For the threading part of this assignment, originally I created two threads to help speed up processing. They were namely Mthread and Cthread. Mthread stood for “Moving platform thread” and Cthread stood for “Collision thread”. As I went, I refined this a bit. In essence, main handled user input, and the collisions that occur from user input. Mthread handled ALL moving platforms, and Cthread handled physics, or “effects on the player that happen without user input”. These include things such as gravity, moving along with a moving platform, etc. I implemented this, but it wasn’t quite right. While I was (mostly) thread-safe, my synchronization was pretty off. Until I eventually got to part four and did networking. Basically I made it so that the cthread had to wait until I received updated platforms from the server, that way new platforms wouldn’t be coming in while cthread was processing collisions. I noticed a problem where polling for events would halt the progress of the server and thus slow down all clients. So I made a thread for that, called “EventThread”, which ended up being a failure due to the fact that opening a window in a separate thread while also being able to draw from main proved to be seemingly impossible. So I stuck with my two threads, and they seemed to work pretty well, even with networking, though they didn’t end up being as separate of processes as I hoped.

For the timeline part, I implemented a rather simple design. We discussed in class the process of “Frame limiting”. I ended up doing something similar. Basically the process has its own tic rate, and it tries to meet that tic rate, but really it tries not to update too fast. For example, if the tic size is 8, the process will keep checking back until 8 milliseconds has gone by. Then it will run the process. If that process took longer than 8 milliseconds, say 16 milliseconds, that is okay because the program knows that and adjusts. For example my PC was able to make a tic size of 8 every time, but my crummy old laptop couldn’t make it, and so fluctuated between 16 and 24 milliseconds, but the character or platforms would adjust if it couldn’t make it. Anyway,

my timeline was able to sync with real time using my method “getRealTicLength”, which would return the current real time length of a tic. That way you could put all of your speeds in terms of pixels per second. For example on a tic rate of 8, with a speed of 100, my character would move at a speed of .008 * 100 pixels per tic (A1). Same with gravity, moving platform speed, etc.

For speeding up and slowing down, I simply added a “scale” variable to my getRealTicLength. If the scale was .5 it would multiply that value by .5, and whichever process was accessing it would think that the object simply moved at that rate (A2).

For pausing, I instead focused on my getTime method. Originally I considered simply changing the scale to zero, but that wouldn’t help because while that would freeze the movement, it wouldn’t unpause correctly. Instead, I did what we discussed in class, and kept track of the total elapsed time and then subtracted that from my return value from getTime. That’s why it was important for elapsed_paused_time to be the TOTAL elapsed paused time, not just the most recent period. So then, when unpausing, it would appear to the processes that no tics had gone by while paused, even though they actually had (A3).

For networking, aside from some minor confusion (I thought we had to use pub-sub OR req-rep) my plan pretty much worked correctly immediately. As I read over the documentation, I realized that using solely pub-sub or solely req-rep would not be good enough. That’s because ideally the server would be a publisher and the client would be the subscriber. But that wouldn’t work here because the server needed various pieces of information from the clients, such as the positions of their characters. So, the key was to combine req-rep and pub-sub, and I figured the best way to do it was to have the client send an initial message to the server, basically just telling it “Hey I connected” (A4). That way the server could then keep track of how many iterations each client was on.

However, it quickly became clear that this concept would not be working for part four of the assignment. For part four, I realized it wouldn't be good enough to simply tell the publisher I had connected, I needed the client to repeatedly send it information about where its character is at any given iteration. So I modified my original design, and had my server check back in with my clients every iteration. It would also be keeping an eye out for any new clients that wanted to join (A5). Then it would publish all of the information it received (the positions of each client) as well as the information it had changed (the positions of the moving platforms). I did this using strings, as the clients would already have information about each platform and character, and so would only need a few details (A6). In the future (especially if I want to use conflate), I may need to combine these strings so that the clients get these all in one go.

