**Basic design**

| **Q:** |  | **How do I start this assignment?** |
| --- | --- | --- |
|  | | |
| **A:** |  | Forget about handling <Ctrl+C> for now. I would have the main thread create 4 child threads and immediately join with all of them. When all 4 child threads are dead, the main thread can print statistics and exit the program.  The 4 child threads will collaborate to update the shared data structures (Q1, Q2, token count, and anything that's shared by these threads), which is protected by a single mutex (with a single condition variable). In order to gain access (read or write) to any of these data structures, a thread have to lock the mutex, do something to the data structure, and unlock the mutex. If a thread has to wait for **anything**, you must not do busy-waiting and must have your thread go to sleep.  You can imagine 4 people trying to accomplish the tasks described by the spec and each person must follow the protocol specified in the spec. What does each person/thread have to do to accomplish the tasks so that things will always work according to the spec? The packet arrival person/thread needs to generate packet in the specified way, then create a packet and add it to Q1 (and in some cases, it needs to then remove this packet from Q1 and move it into Q2). If it moves a packet into Q2, it needs to wake up a sleeping server thread (the spec says that you must call pthread\_cond\_broadcast() and not pthread\_cond\_signal()). The token depositing person/thread needs to generate tokens in the specified way, then add a token to the token bucket (i.e., just increment the token count), and in some cases, it needs to remove a packet from Q1 and move it into Q2. If it moves a packet into Q2, it needs to wake up a sleeping server thread using pthread\_cond\_broadcast(). Of course, if you call pthread\_cond\_broadcast() and both server threads are sleeping in the condition variable queue, both server threads will be woken up (and they will return from pthread\_cond\_wait()). A server person/thread checks if Q2 is empty or not. If Q2 is empty, it goes to sleep (in a CV queue). If Q2 is not empty, it must remove a packet from Q2, unlock the mutex, and simulate the transmission of the packet by calling usleep() to sleep the service time of the packet.  Please remember that whenever you need to use any of the shared data, you must have the mutex locked. What about when the packet arrival thread sleeps to wait for the right time to generate the next packet or when the token depositing thread sleeps to wait for the right time to generate the next token? Well, you should make sure that they go to sleep with the mutex unlocked. What about when a server thread sleeps to simulate the transmission of the packet? You should also make sure that it goes to sleep with the mutex unlocked. |

| **Q:** |  | **No, I mean how do I start this assignment from scratch?** |
| --- | --- | --- |
|  | | |
| **A:** |  | Start with your "warmup1.c" and rename it to "warmup2.c" and delete all the code you don't need and fix your Makefile. In main(), just create 4 threads and then join with them. Write the first procedure for the packet thread and the token thread and share the same first procedure for your server threads and make sure that one of the server can identify itself as S1 and the other server can identify itself as S2. Have all these 4 threads just self-terminate. Make sure this works and that you understand how this works and why this works. I think that would be a good starting point. Make sure it works perfectly!  Then set [n](http://merlot.usc.edu/cs402-m23/projects/warmup2/index.html#commandline) to 1 (just hard-code it) and modify the first procedure of all your threads to work with n=1 and use the default values for all other simulation parameters. This means that lambda = 1, mu = 0.35, r = 1.5, B = 10, and P = 3 and you should use global variables for all these values. How should your simulation go in this case? Since packet inter-arrival time is one second, packet p1 should arrive soon after 1 second into the simulation. What about the arrival of tokens? Since the token inter-arrival time is 666.667ms, token t1 should arrive before the first packet. Therefore, when p1 arrives, you should create a packet object and append it to Q1. At this point, the packet thread should see that there is not enough token in the token bucket and p1 is left at the head of Q1. The packet arrival thread should self-terminate because all packets have arrived.  When token t2 arrives, the token bucket should then have 2 tokens. But that's not enough to move p1 to Q2. When token t3 arrives, the token bucket should then have 3 tokens and the token thread should remove p1 from Q1 and then append p1 to Q2 and broadcast the CV to wake up all the server threads. The token arrival thread should self-terminate because all packets have arrived and Q1 is empty.  Whichever the server thread that wakes up first should dequeue p1 from Q2 and start transmitting p1 by sleeping for 2857ms. The server thread that wakes up next should self-terminate because all packets have arrived and both Q1 and Q2 are empty. It should also broadcast the CV because it doesn't know what the other server thread is doing. When the server thread that's transmitting p1 wakes up, it should self-terminate and broadcast the CV.  The main thread should return from joining with all these 4 threads and self-terminate. What should be the total simulation time for this case? It should be the time to create 3 tokens plus the service time of a packet. So, your emulation end time should be a little longer than 4.857 seconds. If your emulation end time is less than 4.857 seconds or much longer than 4.857 seconds, you need to figure out if there is a bug in your code.  Below is a sample printout for your reference. Of course, due to the nature of a time-driven simulation, your printout should be slightly different. What's important is that you **understand** exactly why **the sequence of events below is the ONLY correct sequence of events for this set of simulation parameters**!  Emulation Parameters:  number to arrive = 1  lambda = 1  mu = 0.35  r = 1.5  B = 10  P = 3    00000000.000ms: emulation begins  00000667.976ms: token t1 arrives, token bucket now has 1 token  00001045.872ms: p1 arrives, needs 3 tokens, inter-arrival time = 1045.872ms  00001046.029ms: p1 enters Q1  00001335.384ms: token t2 arrives, token bucket now has 2 tokens  00002127.836ms: token t3 arrives, token bucket now has 3 tokens  00002128.261ms: p1 leaves Q1, time in Q1 = 1082.232ms, token bucket now has 0 token  00002129.187ms: p1 enters Q2  00002130.350ms: p1 leaves Q2, time in Q2 = 1.163ms  00002130.817ms: p1 begins service at S2, requesting 2857ms of service  00004991.834ms: p1 departs from S2, service time = 2861.017ms, time in system = 3945.962ms  00004992.989ms: emulation ends    Statistics:    average packet inter-arrival time = 1.04587  average packet service time = 2.86102    average number of packets in Q1 = 0.21675  average number of packets in Q2 = 0.000232927  average number of packets at S1 = 0  average number of packets at S2 = 0.573007    average time a packet spent in system = 3.94596  standard deviation for time spent in system = 0    token drop probability = 0  packet drop probability = 0  Later on, when you get your simulation working, you should run:  ./warmup2 -n 1  and the **order of events** you get in your printout should be **identical** to the above (although you may get a different server to transmit p1). Of course, the actual timestamps may be different, but the order of events must be identical to the above because this is the **only** correct order of events (and you need to figure out why)!  Please understand that even if you run the same command over and over again, you should get a slightly different printout every time. Also, if you run this case over and over again, about 50% of the time, p1 should be transmitted by S1 and the rest of the time, p1 should be transmitted by S2. (This probably won't happen on a 32-bit Ubuntu 16.04 system. But it can happen in some other system.) If it turns out that p1 is served by the same server in every run, it does not necessarily mean that you have a bug in your code! If that happens, you should check your code to make sure that you don't have a bug in your code that creates a condition that makes p1 to be served by the same server every time. Later on, when you have more code implemented and test with "txfile.txt in the spec", you need to make sure that in every run of your code with that input file, there would be a time when both servers are "transmitting" simultaneously. |