CSC 343 Operating Systems, Spring 2024

Dr. Dale E. Parson, Assignment 1, Implementing and testing a first state machine simulation.
This assignment is due via **make turnitin** from the prisonerd2024 directory by **11:59 PM on Thursday February 22**. There is a 10% penalty for each day it is late, and I will not accept solutions after I go over my solution in class. Points values are documented with STUDENT comments in state machine definition file prisonerd2024.stm. Each bug I need to fix costs the point value of that transition.

The goal of this assignment is to learn how to write an introductory state machine in this semester's STM language. We will be simulated the Iterated Prisoner's Dilemma, for reference see:
https://en.wikipedia.org/wiki/Prisoner%27s_dilemma

Perform the following steps to get my handout. You will need to test on machine mcgonagall as previously explained (**ssh mcgonagall** from acad). I usually edit in one window on acad and test I another on mcgonagall, so I can run **make graphs** on **acad** after my program compiles on mcgonagall to generate one or more graphical image files for the project state machine(s).

```
cd  $HOME                # or start out in your login directory
mkdir  OpSys  # All of this semester's work goes under here, skip if you did it before.
cd  ./OpSys
cp  ~parson/OpSys/prisonerd2024.problem.zip  prisonerd2024.problem.zip
unzip prisonerd2024.problem.zip
cd ./prisonerd2024
make clean test csv
```

Testing fails with the handout directory as follows. Look at both the first and last lines of any error message to decode the error.

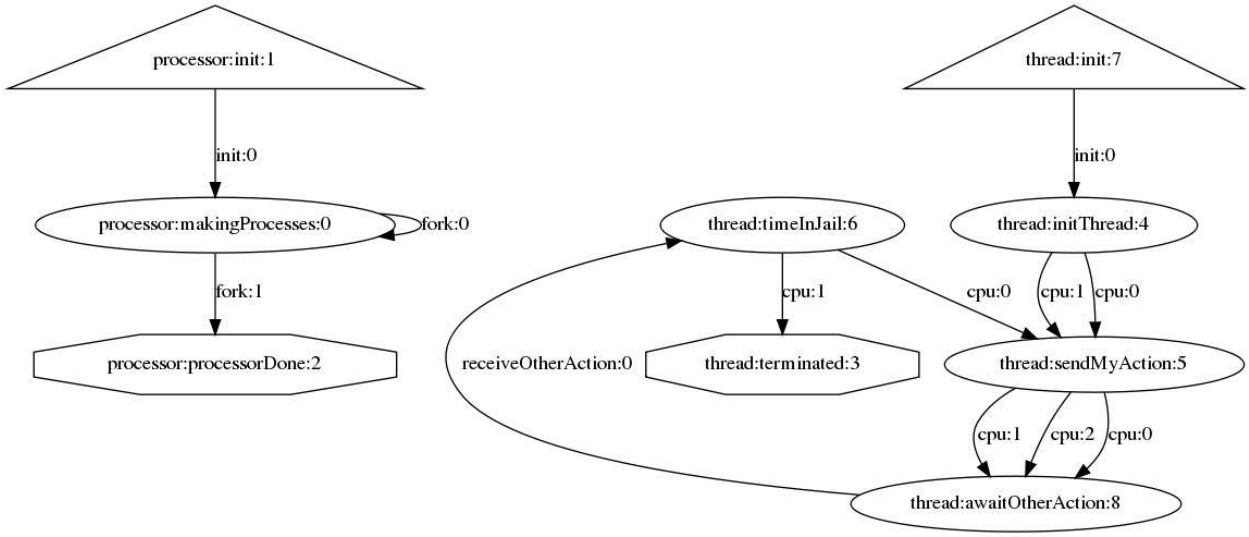
ERROR, Invalid transition from state sendMyAction -> awaitOtherAction, awaitOtherAction not in machine thread.
ERROR, Invalid transition from state awaitOtherAction -> timeInJail, awaitOtherAction not in machine thread.
ERROR, Invalid transition from state awaitOtherAction -> timeInJail, timeInJail not in machine thread.

It fails because you need to write state declarations, initialize some variables, and write transitions that are missing from file **prisonerd2024.stm**. All of your work goes into that file. Look for the STUDENT comments in the file.

We will go over the Iterated Prisoner's Dilemma in class. You can go over the above linked page for more detail. Essentially, in each game of the Iterated Prisoner's Dilemma there are two partners, a process thread ID (tid) of 0, and a tid of 1. At each turn, each player makes a move of "**defect**" or "**cooperate**" without knowing what the other player will do. Then the player computes the penalty for its move in relation to the move made by its partner, according to this scoring table.