From the client's end, it receives the information. Using synchronization, I set it up so that cthread would run only after it received the OK from main after it received the updated platforms. This had mixed success. On the one hand, my collision detection is always working off of the most updated information, and can only go once it's received word from the server. This could help with things like anti cheat but also just generally making sure your information is accurate, as well as preventing new information getting sent while cthread is running. However, it also means that the server might not have the most accurate information about the client. For example, let's say that the client collides with a platform, and sends that to the server. Then, afterwards, it fixes it and all of the clients draw. Well now none of the clients have the updated position of the collided client, and it can appear that the client is stuck in a platform when it actually isn't. However, this only happens sometimes, and with a lower tic rate (it isn't noticeable for me on 8, but is on 16 tic size). This is because sometimes when jumping on a platform, the collision will happen when the client sends information, and sometimes it will

happen when the server sends information. Or sometimes it will only barely collide, or fully collide, and it will look fine. This will hopefully be fixed in the next homework because the cthread will be able to update whenever it wants because receiving from the server will be handled in a separate thread. Also having a separate thread will allow the tic rate to be higher in theory.

For part five, I didn't do much for it. The only thing I will mention is that slowing down or speeding up my clients already works as intended thanks to my getRealTicLength method. It only slows down the one client's character and nobody else's, because it doesn't affect the rate at which it receives messages. Now if you slowed down the tic rate for one client, that would in fact affect all other clients, which I think is what part five is getting at.

KNOWN BUGS:

Unlike Homework One, there are uncommon, but noticeable bugs here. I thought I should include a section here to talk about bugs my program has and either why I haven't fixed them yet or what I think/know the issue is. None of them are technically "bugs" in that they violate the requirements for the assignment, but they "bug" me. A lot.

First, platform "slidiness". You may notice during testing that when standing on a horizontal moving platform, the character will very slightly (depending on tic rate and speed) slide to the left or right when the platform switches directions. I've tried to fix this numerous different ways and haven't come up with a good answer. Near as I can tell, it's (probably) a problem with threading or with the way I move the character on a moving platform. I use a method in MovingPlatform called "getLastMove" in order to know how much to move my character on a moving platform. I have to use this to circumvent the fact that these things can

change scale and so don't have an exact speed. They could also move more or less depending on the tic rate. The problem is, if this doesn't get updated consistently, every time, before cthread runs, then it will execute an extra move or two in the wrong direction.

This next bug goes along with the first one. On a vertical moving platform, the character gets stuck in the platform a bit and jitters around. Part of the jittering is that SFML seems to do a much better job at moving the character to the left or right, but not so good at moving it up or down. I know this because moving to the left or right looks fine, but moving up looks jittery, even when not on a platform. Though it could also be my computer, as my laptop seems to have less of a problem with this. The other problem is what I mentioned earlier; my networking setup. I will be changing my networking quite a bit in the next homework, so I plan on fixing it then.

Next, resizing or moving the window causes the character to glitch through the floor. On this front, I know the exact cause behind it. The problem is that my game loop is in the "Poll Event" loop the entire time the resize or move is happening, which means it fails to get updates from the server, which means it fails to synchronize with cthread, which means that when it is done, cthread updates much more than intended and bypasses colliding with the platform. This can also mess with server processes, because halting one client halts the entire server. This can be fixed, but it will happen in my next homework, when I make separate threads for talking with the server. As I mentioned, I attempted to do the complement of this, and move my Poll Event loop to a separate thread, but was met with technical difficulties when attempting this.

Lastly is disconnecting. For this one, I have no real answers because I only implemented disconnecting recently to make it easier on myself when testing and on the grader, since it wasn't a requirement for this homework. The problem is how rarely it occurs because most of the time disconnecting works as intended. The character disappears from all other clients (It's still there in

the memory but not being drawn or updated), and everything continues as normal. I have a small suspicion that it only happens at a very specific time such as to create a deadlock with one or more of the other clients (or the server). Perhaps now that we've actually covered deadlocking in Operating Systems, I'll be able to fix it next homework. Or maybe multithreading my networking will fix it too. By the way I'm specifically talking about part FOUR. Part three never crashes on disconnects.

