CS61 Scribe Notes: November 29, 2012

Michelle Luo, Jamie Ryan

Table of Contents

General announcements

Assignment 5 will be released after class, due last day of Reading Period

- pong61
- Adversarial network pong
    - Each circle will be a new connection to the server
    - The server will periodically drop a few packets
        - These are the black explosions
        - Handout code doesn't know what to do when this happens
- Phase 2
    - Server delaying responses
    - Waits for a connection to return before moving to another
    - Will slow down considerably
- Phase 3
    - Not only loss and delay, but server doesn't want you to overload it with connection attempts

Threads

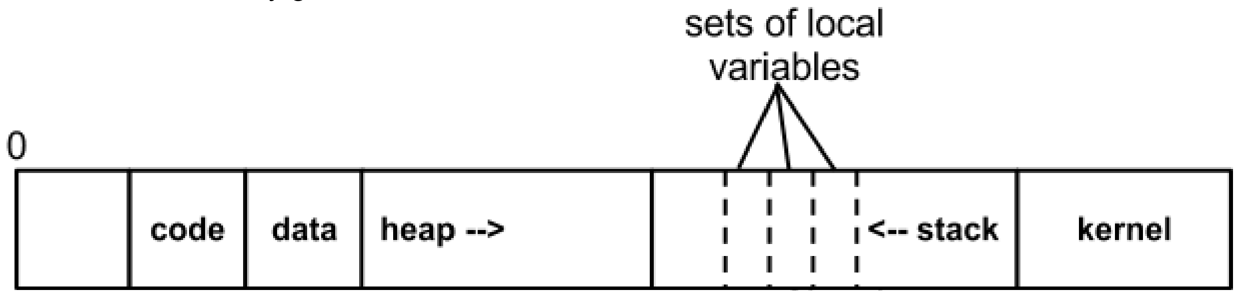
- Last time: threads used to process multiple connections in parallel
- A thread abstracts a CPU

Crashing serviceserver

When we run `serviceserver` with `serviceblaster`, `serviceserver` dies

- Commands run:
    - in `123` repo

        - `./serviceserver07`
        - `./serviceblaster`
        - `error: Cannot allocate memory`
- Why? What memory is being allocated?
    - Every thread has its own stack
    - Multiple threads need multiple stacks because stacks store local variables with automatic duration.
        - i.e. The lifetime of the variable is the lifetime of the containing function
    - The stack contains a set of activation records for functions
    - Memory gets laid out like this:



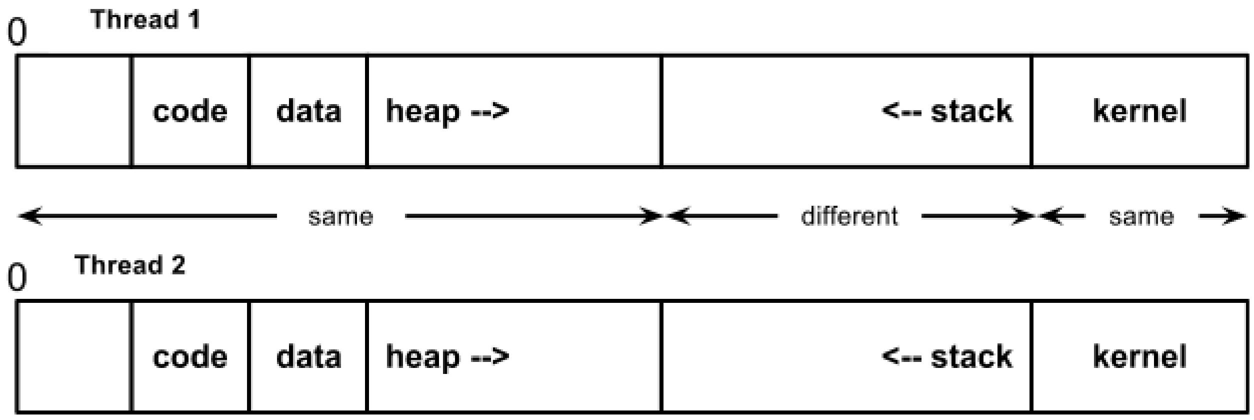
        - Stack grows down, heap grows up

Main difference between threads and processes:
- Threads share certain data (code, data, heap, kernel) because they abstract a CPU rather than the whole computer
    - Threads therefore have better utilization of memory than multiple processes

