## Contents

1 Vocabulary 2  

2 Condition Variables 4  
   2.1 Yeet Haw 4  
   2.2 Office Hours 5  

3 Sockets 7  
   3.1 Multi-threaded Echo Server 7
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 semaphore 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 Condition Variables

2.1 Yeet Haw

Consider the following block of code. Add some pseudocode to ensure that the program always print out “Yeet Haw”? Assume the scheduler behaves with Mesa semantics.

```c
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);
}
```

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

void main() {
    pthread_t thread;
    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);
}
```
2.2 Office Hours

Suppose we want to use condition variables to control access to a CS162 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 following are each type’s conditions:

- 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

```c
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 room_lock{
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->activeProfs--;
    }
    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();
}
3 Sockets

3.1 Multi-threaded Echo Server

Write a server implementation that uses multiple threads in a single process. Each connection is handled in its own thread, and threads should be allowed to handle connections concurrently. For simplicity assume `read()` and `write()` do not return short.

#define BUF_SIZE 1024

struct addrinfo *setup_address(char *port) {
    struct addrinfo *server;
    struct addrinfo hints;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;

    int rv = getaddrinfo(NULL, port, &hints, &server);
    if (rv != 0) {
        printf("getaddrinfo failed: %s\n", gai_strerror(rv));
        return NULL;
    }
    return server;
}

void *serve_client(void *client_socket_arg) {
    int client_socket = (int)client_socket_arg;
    char buf[BUF_SIZE];
    ssize_t n;

    while ((n = read(client_socket, buf, BUF_SIZE)) > 0) {
        buf[n] = '\0';
        printf("Client Sent: %s\n", buf);
        if (write(client_socket, buf, n) == -1) {
            close(client_socket);
            pthread_exit(NULL);
        }
    }
    close(client_socket);
    pthread_exit(NULL);
}

int main(int argc, char **argv) {
    if (argc < 2) {
        printf("Usage: %s <port>\n", argv[0]);
        return 1;
    }
struct addrinfo *server = setup_address(argv[1]);
if (server == NULL) {
    return 1;
}
int server_socket = socket(server->ai_family,
    server->ai_socktype, server->ai_protocol);
if (server_socket == -1) {
    return 1;
}
if (bind(server_socket, server->ai_addr,
    server->ai_addrlen) == -1) {
    return 1;
}
if (listen(server_socket, 1) == -1) {
    return 1;
}
while (1) {
    int connection_socket = accept(server_socket, NULL, NULL);
    if (connection_socket == -1) {
        perror("accept");
        pthread_exit(NULL);
    }

    pthread_t handler_thread;
    int err = pthread_create(&handler_thread, NULL,
        serve_client, (void *)connection_socket);
    if (err != 0) {
        printf("pthread_create: %s\n", strerror(err));
        pthread_exit(NULL);
    }
    pthread_detach(handler_thread);
}
pthread_exit(NULL);