Appendix

A1:

```
    ticLength = line->getRealTicLength();

    { // anonymous inner block to manage scop of mutex lock
        //Take ownership of the lock and lock it
        std::unique_lock<std::mutex> cv_lock(*this->_mutex1);
        _condition_variable1->wait(cv_lock);
        *busy = false;
    }
    *upPressed = false;
    //Get gravity as a function of time
    float gravity = character->getGravity() * ticLength * (currentTic - tic);
    character->move(0.f, gravity);
```

A2:

```
float Timeline::getRealTicLength() {
    return anchor ? tic * anchor->getRealTicLength() : scale * sf::milliseconds(1).asSeconds();
}
```

A3:

```
int64_t Timeline::getTime() {
    std::lock_guard<std::recursive_mutex> lock(timeMutex);
    //If we are paused, return the last paused time. This will make it appear as though no time is passing.
    if (paused) {
        return last_paused_time;
    }
    //If we are not the anchor, refer to the anchor's start time.
    if (anchor) {
        return (anchor->getTime() - start_time) / tic;
    }
    //If we are the anchor, use the library to get the correct time.
    else {
        return (duration_cast<milliseconds>(system_clock::now().time_since_epoch()).count() - start_time - elapsed_paused_time) / tic;
        //Need to subtract by "elapsed_paused_time" so that when unpausing it looks like no time has passed.
    }
}
```

A4:

```
while (true) {
    //  Wait for next request from client
    zmq::message_t reply;
    zmq::recv_result_t received(repSocket.recv(reply, zmq::recv_flags::dontwait));
    if ( ( received.has_value() && ( EAGAIN != received.value() ) ) ) {
        zmq::message_t reply;
        memcpy(reply.data(), "Connection Received", 20);
        repSocket.send(reply, zmq::send_flags::none);

        Client newClient;
        newClient.id = ++numClients;
        newClient.offset = iterations - 1;
        clients.push_front(newClient);
    }
```

A5:

```
    //We are expecting numClients to connect, so we loop for them
    for (int i = 0; i < numClients; i++) {

        //Receive message from client
        zmq::message_t update;
        zmq::recv_result_t received(repSocket.recv(update, zmq::recv_flags::none));
        char* current = (char*)update.data();

        CharStruct newCharacter;
        char status; // 'd' for disconnect, 'c' for connect
        //Get client info
        int matches = sscanf_s(current, "%d %f %f %c", &(newCharacter.id), &(newCharacter.x), &(newCharacter.y), &status, 1);

        //While this loop is supposed to be for previously connected clients, a new one might slip in. Set them up!
        if (newCharacter.id == -1 && status == 'c') {
            newCharacter.id = numCharacters;
            //Only if we have room
            if (numCharacters < 10) {
                characters[numCharacters] = newCharacter;
                numClients++;
                numCharacters++;
            }
        }
```

A6:

```
        //TODO: Cmbine these strings?

        //Construct platform position string
        char platRtnString[MESSAGE_LIMIT] = "";
        for (MovingPlatform *i : movings) {
            char platString[MESSAGE_LIMIT];
            sprintf_s(platString, "%f %f ", i->getPosition().x, i->getPosition().y);
            strcat_s(platRtnString, platString);
        }
        //Send platform information
        zmq::message_t sendPlatforms(strlen(platRtnString) + 1);
        memcpy(sendPlatforms.data(), platRtnString, strlen(platRtnString) + 1);
        pubSocket.send(sendPlatforms, zmq::send_flags::none);

        //Construct character position string
        char charRtnString[MESSAGE_LIMIT] = "";
        for (int i = 0; i < numCharacters; i++) {
            char charString[MESSAGE_LIMIT];
            sprintf_s(charString, "%d %f %f ", characters[i].id, characters[i].x, characters[i].y);
            strcat_s(charRtnString, charString);
        }
        //Send character information
        zmq::message_t sendCharacters(strlen(charRtnString) + 1);
        memcpy(sendCharacters.data(), charRtnString, strlen(charRtnString) + 1);
        pubSocket.send(sendCharacters, zmq::send_flags::none);
    }
```