My player's move (tid)	Other player's move (othertid)	My penalty for this move
"defect"	"defect"	2
"defect"	"cooperate"	0
"cooperate"	"defect"	3
"cooperate"	"cooperate"	1

See tables **processor.action**, **processor.penalty**, and **processor.processorSampling** in the handout **prisonerd2024.stm** for implementation details. The processor variable is a built-in variable that points to the current processor object. The thread variable points to the current thread object, and the pcb (Process Control Block) variable points to an object shared by the two threads in a single process. This simulation fork()s 10 processes, and thread 0 spawn()s an additional thread 1 for its partner in its process.



This is the diagram of the final **prisonerd2024.stm** showing all states and transitions. The goal of any player (thread within a process) is to minimize its time spent in the **timeInJail** state. Our simulation records that time for us to analyze. There are four strategies enumerated in **processor.action**, which my handout code places into each thread's **mystrategy** variable. Here are the strategies.

Strategy	Resulting action (how to find action to send to partner)
"defect"	Always send a "defect" message.
"cooperate"	Always send a "cooperate" message.
"halfsy"	Use sample(0,1,"uniform") to get a 0 or 1; send "defect" on a 0 and "cooperate" on a 1. This is a pseudo-random strategy with a 50% probability of defecting and a 50% probability of cooperating on each move.
"reciprocate"	Send "cooperate" on the first move, and on every move thereafter, simply echo the partner's **previous** move. (You don't know what its current move will be.)

The game must be coded to make exactly 100 moves, and then go into the terminated state, using variables **loopCount**, **loopLimit**, and a guard condition on the transition into the terminated state that we will discuss in class.

I have written the **processor** state machine (do not change it), the state declarations and the several transitions of the **thread** state machine, and the variable initializations in that state machine. I have given detailed comments in the code for completing your documentation comments at the top and your four transitions.

The transition from **init** to **initThread** initializes variables machineID, **pid**, **tid**, and **mystrategy**. It schedules the cpu event that will take it out of initThread in a final action call to cpu(0). Any cpu scheduling that you perform as a transition's final action must be **cpu(0**), with one exception. The only exception is the cpu(N) call into state **timeInJail**, which must compute a penalty between 0 and 3, inclusive, based on the table at the bottom of the previous page. The N in cpu(N) is that per-move penalty for transiting into **timeInJail**.

You must code 2 transitions from **sendMyAction** to await **otherAction**. Python uses keywords "and" and

"or" instead of "&&" and "||" for combining multiple Boolean conditions. The transitions correspond to the 4 strategies in the above table. Compute a value for variable **SendRecvAction**, based on the appropriate strategy for this thread, and then invoke **SendRecvSync@** as the final action. We will discuss macro **SendRecvSync** and its variable **SendRecvAction** in class. **SendRecvSync** sends the value in **SendRecvAction** to the partner, and then waits for the partner to send its own **receiveOtherAction** event. That **receiveOtherAction(action)** event carries the partner's action ("defect" or "cooperate") as its event argument; this event triggers the transition into state **timeInJail**.

On the transition into timeInJail, my code performs the following statement as the first action.

```
pcb.incomingMessage[tid] = None;
```

Macro **SendRecvSync** uses a combination of the **receiveOtherAction(action)** event and message buffer pcb.incomingMessage[tid] to solve a synchronization problem that we will discuss in class. The above assignment statement clears the buffer after this player has consumed its message in preparation for a later move interaction. **None** is Python's equivalent of the NULL pointer.

The transition into **timeInJail** computes the penalty based on the above information for a call to cpu(penalty), with the thus-generated cpu() event getting the machine out of **timeInJail**. This simulation is profiling the cpu time spent in **timeInJail**.

The transition guard expressions out of **timeInJail** compare variable **loopCount** to **loopLimit**, going to the terminated state when loopCount >= loopLimit.

