Section 5:

Sunday, May 12, 2019 11:49 AM

section5

Section 5: Thread Synchronization

CS162

September 25-26, 2018

Contents

1 Warmup

1.1 Thread safety

Given a global variable:

```
int x = 0;
```

Are these lines of code thread-safe? In other words, can multiple threads run the code at the same time, without unintended effects? Assume an x86 architecture.

```
1: printf("x is %d\n", x);
2: int *p = malloc(sizeof(x));
3: x = 1;
4: int y = x;
5: x++;
6: x = rand();
```

No. x++ is a non-atomic read/write.

2 Vocabulary

- **thread** - a thread of execution is the smallest unit of sequential instructions that can be scheduled for execution by the operating system. Multiple threads can share the same address space, but each thread independently operates using its own program counter.
- **atomic operation** - An operation that appears to be indivisible to observers. Atomic operations must execute to completion or not at all.
- **critical section** - A section of code that accesses a shared resource and must not be concurrently run by more than a single thread.
- **race condition** - A situation whose outcome is dependent on the sequence of execution of multiple threads running simultaneously.
- **lock** - A common synchronization primitive. Possible actions are `acquire` and `release`. Locks can be either free or taken. Acquiring a free lock will cause it to become taken. Acquiring a taken lock will cause the acquirer to stop execution until the lock becomes free. A lock holder can release a lock to make it free again.
- **test_and_set** - An atomic operation implemented in hardware. Often used to implement locks and other synchronization primitives. In this handout, assume the following implementation.

```
int test_and_set(int *value) {
    int result = *value;
    *value = 1;
    return result;
}
```

This is more expensive than most other instructions, and it is not preferable to repeatedly execute this instruction.

Code chunks
Fukkin w/
Shared objects

Less privacy w/ threads
They live in the same box (process)
They do have their own individual state

Some outcome is said to "have a race condition" if it depends on sequence of simultaneous threads (cp)

What was there?
Anyways, set the value to 1
Return what it was before. (great for lock implementation)

3 Problems

3.1 The Central Galactic Floopy Corporation

It's the year 3162. Floopies are the widely recognized galactic currency. Floopies are represented in digital form only, at the Central Galactic Floopy Corporation (CGFC).

You receive some inside intel from the CGFC that they have a Galaxynet server running on some old OS called x86 Ubuntu 14.04 LTS. Anyone can send requests to it. Upon receiving a request, the server forks a POSIX thread to handle the request. In particular, you are told that sending a transfer request will create a thread that will run the following function immediately, for speedy service.

```
void transfer(account_t *donor, account_t *recipient, float amount) {
  assert (donor != recipient); // Thanks CS161

  if (donor->balance < amount) {
    printf("Insufficient funds.\n");
    return;
  }
  donor->balance -= amount;
  recipient->balance += amount;
}
```

Assume that there is some struct with a member `balance` that is typedef-ed as `account_t`. Describe how a malicious user might exploit some unintended behavior.

Das KOO

Hmm, so you're doing some stuff. You're fucking with it and
At various stages of execution, certain INVARAINTS on account balances are being broken
But htat's cool right, we assume precondition its unfucked. Then we make sure its unfucked when we leave. Clean up as you're done

Since you're a good person who wouldn't steal floopies from a galactic corporation, what changes would you suggest to the CGFC to defend against this exploit?

leave. Clean up as you're done

Since you're a good person who wouldn't steal floopies from a galactic corporation, what changes would you suggest to the CGFC to defend against this exploit?

Set up one donor account.
Set up n recipient accounts.
Create n threads to transfer some money, simultaneously, from each recipeint account to the SAME donor account.
When there is parallel execution over shared object donor balance, sometimes we mgiht UNDERdeduct.

But lucky lucky
Recipeints, since we create
MX by only allowing 1 thread fukkin widdit
At thes ame time, we always add :)
We sometoimes remove

Use the monitor pattern dummy.
Ensure MX for any methods that can modify the SO of account_t.

1) Add a lock, a type of synch var, which is a readily available primitive in most OS, as part of the attribute
2) For any method that interacts w/ the SO (instances of account_t), force it to acquire lock at method start, and release it at end.

For some reason, assholes are given power to "micro-optimize" by doing code blocks.
Just use a whole fucking method bro!

Deadlocoking is a TRAGIC case of concurrent programs where threads that require multiple lock acquisitions acquire some locks, but contest with ANOTHER thread that has the lock the other thread needs, and needs some locks from the other thread as well.

Most lock implementations will have both locks patiently wait for the other to release until the end of time.

