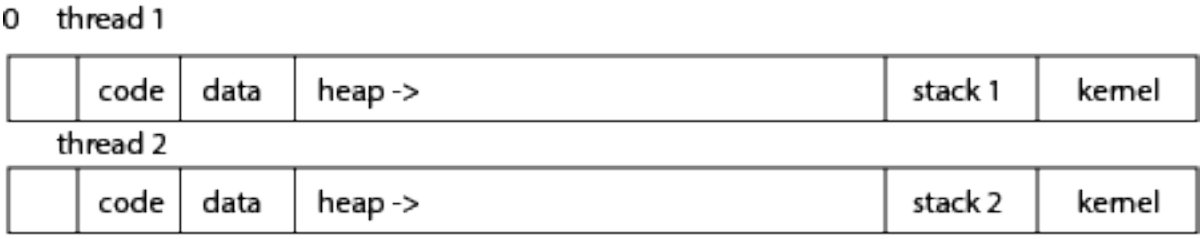
- Assignment 5 will be released an hour after class. Due Wednesday, the last day of reading period.
- `./pong61`
- Visit URL `http://cs61.seas.harvard.edu:6168/test/`
- What's going on here? Adversarial network pong is getting played against the server. Each circle is a new connection to the server.
- In phase 0, the code is making one connection after another, denoted by a circle bouncing around.
- In phase 1, server will occasionally drop some packets, denoted by black explosions and holes. The code handed out doesn't know what to do when a packet is dropped. The server notices this and says "no, no, no!"
- In phase 2, it gets more interesting as the server is delaying each of its responses to the pong client. The server wants to move quickly but it's delaying its responses. Our code like the code from last time for service server only processes one connection at a time so it waits for a connection to return before it makes its next connection. So when the server starts to delay its responses, the trail of its balls slows down significantly.
- In phase 3, not only do you have loss and delay, the server doesn't want you to overload it with connection attempts. And we get a BOOM! Everything goes red.
- There are two more phases after that.

Threads

- We encountered threads last time when we used threads to process multiple connections in parallel from service server.
- What is a thread? A thread abstracts a CPU.
- When we ran the service blaster program against the service server that used threads the service server died with an error
- Let's recreate that error via the lecture23 repo by running the service server and the service blaster against it
  - `./serviceserver07`
  - `./serviceblaster`
- We get the error message `Cannot allocate memory`
- What memory is being allocated? When we have an abstract CPU, it's not just an abstraction of the set of registers because <u>every thread has its own stack</u>
- On the stack, we have local variables with automatic storage duration. I.e. when a function exits, the variables contained within disappear; said another way, the lifetime of a variable is the lifetime of the functions that contains it
- The stack contains a list of activation records for function
- Let's draw a memory diagram:

- Why do multiple threads need multiple stacks?
- What would happen if all threads shared the same stack?
- What a crazy notion! There would be all kinds of weird errors.

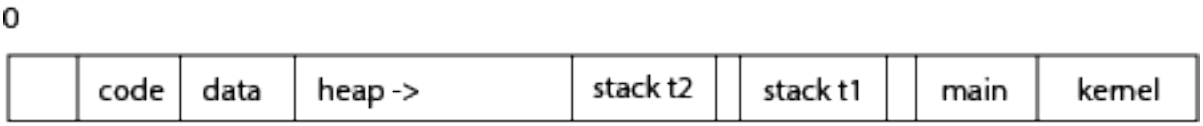
- In order to create a new thread with one stack, we need to find space for each thread's stack.
- What if we had multiple threads on top of each other like this with two threads but with code, data, heap shared and only different stacks.



- This is not how it works but we could try to do this with virtual memory so each thread would have its own virtual address space with different addresses in stack.
- Multiple threads have better utilization of memory than multiple processes.
- Ok so what experiment could we run to find where theses stacks are going?
- Print out a stack pointer
  - in serviceserver07 add:

```
void *connection_thread(void *arg) {
    FILE *f = (FILE *) arg;
    fprintf(stderr, "%p\n", &f);
    pthread_detach(pthread_self());
    handle_connection(f, f);
    fclose(f);
    pthread_exit(0);
}
```

- The address of f is 0xb771b35c.
- A normal stack pointer inside the main() function is 0xbf… that change each time. Recall that this is a security measure. Without this bit of randomization, it would be easier to do a buffer overflow attack if you know exactly what address the stack has.
- Main thread and child threads have a bit of randomization. This is called address space layout randomization.
- Ok so let's erase thread 2 from the diagram and draw the actual picture:



