# Section 3: Synchronization

CS 162

February 12, 2021

## Contents

# 1 Vocabulary

- **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** - Synchronization primitives that provide mutual exclusion. Threads may acquire or release a lock. Only one thread may hold a lock at a time. If a thread attempts to acquire a lock that is held by some other thread, it will block at that line of code until the lock is released and it successfully acquires it. Implementations can vary.
- **semaphore** - Synchronization primitives that are used to control access to a shared variable in a more general way than locks. A semaphore is simply an integer with restrictions on how it an be modified:
  - When a sempahore is initialized, the integer is set to a specified starting value.
  - A thread can call **down()** (also know as **P**) to attempt to decrement the integer. If the integer is zero, the thread will block until it is positive, and then unblock and decrement the integer.
  - A thread can call **up()** (also known as **V**) to increment the integer, which will always succeed.

  Unlike locks, semaphores have no concept of "ownership", and any thread can call **down()** or **up()** on any semaphore at any time.
- **Condition Variable** - A synchronization variable that provides serialization (ensuring that events occur in a certain order). A condition variable is defined by:
  - a lock (a condition variable + its lock are known together as a **monitor**)
  - some boolean condition (e.g. `hello < 1`)
  - a queue of threads waiting for the condition to be true

  In order to access any CV functions **OR** to change the truthfulness of the condition, a thread must/should hold the lock. Condition variables offer the following methods:
  - **cv_wait(cv, lock)** - Atomically unlocks the lock, adds the current thread to **cv**'s thread queue, and puts this thread to sleep.
  - **cv_notify(cv)** - Removes one thread from **cv**'s queue, and puts it in the ready state.
  - **cv_broadcast(cv)** - Removes all threads from **cv**'s queue, and puts them all in the ready state.

  When a **wait()**ing thread is notified and put back in the ready state, it also re-acquires the lock before the **wait()** function returns.
  When a thread runs code that may potentially make the condition true, it should acquire the lock, modify the condition however it needs to, call notify() or broadcast() on the condition's CV, so waiting threads can be notified, and finally release the lock.
  Why do we need a lock anyway? Well, consider a race condition where thread 1 evaluates the condition $C$ as false, then thread 2 makes condition $C$ true and calls **cv.notify**, then 1 calls **cv.wait** and goes to sleep. Thread 1 might never wake up, since it went to sleep too late.
- **Hoare Semantics** - In a condition variable, wake a blocked thread when the condition is true and transfer control of the CPU and ownership of the lock to that thread immediately. This is difficult to implement in practice and generally not used despite being conceptually easier to deal with.

- **Mesa Semantics** - In a condition variable, wake a blocked thread when the condition is true with no guarantee on when that thread will actually execute. (The newly woken thread simply gets put on the ready queue and is subject to the same scheduling semantics as any other thread.) The implications of this mean that you must check the condition with a while loop instead of an if-statement because it is possible for the condition to change to false between the time the thread was unblocked and the time it takes over the CPU.

# 2 Synchronization

## 2.1 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.

## 2.2 Crowded Video Games

A recent popular game is having issues with its servers lagging heavily due to too many players being connected at a time. Below is the code that a player runs to play on a server:

```
void play_session(struct server s) {
    connect(s);
    play();
    disconnect(s);
}
```

After testing, it turns out that the servers can run without lagging for a max of up to 1000 players concurrently connected.

How can you add semaphores to the above code to enforce a strict limit of 1000 players connected at a time? Assume that a game server can create semaphores and share them amongst the player threads.

Introduce a semaphore for each server, initialized to 1000, to control the ability to connect to the game. A player will `down()` the semaphore **before** connecting, and `up()` the semaphore **after** disconnecting.

The order here is important - downing the semaphore after connecting but before playing means that there is no block on the `connect()` call, and upping the semaphore before disconnecting could lead to "zombie" players, who were pre-empted before disconnecting. Both of these cases mean that the limit of 1000 could be violated.

## 2.3 Hello World

Will this code compile/run?
Why or why not?

```
pthread_mutex_t lock;
pthread_cond_t cv;
int hello = 0;

void print_hello() {
    hello += 1;
    printf("First line (hello=%d)\n", hello);
    pthread_cond_signal(&cv);
    pthread_exit(0);
}

void main() {
    pthread_t thread;
    pthread_create(&thread, NULL, (void *) &print_hello, NULL);
    while (hello < 1) {
      pthread_cond_wait(&cv, &lock);
    }
    printf("Second line (hello=%d)\n", hello);
}
```

This won't work because the main thread should have locked the lock before calling pthread_cond_wait, and the child thread should have locked the lock before calling pthread_cond_signal. (Also, we never initialized the lock and cv.)

