Discussion 3: Mutual Exclusion, Condition Variables

February 14, 2025

Contents

1 Mutual Exclusion

Oftentimes, a multithreaded program will involve threads having shared states, meaning threads will share data with one another. After all, that is one of the main purposes of using a multithreaded program since all threads in the same process share the same address space. These are known as **cooperating threads**. If threads do not share any states, they are known as **independent threads**.

Cooperating threads pose a difficult challenge of synchronization. Since threads are not guaranteed to be scheduled in any order, there is no deterministic output which we can expect. This is known as a **race condition** where the output is dependent on the interleaving of operations of different threads.

To ensure cooperation between threads (i.e. **synchronization**), we can use **atomic operations** which always run to completion or not at all. Atomic operations are indivisible, meaning they cannot be stopped in the middle nor can the state be modified in the middle. On most machines, memory loads and stores are typically atomic operations.

A popular and basic way of synchronization is **mutual exclusion** where only one thread does a particular action at a time. In essence, one thread excludes the other while doing its task. The code that runs as part of mutual exclusion is known as the **critical section**. Critical sections allows for the illusion of each thread operating atomically on a shared state. To allow for maximum concurrency, critical sections should be as short as possible.

Locks

A **lock** is a synchronization variable that provides mutual exclusion. When one thread holds a lock, no other thread can hold it (i.e. other threads are excluded).

Locks provide two atomic operations.

Acquire / Lock
: Waits until the lock is free, then mark it as busy. When returning from this call, the thread is said to be holding the lock.

Release / Unlock
: Marks lock as free. When returning from this call, the thread no longer holds the lock. This can only be called if the thread is holding the lock.

When using locks, the code in between an acquire and release is the critical section. A typical workflow will involve acquiring the lock, performing memory operations on shared data, then releasing the lock.

Semaphores

A **semaphore** is a synchronization variable with a nonnegative integer.

Semaphores provide two atomic operations.

Down / Wait / P
: Waits for semaphore to become positive, then decrements it by 1.

Up / Post / V
: Increments the semaphore by 1.

Note that you are technically not allowed to examine the value of a semaphore or set it to a specific value. While most implementations will allow you to do this since it's just a member of a struct, this is not the use case of a semaphore.

While a semaphore can be used like a lock for mutual exclusion, it has another workflow of a scheduling constraint where one thread needs to wait for another thread to terminate. The two workflows are demonstrated below.

Mutual Exclusion

1. Initialize semaphore to 1.
2. Down semaphore when entering a critical section.
3. Up semaphore when exiting critical section.

Scheduling

1. Initialize semaphore to 0.
2. Down semaphore in the waiting thread.
3. After finishing desired work, up semaphore in the working thread.

1.1 Synchronization Stuffs

1. 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 creates 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.

```c
typedef struct account {
  /* ... */
  int balance;
  /* ... */
} account_t;

void transfer(account_t *donor, account_t *recipient, float amount) {
  assert (donor != recipient);
  if (donor->balance < amount) {
    printf("Insufficient funds.\n");
    return;
  }
  donor->balance -= amount;
  recipient->balance += amount;
}
```

cgfc.c

Describe how a malicious user might exploit some unintended behavior. What changes could you make to the CGFC to defend against exploits?

We observe that the money transfer is not an atomic instruction, meaning we can break down a money transfer into further lines.

```
donor->balance -= amount;
```

```
temp = donor->balance;
temp = temp - amount;
donor->balance = temp;
```

```
recipient->balance += amount;
```

```
temp = recipient->balance;
temp = temp + amount;
recipient->balance = temp;
```

While the `temp = temp - amount` and `temp = temp + amount` are still not technically atomic operations, this level of granularity suffices to see race conditions.

Let's say we have two accounts of `account_t*` named `alice` and `bob`. We send two requests: `transfer(alice, bob, 5)` and `transfer(bob, alice, 5)`. Let's say the following interleaving of threads occurs.

```
/* transfer(alice, bob, 5) */          /* transfer(bob, alice, 5) */

temp1 = alice->balance;
temp1 = temp1 - 5;
alice->balance = temp1;
temp1 = bob->balance;
temp1 = temp1 + 5;
                                       temp2 = bob->balance;
                                       temp2 = temp2 - 5;
                                       bob->balance = temp2;
                                       temp2 = alice->balance;
                                       temp2 = temp2 + 5;
                                       alice->balance = temp2;
bob->balance = temp1;
```

This will result in `bob->balance` being 10 while `alice->balance` being 5, essentially making money out of thin air.

Another possible race condition achieves a negative balance. If there is a thread switch after the `donor->balance < amount` check succeeds, the `donor->balance` may fall below `amount`. However, the transaction will still go through, resulting in money transfer that the `donor` does not have.

The solution is to simply make the entire function a critical section. A lock can be acquired at the first line of the function and released at the last line.

