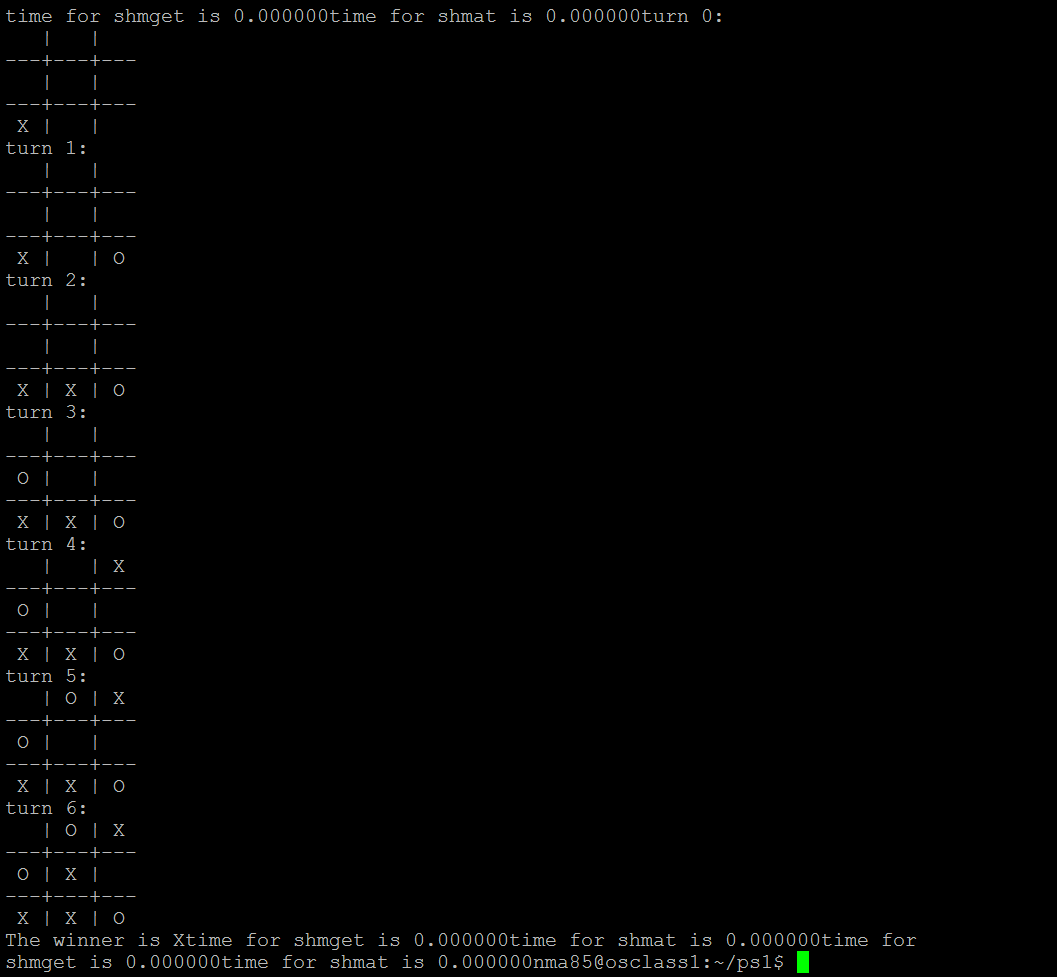
**1)**

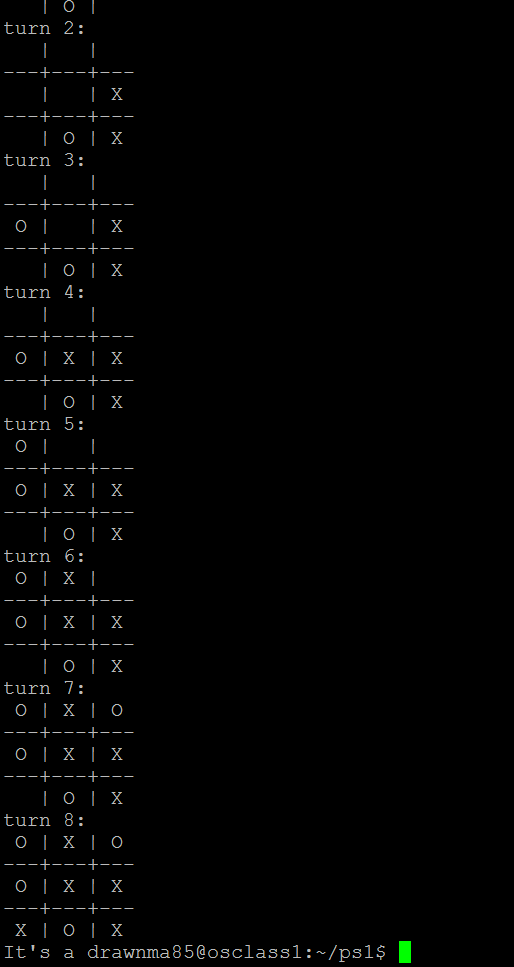
Running the program



I implemented both random moves as well as the minimax algorithm. Both processes wait 4 seconds before making a move. I allocated the board as an array in shared memory, and also used shared memory for IPC. The judging process also makes sure neither process is cheating by always checking whose turn its is and preventing illegal moves.