## 2.4 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.

## 2.5 Condition Variables

Consider the following block of code. How do you ensure that you always print out "Yeet Haw"? Assume the scheduler behaves with Mesa semantics. (Pseudocode is OK) You may only add lines, so the trivial answer of not checking the value of ben before printing is not correct.

```
int ben = 0;

void main() {
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, NULL);
    pthread_yield();
    if (ben == 1) {
        printf("Yeet Haw\n");
    } else {
        printf("Yee Howdy\n");
    }
    exit(0);
}

void *helper(void *arg) {
    ben += 1;
    pthread_exit(0);
}
```

```
int ben = 0;
//LOCK = L
//CONDVAR = C

void main() {
    pthread_t thread;
    //LOCK L ACQUIRE
    pthread_create(&thread, NULL, &helper, NULL);
    pthread_yield();
    //WHILE BEN != 1
        //CONDVAR C WAIT
    if (ben == 1) {
        printf("Yeet Haw\n");
    } else { ... }
    //LOCK L RELEASE
    exit(0);
}

void *helper(void *arg) {
    //LOCK L ACQUIRE
    ben += 1;
    //CONDVAR C NOTIFY
    //LOCK L RELEASE
    pthread_exit(0);
}
```

## 2.6 CS162 (Online) Office Hours

Suppose we want to use condition variables to control access to a CS162 (digital) office hours room for three types of people: students, TA's, and professors. A person can attempt to enter the room (or will wait outside until their condition is met), and after entering the room they can then exit the room. The follow are each type's conditions:

- Suppose professors get easily distracted and so they need solitude, with no other students, TA's, or professors in the room, in order to enter the room.
- TA's don't care about students inside and will wait if there is a professor inside, but there can only be up to 8 TA's inside (any more would clearly be imposters from CS161 or CS186).
- Students don't care about other students of TA's in the room, but will wait if there is a professor inside.
- Students and TAs are polite to professors, and will let a waiting professor in first.

To summarize the constraints:

- Professor must wait if anyone else is in the room
- TA must wait if there are already 8 TA's in the room
- TA must wait if there is a professor in the room or waiting outside
- Students must wait if there is a professor in the room or waiting outside

```
typedef struct lock { . . . }   lock        //  lock.acquire(), lock.release()
typedef struct cv { . . . } cv             //    cv.wait(&lock), cv.signal(), cv.broadcast()

#define TA_LIMIT    8
typedef struct  {
    lock    lock;
    cv  student_cv;
    int waitingStudents, activeStudents;
    cv  ta_cv, prof_cv;
    int waitingTas, waitingProfs;
    int activeTas, activeProfs;
}   room_lock;
```

```
/* mode = 0 for student, 1 for TA, 2 for professor */
enter_room(room_lock *rlock, int mode) {
    rlock->lock.acquire();
    if (mode == 0) {
        while ((rlock->activeProfs+rlock->waitingProfs) > 0) {
                rlock->waitingStudents++;
                rlock->student_cv.wait(&rlock->lock);
                rlock->waitingStudents--;
            }
        rlock->activeStudents++;
    } else if (mode == 1) {
        while((rlock->activeProfs+rlock->waitingProfs) > 0 || rlock->activeTas >= TA_LIMIT) {
            rlock->waitingTas++;
            rlock->ta_cv.wait(&rlock->lock);
```

```
            rlock->waitingTas--;
        }
        rlock->activeTas++;
    } else {
        while((rlock->activeProfs + rlock->activeTas + rlock->activeStudents) > 0) {
            rlock->waitingProfs++;
            rlock->prof_cv.wait(&rlock->lock);
            rlock->waitingProfs--;
        }
        rlock->activeProfs++;
    }
    rlock->lock.release();
}

exit_room(room_lock *rlock, int mode) {
    rlock->lock.acquire();
    if (mode == 0) {
            rlock->activeStudents--;
            if ((rlock->activeStudents + rlock->activeTas) == 0 &&  rlock->waitingProfs) {
                rlock->prof_cv.signal();
        }
    } else if (mode == 1) {
        rlock->activeTas--;
        if ((rlock->activeStudents  + rlock->activeTas) == 0 && rlock->waitingProfs) {
            rlock->prof_cv.signal();
        } else if (rlock->activeTas < TA_LIMIT && rlock->waitingTas) {
            rlock->ta_cv.signal();
        }
    } else {
        rlock->activeProfs--;
        if (rlock->waitingProfs) {
            rlock->prof_cv.signal();
        } else {
            if (rlock->waitingTas)
                rlock->ta_cv.broadcast();
            if (rlock->waitingStudents)
                rlock->student_cv.broadcast();
        }
    }
    rlock->lock.release();
}
```