In order to create a new thread, we need to find space for it's stack

How to Organize the Stacks

Idea: stacks on top of each other:
- This can be achieved w/ virtual memory:

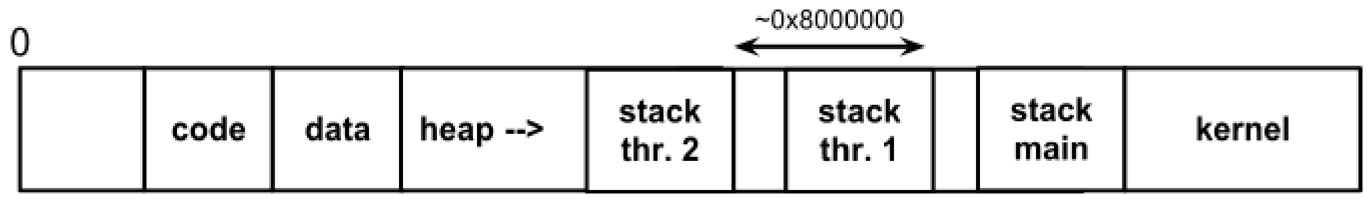


- But this is not how multiple threads are actually implemented

To discover where stacks actually occur, we can print out the addresses of local variables.
In `serviceserver07`:

- We get `0x6bf7935c, 0xb777a35c`
- Typically, a stack address would be `0xbffff` + some noise
- Here, the stack addresses change every now and then
    - A security measure
    - This makes it harder to execute a stack smashing attack
    - This technique is called *address space layout randomization*

Actual picture of the stack:



Here, we see that a significant chunk of memory exists between the stacks. This is so that they have room to grow down.

- This allows us to call deeply recursive functions
- If one stack hits another or the heap, this would be a stack overflow
- The error message above doesn't mean the machine is out of physical memory, it means that there is no more space for stacks in the address space.

What is the advantage of this over placing stacks on top of each other (virtual memory)?

- The code in `serviceserver07.c` calls `pthread_create` with these arguments
    - `(<thread id>, <attributes for the new thread>, <function that should run when thread starts>, <arguments for that function>)`
    - Here, we are passing `f` to `connection_thread`
- If different threads had different virtual memory spaces, it would be impossible to pass addresses of arguments on the main stack, since threads wouldn't be able to access other thread stacks

In modern machines, the stack pointers are stored in the kernel, along with each thread's registers. Consequently, the kernel manages switches from one thread to another.

Synchronization

synchronization: The art of writing correct, multithreaded code.

- What does *correct* mean?

Multithreaded Example

Suppose we have 2 thread functions:

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

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

main(...) {
    int x = 0;
    pthread_create(..., t1, &x);
    pthread_create(..., t2, &x);