- There's roughly ~0x8000000 difference between the stack for thread 1 and the stack for thread 2. This allows plenty of room for the stacks to grow down. For example, we could call deeply recursive functions in each of the threads such that the stack for thread 1 will hit the stack for thread 2. This would be a stack overflow.
- In fact, the error message `Cannot allocate memory` means that the machine ran out of space for additional stacks because so much space is allocated for each thread's stacks. So we can't fit that many stacks into an address space, particularly for 32-bit machines like the lovely Mr. Brick. Note that this does not mean we ran out of physical memory, we ran out of memory for stacks.
- Question: what's the advantage of this over multiple stacks?
- Let's look at the serviceserver07 code in main()

```
// Start thread to handle connection
pthread_t t;
FILE *f = fdopen(cfd, "a+");
int r = pthread_create(&t, NULL, connection_thread, (void *) f);
```

- `&t` – name of thread id
- `NULL` – gives attributes for new thread, usually NULL
- `connection_thread` – function that should run when thread starts
- `(void *) f` – argument into that function, in this case connection_thread

- When you go back up to connection_thread, we see that the first argument is of void* type, basically any pointer
- We're passing the pointer f to connection thread
- Question: where does the argument f come from?
    - In serviceserver04, a process is created to handle each connection; i.e. we have forking and see that the child is handling a specific file

```
// Fork child to handle connection
pid_t p = fork();
if (p == 0) {
    FILE *f = fdopen(cfd, "a+");
    handle_connection(f, f);
}
```

    - In the threaded version it's analogous to the process version in serviceserver04 in that each thread is supposed to handle a specific file passed to it
- So if different threads have different virtual memory spaces, we would not be able to pass addresses of arguments on the main stack. In a program with multiple threads, all threads using the same address space so they can share variables. A situation with some memory shared and some not is more complicated.
- Where are all the stack pointers for each thread stored? In kernel in Linux.
- Recall from Assignment 3 had a set of registers in kernel corresponding to each process called pregisters. You had to set the eax register when you created a child process. In Linux, each thread has its own set of registers. When a thread makes a system call or a timer interrupt happens to stop thread, Linux takes a snapshot of the registers and stores them. Because threads share memory, there have been in the past thread packages on other OS's where thread registers stored held in application memory; in modern machines, however, it's usually the kernel.
- Let's say again why we can't have different virtual memory spaces. Here's an example that would be impossible with that scenario:
- So let's say we wanted to pass a file ptr and file number.
- We create a struct in serviceserver07.c, change the connection_thread function and then change how the thread was created.

```
struct pass {
    FILE *f;
    int fd;
};

void *connection_thread(void *arg) {
    ~~FILE *f = (FILE *) arg;~~
    struct pass *p = (struct pass*) arg;
    ~~fprintf(stderr, "%p\n", &f);~~
```

```
    fprintf(stderr, "%p\n", p);
    pthread_detach(pthread_self());
    handle_connection(p->f, p->f);
    fclose(p->f);
    pthread_exit(0);
}

// Start thread to handle connection
pthread_t t;
FILE *f = fdopen(cfd, "a+");
struct pass p;
p.f = f;
p.fd = cfd;
int r = pthread_create(&t, NULL, connection_thread, (void *) &p f);
```

- This works in the actual design in which threads share memory. It would not work with the other world with different virtual address spaces for each thread. The pointer would be meaningless there.
- What a great segue into the "awful" world of synchroniczation!

Synchronization

- Synchronization: the art of writing correct multi-threaded code
- What does it mean for multi-threaded code to be correct? No bugs.
- E.g. let's say we have two thread functions, t1 and t2 that both take on a void* argument that is actually a pointer to an integer

void t1(void *arg) {
  int *p = (int*) arg;
  (*p)++;
  return 0;
}
	void t2(void* arg) {
	  int *p = (int*) arg;
	  (*p)--;
	  return 0;
	}

- Let's say I started these threads in parallel, in the following way:

```
main(…) {
    int x = 0;
    pthread_create(..., t1, &x);
    pthread_create(…, t2, &x);
   // then wait for both threads to exit which happens after
incrementing or decrementing p //
    printf("%d\n", x);
}
```

- What do you expect to be printed out? We would expect 0. You add 1 with t1 then subtract 1 with t2.
- But a lot of things could go wrong because this program has no semantics. Including demons flying out of your nose! Because these threads are accessing the same variable at same time and doing writes at same time—this is illegal!

