Section 4: Synchronization, Sockets

CS 162

September 25, 2020

Contents

1 Vocabulary

- **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.

- **TCP** - Transmission Control Protocol (TCP) is a common L4 (transport layer) protocol that guarantees reliable in-order delivery. In-order delivery is accomplished through the use of sequence numbers attached to every data packet, and reliable delivery is accomplished through the use of ACKs (acknowledgements).

- **Socket** - Sockets are an abstraction of a bidirectional network I/O queue. It embodies one side of a communication channel, meaning that two must be required for a communication channel to form. The two ends of the communication channel may be local to the same machine, or they may span across different machines through the Internet. Most functions that operate on file descriptors like read() or write() work on sockets. but certain operations like lseek() do not.

2 Synchronization

2.1 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.)

2.2 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.

2.3 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);
}
```

2.4 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();
}
```

3 Socket Programming

3.1 Multi-threaded Echo Server

Please look at the three versions of server code provided with Lecture 8. The first version uses a single process and single thread, the second version sequentially handles each client connection in a child process, and the third version allows child processes to handle connections concurrently.

Write a fourth version of the 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);
}
```