2. 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:

```
1 void play_session(struct server* s) {
2   connect(s);
3   play();
4   disconnect(s);
5 }
```

game_server.c

After testing, it turns out that the servers can run without lagging for a max of up to 1000 players connected concurrently. How can you use synchronization to enforce a strict limit of 1000 players connected at a time? Assume that a game server can create synchronization variables 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 preempted before disconnecting. Both of these cases mean that the limit of 1000 could be violated.

1.2 Atomic Punch

1. Other than implementing synchronization primitives, atomic instructions can be useful in implementing a synchronized algorithm.

   The following piece of code makes use of pthreads to parallelize summing up an array of integers.

```c
int sum = 0;

void* adder(void* arg) {
    int term = (int) arg;
    do {
        _________________;
        _________________;
    } while (_________________________);
    return NULL;
}

int main(void) {
    int n = 100;
    pthread_t threads[n];
    int terms[n];
    init_array(terms); // Initializes the array.

    for (int i = 0; i < 100; i++)
        pthread_create(threads + i, NULL, adder, terms[i]);

    for (int i = 0; i < 100; i++)
        pthread_join(threads[i], NULL);

    printf("The total is %d\n", sum);
}
```

adder.c

   Make use of `compare_and_swap` to complete the implementation.

```c
int old = sum;
int new = old + term;
while (!compare_and_swap(&sum, old, new));
```

adder.c

1.3 Shared Data

This problem is designed to help you implement the `wait` syscall in Project User Programs by implementing a wait functionality that allows the main thread to wait for another thread to finish writing some data.

Assume you don't have access to `pthread_join` for this problem. Instead, you're going to use synchronization primitives, namely locks and semaphores.

You're going to implement three functions to initialize the shared struct and synchronize the 2 threads. `initialize_shared_data` will initialize `shared_data`. `wait_for_data` (called by the main thread) will block until the data is available. `save_data` (called by the other thread) will write 162 to the `data` member of `shared_data`. Another requirement is that the shared data needs to be freed once as soon as it is no longer in use. When using synchronization primitives, the critical section should be as few lines as possible while still ensuring a race free environment.

Consider you're given the following code.

```c
typedef struct shared_data {
  sem_t semaphore;
  pthread_mutex_t lock;
  int ref_cnt;
  int data;
} shared_data_t;

void initialize_shared_data(shared_data_t* shared_data) {
  ________________________;
  ________________________;
  ________________________;
  shared_data->data = -1;
}

int wait_for_data(shared_data_t* shared_data) {
  ________________________;
  int data = shared_data->data;
  ________________________;
  int ref_cnt = --shared_data->ref_cnt;
  ________________________;
  if (ref_cnt == 0)
    ________________________;
  return data;
}

void* save_data(void* shared_pg) {
  ________________________;
  shared_data->data = 162;
  ________________________;
  ________________________;
  int ref_cnt = --shared_data->ref_cnt;
  ________________________;
  if (ref_cnt == 0)
    ________________________;
  return NULL;
}

int main() {
  void *shared_data = malloc(sizeof(shared_data_t));
  initialize_shared_data(shared_data);
  pthread_t tid;
  int error = pthread_create(&tid, NULL, &save_data, shared_data);
  int data = wait_for_data(shared_data);
  printf("Parent: Data is %d\n", data);
  return 0;
}
```

wait_shared.c

1. Explain the purpose of each member in the `shared_data_t` struct.

> `semaphore` is used to implement the scheduling workflow. A waiting thread downs the semaphore while the thread that saves the data ups the semaphore.
>
> `lock` allows for mutual exclusion on members in the struct that can be modified by both threads.
>
> `ref_cnt` allows for reference counting, which is an indicator for how many threads still hold a reference to this struct. Once the reference count reaches 0, we can safely deallocate the struct

from memory.

`data` is the actual data.

2. Fill in the missing lines in `initialize_shared_data` to correctly initialize the members of `shared_data`.

`semaphore` needs to be initialized to 0. When the main thread is waiting for the data to be updated it needs to block until the other thread ups the semaphore. There's no guarantee of thread execution order, meaning the main thread could end up downing the semaphore first.

`lock` is initialized with default attributes which is specified by the second argument as `NULL`.

`ref_cnt` is initialized to 2 since there are two threads who will have access to the shared data.

```c
void initialize_shared_data(shared_data_t* shared_data) {
  sem_init(&shared_data->semaphore, 0, 0);
  pthread_mutex_init(&shared_data->lock, NULL);
  shared_data->ref_cnt = 2;
  shared_data->data = -1;
}
```

wait_shared_initialize.c

3. Fill in the missing lines in `wait_for_data` for the main thread to correctly wait for the other thread until the data is updated.