- This program might generate 0, -1, or 1 on a x86 machine. You might get any of these values. But you would need to run this a zillion times to get a bad example.
- This is a **race condition**, a bug dependent on scheduling order.
- Eddie just logged remotely onto a machine with 48 cores!
- serviceserver10.c (which is not yet up in the repo)

```
unsigned thread_count;
void *connection_thread(void *arg) {
    ++thread_count;
    FILE *f = (FILE *) arg;
    pthread_detach(pthread_self());
    handle_connection(f, f);
    fclose(f);
    --thread_count;
    pthread_exit(0);
}
```

- thread_count is a global variable, analogous to *p on board. thread_count is intending to keep track of the number of outstanding threads.
- Now another version of `connection_thread`. This code increments 1000 times and decrements 1000 times. By hiding the incrementing in another file, Eddie is preventing the compiler from optimizing it away. Logically it should be same as not doing this, i.e. adding 1000 and subtracting 1000 should give you zero.
- Eddie runs this serviceserver and in another window run a bunch of service blasters, making 400 connections in total.
- The service server thread count variable should have value of 400. To check, gdb into it in a mode that allows you to attach to a running, existing process using the pid of the serviceserver, instead of running a new version of serviceserver10.
- Print thread_count. Wowza…3071486790…that's a bit large!

--BREAK--

- What happens when you increment a variable in one thread and decrement in another thread and result is crazy? Let's look at how the processor does this.
- in x86 have single instruction that does increment called incl(%eax). Can do incl on a memory address. Another way to write this is addl(%eax, 1).
- However the actual processor doesn't implement this incl instruction in a single atomic step. It's actually 3 smaller instruction fragments.

  0.1. load (%eax) into some temporary space, register t
  0.2. T<-T+1
  0.3. store T into (%eax)

- If you have two cores, one implementing incl and one implementing decl. decl also breaks down to these 3 instruction fragments:

  1.1. load (%eax) into U
  1.2. u<-u – 1
  1.3. store U into (%eax)

- You can do arithmetic on a register. You can't do it on memory. Recall the difference between incl %eax vs. incl (%eax). The first is the register and can be done in a single step. The second is memory which uses the 3 steps above. T and U are hidden registers you can't name used as temporary storage. The instructions above are prefixed by their core (0 or 1).
- Assume the initial value in %eax is 0.
- If you perform 0.1, 0.2, 0.3, 1.1, 1.2, 1.3 in that order, the result is (%eax) is 1.
- But if the order is 0.1, 1.1, 1.2, 1.3, 0.2, 0.3, the result in (%eax) is 1.
- Let's see why:

	T	**U**	**%(eax)**
0.1	0		0
1.1	0	0	0
1.2	0	-1	0
1.3	0	-1	-1
0.2	1	-1	-1
0.3	1	-1	1

- Could get -1 by permuting instructions in another way.
- But actually, this order of instructions could also generate -1 as well. Either 1 or -1.
- What am I assuming when writing a list of instructions? You're assuming they don't interfere with each other and the core's caches don't interfere and as soon as a core does write, it goes into main memory immediately. This assumption is false. When a write goes to a cache, processor caches aren't write-through caches. Processor caches are not immediately coherent. Write to 1 and -1 (i.e. instructions 0.3 or 1.3 in the above diagram) might be reordered by processor caches.
- When we access the same memory location in multiple cores and not all accesses are reads, we open ourselves up to a world of pain! We need to enforce a critical section. The problem is **race conditions**. These are caused by simultaneous access to shared variables where at least one access is a write. If all accesses to a piece of shared memory are reads, they all return the same value and the accesses are safe.
- We want to enforce a **critical section** – a set of instructions so that no race condition occurs unless 2+ cores are executing those instructions at the same time.
- What is a trivial critical section for these logical instructions from above? Said another way, for what instructions can no 2+ cores be in those instructions at the same time? The 3 logical instructions for incl and the 3 logical instructions for decl form a single critical section.
- At most one core can execute a critical section. If 2+ cores are executing, that is a race condition.
- How do we enforce a critical section?

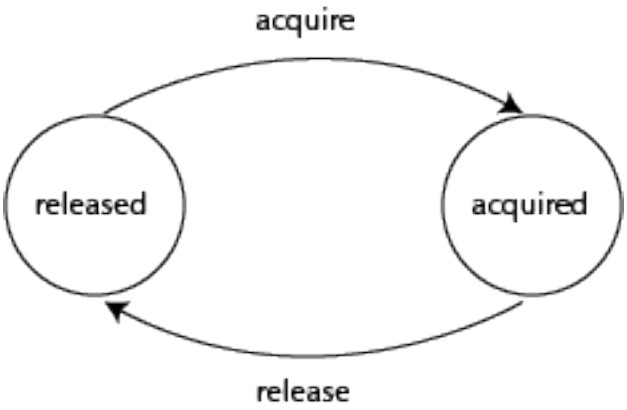
Locks and Mutual Exclusion

- **Mutual exclusion** (abbreviated as mutex) means that multiple threads or cores can't be in the same set of instructions simultaneously
- A **lock** is an object that provides mutual exclusion
- Let's say you have two threads: one doing increment, one doing decrement as before

```
void t1(void *arg) {
  int *p = (int*) arg;
  (*p)++;
  return 0;
}
```

```
void t2(void* arg) {
  int *p = (int*) arg;
  (*p)--;
  return 0;
}
```