**2)**

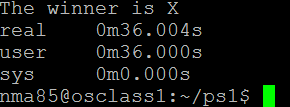
Running the program



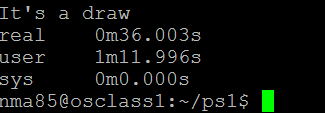
Similar to the multi processed case, I implemented two algorithms for making moves. The main thread (judge thread) decided whose turn it was and also enforced rules (similar to multiprocessed case).

**3.1)**

Running the multiprocess TTT program with random selection, with 4 seconds for “thinking”



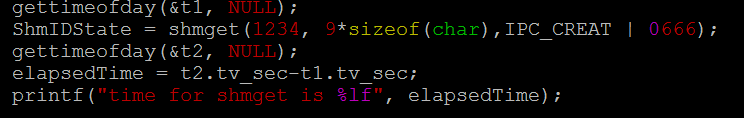
Running the multithreaded TTT program with random selection, with 4 seconds for “thinking”



Time needed for syscalls shmget and shmat in TTT multiprocess is 0 for both, the amount of time they take is negligible



Code used



There are no system calls in TTT multithread

The numbers make sense for multiprocessing. Given that we waited 4 seconds to make a decision each time, 9 times, user time is 36 seconds. Real time is 0.0004 larger for two possible reasons

1. Heavy I/O activity (on server)
2. Lack of CPU (waiting for other processes)

Both seem likely. Since there are other users on the virtual machine, and they might be doing heavy I/O, and my processes could be waiting for there’s.

As for system time, gettimeofday assumed they ran for 0 seconds since they take so little time. Therefore, it makes sense that system time is 0.

The numbers also make sense for multithreading. Real time is almost the same as multiprocessing since we are waiting 4 seconds in both, and they are running almost the same code.

User time, however, is double real time. User time is the amount of time spend in user-mode within the process, since each thread is taking turns, and we have two additional threads, user time is doubled

Comparison:

* Real time is the same for both, since real time is wall clock time (what we could measure with a stopwatch). Therefore, since both multiprocessing and multithreading involve running code concurrently, they would take the same real time.
* User time is the amount of time spend in user-mode within the process. The other processes are not taken into account in user time since they are separate processes, similar to running two separate processes from the command line. The threads are accounted for in multithreading, since all of threads are part of the same process, they are essentially taking turns to execute. Thus the time each was running was recorded.

**3.2)**

Thread Pros and Cons

|  |  |
| --- | --- |
| Pros | Cons |
| No need to allocate shared memory, since address space is shared | Since address space is shared, it is easier to corrupt other threads |
| Context switching between threads takes less time | Harder to debug due to data dependencies |
| Easier to communicate between two threads (due to shared address space) | Harder to understand code |
| Easier to synchronize |  |
| Require less memory (memory shared) |  |

Processes pros and cons

|  |  |
| --- | --- |
| Pros | Cons |
| Separate address space, so threads do not corrupt each other | Inter process ommunication is harder, need shared memory or pipes |
| Easier to code and understand since each process has its own code | Require more memory |
| Child processes are interruptablle/killable | Context switching requires more time |

**3.3)**

The starting player has 9 possible moves, but since some are equivalent, he has 3 equivalent moves.

* Whether the starting player can win depends on the other player’s strategy as well. Assuming the first player is running an optimal strategy (minimax), and the second player is running that as well, then they will draw all the time. However, if the other player is randomly playing his moves, then the minimax player will win most of the time. He will draw only if the other player randomly uses the same moves as a minimax payer would have (highly improbable). However, the answer is no, there are no algorithms that can always win it for the first player.
* For similar reasons, the second player can win depending on his opponent’s strategy. Wining is not guaranteed however, so the answer is also no.
* The starting player can always draw if he is running minimax. He will never lose for the following reason. The second player will only be able to react to his moves every turn, and will not be able to apply his own winning strategy.
* The second player can always draw as well if he is running minimax. He will be able to recognize attempt to fork (two ways victory) and will counter them.