    // wait for threads to exit

    printf("%d\n", x);
}
```

What you might expect to be printed out: 0
But this program has no semantics

- This leads to the nasal demons problems
- The two threads are accessing the same variable at the same time, and doing writes to it at the same time, which is illegal.
- If run on an x86 machine, this might generate 0, or 1, or -1
    - We might need to run many many times in order to actually get a bad example
- This is an example of a *race condition*

race condition: A bug that is dependent on scheduling order

An example of this bug, using a machine with 48 cores:
`./serviceserver10.c`

- `thread_count` is a global variable, analogous to `*p` in the code above
    - The number of threads outstanding at one time
    - When a new thread is started, increment
    - When `serviceserver` exits, decrement the thread count
    - We expect `thread_count` to equal the number of threads

- The code is written to do 1000 increments and 1000 decrements
    - Written in a way that it won't be optimized out
    - Theoretically, this should do nothing
- When we run `serviceblaster` and open 400 connections, we should expect that `thread_count == 400`
- But under `gdb` we see that the actual `value == 3071486790`

What goes on when a variable is incremented in one thread, decremented in another, and a weird results occurs?

Increments and Decrements

possible x86 instructions for incrementing a variable:

- `incl (%eax)`
- `addl (%eax), $1`

But the `incl` instruction is not actually executed in a single *atomic* step, since arithmetic can't be done on memory.

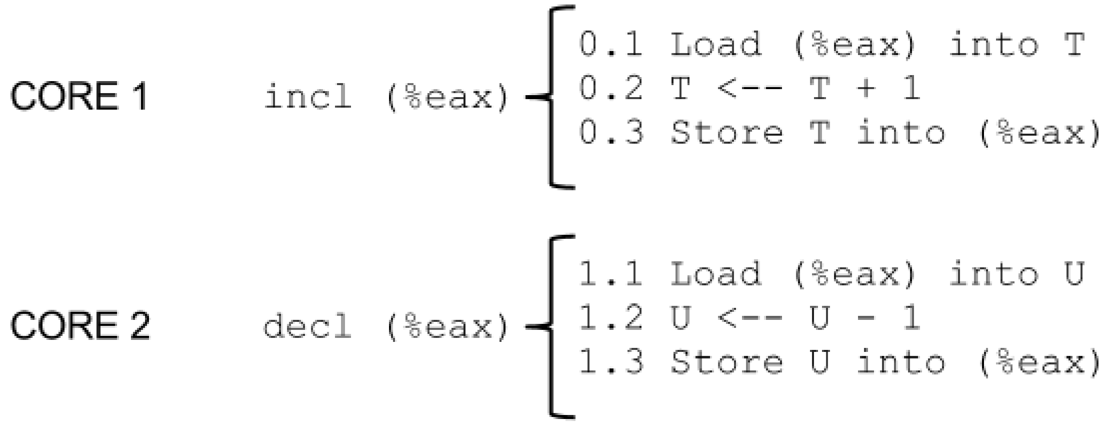
The processor breaks `incl` i into 3 smaller steps:

- 0.1. Load `(%eax)` into `T` (a hidden temporary register)
- 0.2. `T <- T + 1`
- 0.3. Store `T` into `(%eax)`

Similarly, the decrement instruction `decl` is broken into steps

- 1.1. Load `(%eax)` into `U`
- 1.2. `U <- U - 1`
- 1.3. Store `U` into `(%eax)`

Suppose there are two cores, one executing `incl` and one executing `decl`:



```
                                 0.1 Load (%eax) into T
CORE 1      incl (%eax)    -{    0.2 T <-- T + 1
                                 0.3 Store T into (%eax)

                                 1.1 Load (%eax) into U
CORE 2      decl (%eax)    -{    1.2 U <-- U - 1
                                 1.3 Store U into (%eax)
```

Suppose initial value in `(%eax) == 0`

- Potential order of steps:
  - 0.1
  - 0.2
  - 0.3
  - 1.1
  - 1.2
  - 1.3
  - -> final value is 0
- Another possible order:
  - 0.1
  - 1.1
  - 1.2
  - 1.3
  - 0.2
  - 0.3
  - -> final value is 1

How this progresses:

	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

In these cases, we are assuming that the instructions don’t interfere with each other

- In particular, we assume that the cores’ caches don’t interfere
- This is a false assumption
- When a write goes to a cache, the processor caches are not immediately coherent, so the write of -1 and 1 might be reordered by the processor caches
- So it’s also possible to get -1 with this exact same order of instructions

Critical Sections

Point: When we access the same memory location in multiple cores, and not all of them are reads, we get weird results

- We need to enforce a *critical section* to fix this

Problem: Race conditions caused by simultaneous access to shared variables, where **at least one access is a write**

- If all accesses are reads, they will all return the same value, so race conditions only become a problem when writing is involved
- Race conditions for individual instructions don't occur if there is only one core

critical section: Sets of instructions so that no race condition occurs unless 2 or more cores are executing instructions from those sets simultaneously

In our example, a trivial critical section would be the set of all 6 of the instructions.
We use *locks* to convince the processor that these instructions form a critical section.

Locks and mutual exclusion

mutual exclusion (mut ex): Prevents multiple threads/cores from executing a critical section simultaneously
lock: Object that provides mutual exclusion

Code modified from before:

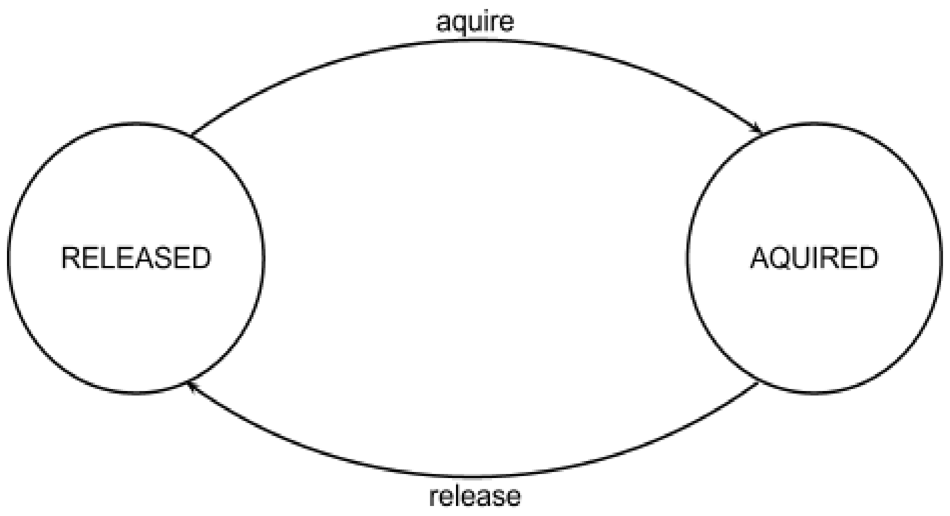
```
void t1(void *arg) {
    int *p = (int *) arg;
    acquire(&l);
    (*p)++;
    release(&l);
    return 0;
}