- How do you provide mutual exclusion? What if you had an object, a piece of memory, supporting two operations—acquire and release with the following semantics:
- **Acquire**: must block until the lock is released. Block means that the acquire function stops and does not continue.
- **Release**: puts block in released state.
- What's key about **acquire** is that when multiple treads calling **acquire** on the same locked object, only one of them is going to win. Acquire will block until the lock is released and in one atomic step it will set the lock to an acquired state.
- Here's a state diagram for the lock. The lock has two states—released and acquired. The acquire operation moves from the released state to the acquired state. The release operation moves from acquired to released. Calling acquire on the acquired states blocks until the lock is released.



- Why does this provide mutual exclusion by enforcing a critical section? Because when a thread is blocked, it can't do anything else. It can't start executing instructions.
- At most one of the two threads will acquire the lock. Once the winning thread has acquire the lock, it can execute the critical section. It then releases the lock and another thread can acquire the lock.
- Clarification: the 6 micro instructions form a single critical section.
- How are the states of acquired and released implemented?
- Here's an idea: what's wrong with it?

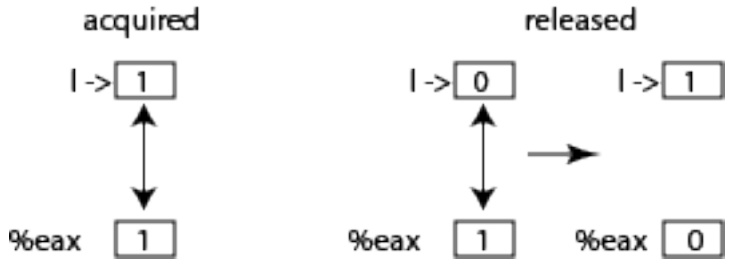
```
// just say mutex is an integer and 0 = release state, 1 = acquired
typedef int mutex_t

void release (mutex_t *l)
     *l = 0;

void acquire (mutex_t *l) {
     int x;
     while (*l != 0)
          /* do nothing */
     *l = 1;
}
```

- This acquire and release doesn't provide mutual exclusion. Have a critical section in acquire, which is a disaster. This is where we ask Intel for help: want a simple instruction for swapping between the register and memory.

- Swap says here's a register and a piece of memory. Swap the values in the register and memory in one atomic step. **Atomic** is something that happens in one step—you either observe the before or after state, never the intermediate state. E.g. those six micro instructions above are individually atomic.
- Let's change the acquire function. Let's say a lock is acquired (is 1). If you swap 1 and 1 nothing happens. If the lock is released, the lock is 0. Swapping 0 and 1 will allow you to break out of the while loop.

```
void acquire (mutex_t *l) {
     while (swap(l,1))
          /* do nothing */
}
```

- The x86 instruction for swap is actually called xchgl.
- There are two cases:
  1. lock is in acquired state – we want the acquire function to not exit or block
  2. lock is in released state – we want the function to exit with the lock in the acquired state. If it atomically puts a lock in the acquired state, anyone that calls acquire will be stuck in the acquired state until the thread that called acquire calls release.



- In the acquired state, the value 1 is in the piece of memory l. When you call swap of 1 with 1, nothing happens as the lock remains in the acquired state and keeps executing the while loop. Once lock is released, in one step we find out that the lock used to be released and in the exact same step set lock to the acquired state. The atomic exchange of releasing and acquiring in same step is key to mutual exclusion.
- Question: what does the swap function look like? We're pretending that swap function looks like this:

```
// exchanges value in *x with y
// returns previous *x
int swap(int* x, int y)
```

- How fast is xchgl? What has to happen to ensure that xchgl truly is atomic? None of the other cores can access that memory at the same time. It also has to make sure that it has the value in memory that is the latest anyone has written. It has to go to all the cores and ask if the value has been modified recently. Xchgl is very, very slow. Much slower than a conventional instruction.
- Here, the more correct our code is the slower it gets. Mutual exclusion is expensive because of slow xchgl and also low utilization. We'll see how to get around that next time!