Section 5: Thread Synchronization

CS162

February 17, 2017

Contents

1 Warmup **2**
- 1.1 Thread safety . . . 2

2 Vocabulary **2**

3 Problems **3**
- 3.1 The Central Galactic Floopy Corporation . . . 3
- 3.2 test_and_set . . . 4
- 3.3 Test and test_and_set? . . . 6

4 Cache synchronization **7**

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();
```

1: It depends on the implementation. Usually guaranteed to be thread-safe.
2: It depends on the implementation. Usually guaranteed to be thread-safe.
3: Thread-safe.
4: Thread-safe.
5: Not thread-safe.
6: It depends on the implementation. Not guaranteed to be thread-safe, but it usually is.

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.

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.

There are multiple race conditions here.

Suppose Alice and Bob have 5 floopies each. We send two quick requests: transfer(&alice, &bob, 5) and transfer(&bob, &alice, 5). The first call decrements Alices balance to 0, adds 5 to Bobs balance, but before storing 10 in Bobs balance, the next call comes in and executes to completion, decrementing Bobs balance to 0 and making Alices balance 5. Finally we return to the first call, which just has to store 10 into Bobs balance. In the end, Alice has 5, but Bob now has 10. We have effectively duplicated 5 floopies.

Graphically:

Thread 1	**Thread 2**
temp1 = Alice's balance (== 5)	
temp1 = temp1 - 5 (== 0)	
Alice's balance = temp1 (== 0)	
temp1 = Bob's balance (== 5)	
temp1 = temp1 + 5 (== 10)	
INTERRUPTED BY THREAD 2	
	temp2 = Bob's balance (== 5)
	temp2 = temp2 - 5 (== 0)
	Bob's balance = temp2 (== 0)
	temp2 = Alice's balance (== 0)
	temp2 = temp2 + 5 (== 5)
	Alice's balance = temp2 (== 5)
	THREAD 2 COMPLETE
RESUME THREAD 1	
Bob's balance = temp1 (== 10)	
THREAD 1 COMPLETE	

It is also possible to achieve a negative balance. Suppose at the beginning of the function, the donor has enough money to participate in the transfer, so we pass the conditional check for sufficient

funds. Immediately after that, the donors balance is reduced below the required amount by some other running thread. Then the transfer will go through, resulting in a negative balance for the donor.

Sending two identical transfer(&alice, &bob, 2) may also cause unintended behavior, since the increment/decrement operations are not atomic (though it is arguably harder to exploit for profit).

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?

The entire function must be made atomic. One could do this by disabling interrupts for that period of time (if there is a single processor), or by acquiring a lock beforehand and releasing the lock afterwards. Alternatively, you could have a lock for each account. In order to prevent deadlocks, you will have to acquire locks in some predetermined order, such as lowerst account number first.

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?

Yes. In steps 3 and 4, the main thread and thread1 make no progress. They can only run to completion after thread2 resets the value to 0.

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

1. No call to test_and_set
2. 0
3. 1, 1, ..., 1
4. 1, 1, ..., 1
5. No call to test_and_set
6. 0
7. 0

Given this sequence of events, what will C print?

```
Child thread: 1
Parent thread: 1
Child thread: 2
```

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

No, this involves a ton of busy waiting.

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() {

    ________________________
}
```

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

```
void lock() {
    do {
        while (locked == 1);
    } while (test_and_set(locked));
}

void unlock() {
    locked = 0;
}
```

Yes. This scheme uses normal memory reads to spin while waiting for the lock to become free. Test_and_set is only used to try to get the lock when normal memory reads say it is free. Thus, expensive atomic memory operations happen less often.

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 cacheEntry {
    int videoID;
    char *video;

    // NEW FIELDS
    bool downloading;
    int users;
};

Lock cacheLock;
struct cacheEntry CACHE[1024];
```

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

```
char *getVideo(int videoID) {
    while (true) {
        bool wait_for_download = 0;
        do {
          lock_acquire(&cacheLock);

          for (int i = 0; i < 1024; i++) {
              if (CACHE[i].videoID == videoID) {
                  if (!CACHE[i].downloading) {
                      CACHE[i].users += 1;
                      lock_release(&cacheLock);
                      wait_for_download = 0;
                      return CACHE[i].video;
                  } else {
                      wait_for_download = 1;
                      lock_release(&cacheLock);
                      break;
                  }
              }
          }
        } while (wait_for_download);

        for (int i = 0; i < 1024; i++) {
            if (CACHE[i].users > 0) {
                continue;
            }

            // Why is this in a for loop? I don't know.
            if (CACHE[i].video != NULL) {
                free(CACHE[i].video);
            }
            CACHE[i].videoID = videoID;
            CACHE[i].users = 1;
            CACHE[i].downloading = true;
            lock_release(&cacheLock);
            CACHE[i].video = downloadVideoFromInternet(videoID);
            CACHE[i].downloading = false;
            return CACHE[i].video;
        }

        // All of the slots are in use. Just keep looping.
        lock_release(&cacheLock);
    }
}
```

Finally, implement the `doneWithVideo` function:

```
char *doneWithVideo(int videoID) {
    lock_acquire(&cacheLock);
    for (int i = 0; i < 1024; i++) {
        if (CACHE[i].videoID == videoID) {
            CACHE[i].users -= 1;
```

```
        }
    }
    lock_release(&cacheLock);
}
```