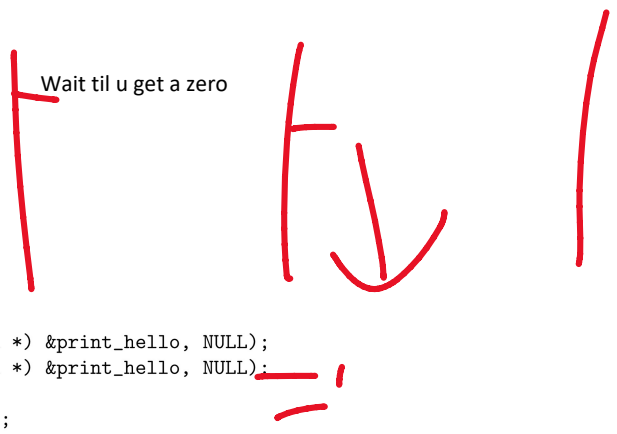
3.2 test_and_set

In the following code, we use test_and_set to emulate locks.

```
int value = 0;
int hello = 0;

void print_hello() {
    while (test_and_set(&value));
    hello += 1;
    printf("Child thread: %d\n", hello);
    value = 0;
    pthread_exit(0);
}

void main() {
    pthread_t thread1;
    pthread_t thread2;
    pthread_create(&thread1, NULL, (void *) &print_hello, NULL);
    pthread_create(&thread2, NULL, (void *) &print_hello, NULL);
    while (test_and_set(&value));
    printf("Parent thread: %d\n", hello);
    value = 0;
}
```



Assume the following sequence of events:
1. Main starts running and creates both threads and is then context switched right after
2. Thread2 is scheduled and run until after it increments hello and is context switched
3. Thread1 runs until it is context switched
4. The thread running main resumes and runs until it get context switched
5. Thread2 runs to completion
6. The thread running main runs to completion (but doesn't exit yet)
7. Thread1 runs to completion

Is this sequence of events possible? Why or why not?

No, thread 1 has never let go of it's "lock"

At each step where `test_and_set(&value)` is called, what value(s) does it return?

112

Given this sequence of events, what will C print?

112

Is this implementation better than using locks? Explain your rationale.

Use locks, this is spinlocking basically

3.3 Test and test_and_set?

To lower the overhead a more elaborate locking protocol test and test-and-set can be used. The main idea is not to spin in test-and-set but increase the likelihood of successful test-and-set by spinning until the lock seems like it is free.
Fill in the rest of the implementation for a test and test_and_set based lock:

```
int locked = 0;

void lock() {
    do {
        ----------------------- // Spin until lock looks empty
    } while (test_and_set(&locked));
}

void unlock() {

    Locked = 0;-------------
}
```

Layered spinlocking?>

Is this a better implementation of a lock that just using test_and_set? Why or why not?

4 Cache synchronization

Let's say you're building a local cache for Netflix videos at your favorite ISP. Here are some requirements for your cache:

- Your goal is to deliver videos to customers as fast as possible, but use as little of your ISP's ingress bandwidth as possible. So, you decide to build a **fully associative, multi-threaded cache**.
- You need to implement these **two** functions:
  - "`char *getVideo(int videoID)`" Returns a pointer to the video data (represented as a byte array).
  - "`void doneWithVideo(int videoID)`" The video needs to remain in the cache while it's being streamed to the user, so until we call this second function, `doneWithVideo`, you cannot evict the video from the cache.
- Your cache should have 1024 entries.
- You can use these functions:
  - "`char *downloadVideoFromInternet(int videoID)`" – Download a video from Netflix's origin servers. This function takes a while to complete!
  - "`void free(char *video)`" – Free the memory used to hold a video.
- You must never return an incomplete half-downloaded video to "`getVideo()`". Wait until the download completes.
- Your cache should never hold 2 copies of the same video.
- You must be able to download multiple videos from Netflix at the same time.
- Once a video is in the cache, it must be able to be streamed to multiple users at the same time.

First, design the struct that you will use for each cache entry. You can add locks or other metadata. You can also add more global variables, if you need them.

```
struct cache_entry {
    int videoID;
    char *video;
};

struct cache_entry CACHE[1024];
```

Next, implement the `getVideo` function. An implementation has already been provided, but it is not **thread-safe**.

```
char *getVideo(int videoID) {
    Lock.acquire();

    for (int i = 0; i < 1024; i++) {
        if (CACHE[i].videoID == videoID) {
                                                    CHA-CHING. CACHE HIT BABY
            While(downloading) {}
                Cache[i].users += 1;

            Lock.release();
           return CACHE[i].video;

        }
    }

    for (int i = 0; i < 1024; i++) {
        If ( CACHE[i].users == 0) {

        // Why is this in a for loop? I don't know.
        if (CACHE[i].video != NULL) {
            free(CACHE[i].video);
        }
        CACHE[i].videoID = videoID;

          Users = 1

        CACHE[i].video = downloadVideoFromInternet(videoID);   DI = false;
          Lock.release()
        return CACHE[i].video;
    }
      } else { lock.release(); getvideo(videoID);}

}
```

Finally, implement the `doneWithVideo` function:

```
char *doneWithVideo(int videoID) {


        For (int I = 0; I < 1024; i++) {
                If (cache[i].videoID == videoID) {
                        Cache[i].users -= 1;
                        Break;
                }

        }

    Lock.release
}
```