The main thread waits for the other thread to finish updating by downing the semaphore. As initialized, the semaphore will start at 0, meaning the main thread will wait until its value is 1 before it can down it.

Since `ref_cnt` can be modified by either thread concurrently, it needs to be put in a critical section by surrounding it with locks.

Finally, if `ref_cnt` is 0, we need to make sure to free the `shared_data`.

```c
int wait_for_data(shared_data_t* shared_data) {
  sem_wait(&shared_data->semaphore);
  int data = shared_data->data;
  pthread_mutex_lock(&shared_data->lock);
  int ref_cnt = --shared_data->ref_cnt;
  pthread_mutex_unlock(&shared_data->lock);
  if (ref_cnt == 0)
    free(shared_data);
  return data;
}
```

wait_shared_wait.c

4. Fill in the missing lines in `save_data` for the other thread to correctly update the data.

Since this function is called through `pthread_create`, we need to cast the argument `shared_pg` to a `shared_data_t*` type.

After we set the desired value of `shared_data->data`, we up the the semaphore, so the parent no longer waits. Keep in mind this does not guarantee the parent will run automatically since as said before, there are no guarantees of exeuction orders.

Just like we did in `wait_for_data`, we need to decrement the `ref_cnt` in a race free manner and make sure to free `shared_data` if necessary.

```c
void* save_data(void* shared_pg) {
  shared_data_t* shared_data = (shared_data_t*) shared_pg;
  shared_data->data = 162;
  sem_post(&shared_data->semaphore);
  pthread_mutex_lock(&shared_data->lock);
  int ref_cnt = --shared_data->ref_cnt;
  pthread_mutex_unlock(&shared_data->lock);
  if (ref_cnt == 0)
    free(shared_data);
  return NULL;
}
```

wait_shared_save.c

5. Why does `shared_data->data` not need to be surrounded by locks when reading and writing to it in `wait_for_data` and `save_data`?

The semaphore guarantees that the main thread will never access `shared_data->data` before the other thread sets it. Similarly, `shared_data->data` is never modified by the other thread afterwards, so the main thread is guaranteed to have the correct value by the time it is unblocked from the semaphore.

2 Condition Variable

Condition variables are synchronization variables that let a thread efficiently wait for a change to a shared state. They store a queue of threads such that the waiting threads are allowed to sleep inside the critical section which is in contrast to other synchronization variables like semaphores.

Condition variables are used in conjunction with a lock which together form a **monitor**.

Condition variables provide the following operations. For all these methods, the thread calling them must be holding the lock.

Wait
Atomically releases lock and suspends execution of calling thread.

Signal
Wake up the next waiting thread in the queue.

Broadcast
Wake up all waiting threads in the queue.

Infinite Synchronized Buffer

Consider an infinite synchronized buffer problem of vending machines where there's a producer and consumer. "Infinite" refers to the fact that the machine has no limit on how much coke it can hold.

```
Lock bufferLock;
ConditionVar bufferCV;

Producer() {
  bufferLock.acquire();
  put 1 coke in machine
  bufferCV.signal();
  bufferLock.release();
}

Consumer() {
  bufferLock.acquire();
  while (machine is empty)
    bufferCV.wait(bufferLock);
  take 1 coke out
  bufferLock.release();
}
```

By using condition variables, we can avoid busy waiting inside a critical section.

Semantics

Different semantics define signal and wait differently.

When a thread is signaled using **Hoare** semantics, the ownership of the lock is immediately transferred to the waiting thread. Furthermore, the execution of this thread resumes immediately. After this thread releases the lock, ownership of the lock is transferred back to the signaling thread. As a result, signaling in Hoare semantics can be thought of as an atomic operation.

On the other hand, **Mesa** semantics makes no guarantees about the execution order when a thread is signaled.

This leads to a subtle but important difference in the code. Using the same bounded buffer example as before,

Hoare

```
if (machine is empty)
  bufferCV.wait(bufferLock);
take 1 coke out
```

Mesa

```
while (machine is empty)
  bufferCV.wait(bufferLock);
take 1 coke out
```

We can use a if statement for Hoare semantics because we're guaranteed an execution order between the waiting and signaling thread. However, that is not the case for Mesa semantics, meaning between the time the waiting thread is signalled and it actually executes, some other thread could have run in between and rendered the condition false again, so a while loop is necessary.

2.1 Condition Check

1. Will this program compile/run?

```c
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);
}
```

cv_hello.c

> This program will not run because the thread needs to be holding a lock before performing a condition variable operation like wait or signal.
>
> Moreover, the lock and condition variable was never initialized, which would lead to undefined behavior.