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

Suppose we have an object that supports two operations:

- acquire: Must block until the lock is released; **in one step** set to acquired state

- release: Puts the lock in released state

State diagram for the lock:



At most 1 thread can pass the acquire state and execute the critical section. It then releases the lock and another thread is allowed to acquire it.

Possible implementation of a lock:

```
typedef int mutex_t;
0 = RELEASED                         1 = ACQUIRED

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

but this requires a lock too! There is a critical section in acquire!

Alternative:

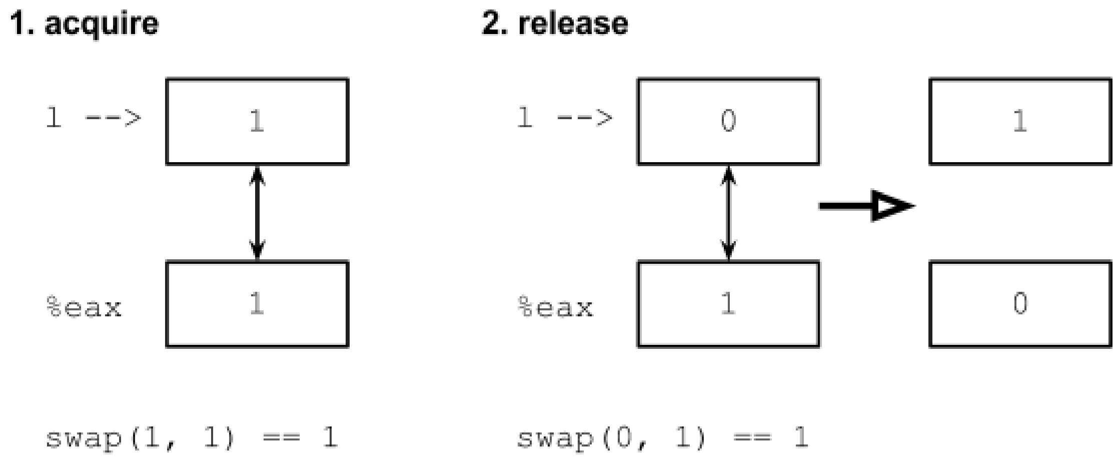
- Consider a function `swap` switches values in a register and memory atomically
    - `swap`'s actual x86 name is `xchgl`

atomic - defined by the architecture to happen in one step

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

void release(mutex_t *l) {
    *l = 0;
}
```

This method of mutual exclusion works!

Two possible situations:



Atomic exchange is key to mutual exclusion
hypothetical swap (to see how the lock/swap `xchgl` function works):

```
int swap(int *x, int y) {
    // Exchange value in *x with y
    // Returns previous *x
}
```

Speed of the swap

- On a machine with 48 cores, to ensure that `lock xchgl` is truly atomic, it has to prevent all of the other cores from accessing that memory at the same time and that it has the most up-to-date version of the memory
  - Must go to all the other cores, get new values
    - This is called the Mezzi protocol
    - Ends ups being very slow
- This will execute thousands of time slower than a typical instruction
- So now our problem is that the more correct the code is, the slower it gets.