2. Fill in the blanks such that the program always prints "Yeet Haw". Assume the system behaves with Mesa semantics.

```c
int ben = 0;
_____________________;
_____________________;

void *helper(void *arg) {
  _____________________;
  ben += 1;
  _____________________;
  _____________________;
  pthread_exit(NULL);
}

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");
  ________________________;
}
```

cv_howdy.c

```
int ben = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;

void *helper(void *arg) {
  pthread_mutex_lock(&lock);
  ben += 1;
  pthread_cond_signal(&cv);
  pthread_mutex_unlock(&lock);
  pthread_exit(NULL);
}

void main() {
  pthread_t thread;
  pthread_create(&thread, NULL, &helper, NULL);
  pthread_yield();
  pthread_mutex_lock(&lock);
  while (ben != 1)
    pthread_cond_wait(&cv, &lock);
  if (ben == 1)
    printf("Yeet Haw\n");
  else
    printf("Yee Howdy\n");
  pthread_mutex_unlock(&lock);
}
```

cv_howdy_sol.c

2.2 Office Hours Queue

Suppose we want to use condition variables to control access to a CS162 (digital) office hours room for three types of people: students, TAs, 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, TAs, or professors in the room, in order to enter the room.
- TAs don't care about students being inside and will wait if there is a professor inside, but there can only be up to 9 TAs inside (any more would clearly be imposters from CS161 or CS186).
- Students don't care about other students or TAs being in the room, but will wait if there is a professor.
- 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 9 TAs 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 room {
  pthread_mutex_t lock;
  pthread_cond_t student_cv;
  int waiting_students, active_students;
  pthread_cond_t ta_cv, prof_cv;
  int waiting_tas, active_tas;
  int waiting_profs, active_profs;
} room_t;

enter_room(room_t* room, int mode) {
  pthread_mutex_lock(&room->lock);
  if (mode == 0) {
    ________________________;
    ________________________;
    ________________________;
    ________________________;
    ________________________;
    room->active_students++;
  } else if (mode == 1) {
    ________________________;
    ________________________;
    ________________________;
    ________________________;
    ________________________;
    room->active_tas++;
  } else {
    ________________________;
    ________________________;
    ________________________;
    ________________________;
    ________________________;
    room->active_profs++;
  }
  pthread_mutex_unlock(&room->lock);
}

exit_room(room_t* room, int mode) {
  pthread_mutex_lock(&room->lock);
  if (mode == 0) {
    room->active_students--;
    ________________________;
    ________________________;
    ________________________;
  } else if (mode == 1) {
    room->active_tas--;
    ________________________;
    ________________________;
    ________________________;
    ________________________;
    ________________________;
  } else {
```

```
    room->active_profs--;
    ______________________;
    ______________________;
    ______________________;
    ______________________;
    ______________________;
    ______________________;
    ______________________;
    ______________________;
    ______________________;
  }
  pthread_mutex_unlock(&room->lock);
}
```

oh.c

1. Fill in `enter_room`.

```
enter_room(room_t* room, int mode) {
  pthread_mutex_lock(&room->lock);
  if (mode == 0) {
    while (room->active_profs + room->waiting_profs > 0) {
      room->waiting_students++;
      pthread_cond_wait(&room->student_cv, &room->lock);
      room->waiting_students--;
    }
    room->active_students++;
  } else if (mode == 1) {
    while (room->active_profs + room->waiting_profs > 0 || room->active_tas >= TA_LIMIT
    ) {
      room->waiting_tas++;
      pthread_cond_wait(&room->ta_cv, &room->lock);
      room->waiting_tas--;
    }
    room->active_tas++;
  } else {
    while (room->active_profs + room->active_tas + room->active_students > 0) {
      room->waiting_profs++;
      pthread_cond_wait(&room->prof_cv, &room->lock);
      room->waiting_profs--;
    }
    room->active_profs++;
  }
  pthread_mutex_unlock(&room->lock);
}
```

oh_enter.c

2. Fill in `exit_room`.

```
exit_room(room_t* room, int mode) {
  pthread_mutex_lock(&room->lock);
  if (mode == 0) {
    room->active_students--;
    if (room->active_students + room->active_tas == 0 && room->waiting_profs > 0) {
      pthread_cond_signal(&room->prof_cv);
```

```
    }
  } else if (mode == 1) {
    room->active_tas--;
    if (room->active_students + room->active_tas == 0 && room->waiting_profs > 0) {
      pthread_cond_signal(&room->prof_cv);
    } else if (room->active_tas < TA_LIMIT && room->waiting_tas && room->waiting_profs
    == 0) {
      pthread_cond_signal(&room->ta_cv);
    }
  } else {
    room->active_profs--;
    if (room->waiting_profs) {
      pthread_cond_signal(&room->prof_cv);
    } else {
      if (room->waiting_tas)
        pthread_cond_broadcast(&room->ta_cv);
      if (room->waiting_students)
        pthread_cond_broadcast(&room->student_cv);
    }
  }
  pthread_mutex_unlock(&room->lock);
}
```

oh_exit.c