When you are ready, run **make clean test** on **mcgonagall** to run tests. We will see successful and failed make test runs in class. If you get a run-time error referring to an entry in the generated __codeTable__[N], run this from the command line, supplying the index N:

./decode.py prisonerd2024.py N

If compilation or testing blows up, you can inspect the log file in **prisonerd2024.log**, searching for error messages and the *defunct* string. Ignore this warning in the log file:

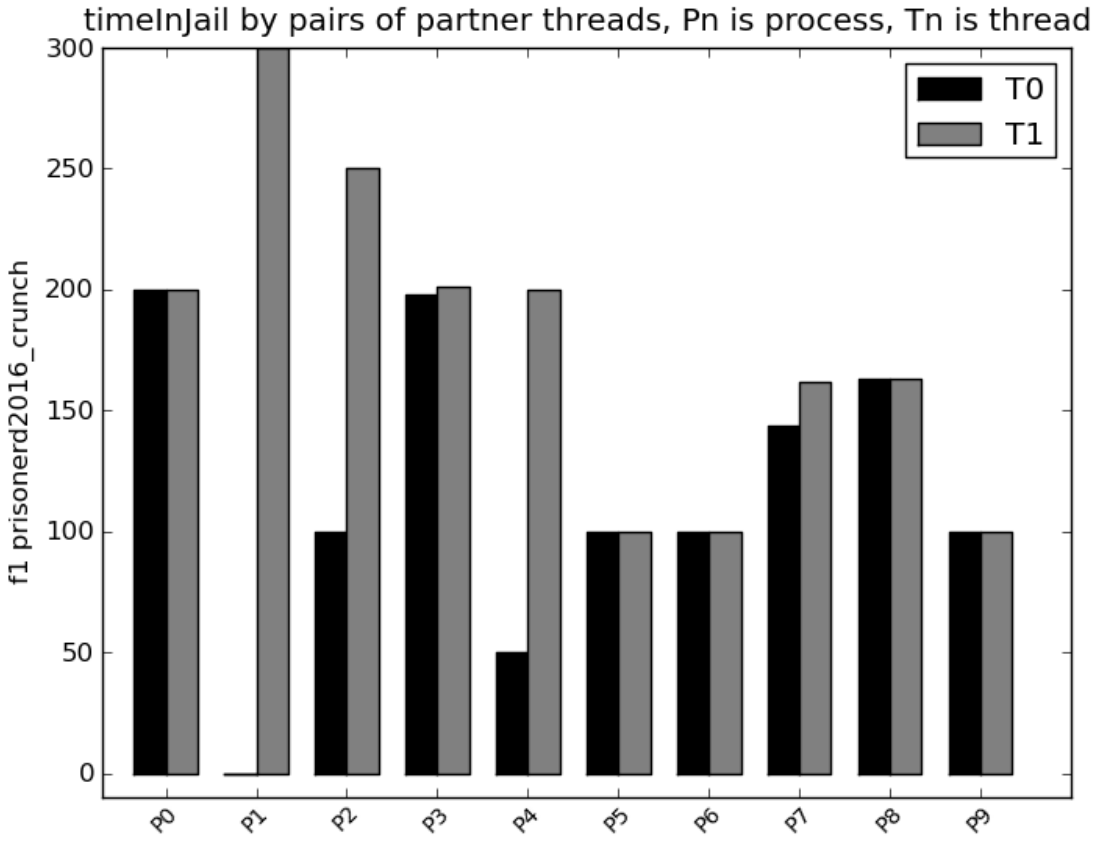
000000000002,MSG,thread 0 process 0,WARNING, signalEvent discards event type receiveOtherAction because model is waiting in queue: waiting on simulation scheduler (simulation sleep) for model: Thread pid 0, tid 0, state sendMyAction, waitingon cpu, __sleepResult__ None, __isdead__False

Macro **SendRecvSync** uses two means to send an action to a partner and await the partner's action, a call to library function **signalEvent** and message buffer **pcb.incomingMessage[othertid]** as discussed above. When the receiving thread state machine is not in a state that responds to the event sent by **signalEvent**, it logs the above WARNING message. However, since this application checks the **pcb.incomingMessage[tid]** buffer in such cases, we can ignore the warning.

When all goes well with **make clean test**, the penalty sums in state **timeInJail** for each thread state machine will match my values in file **prisonerd2024_crunch.ref** to within 20%, and you are ready to turn it in. Check that you have completed the comments. To turn it in do this:

make turnitin

and hit Enter at the prompt.



In the above, P0-T0 is **timeInJail** for thread 0 of process 0, followed by **timeInJail** for thread 1 of process 0, which does not have an X label. Then comes **timeInJail** for thread 0 of process 1, and so on. We will discuss the analysis of this graph in class.

When everything works, run **make turnitin** before the project deadline. Make sure to meet any documentation comment requirements stated in the STUDENT comments in the handout. If you later make changes that you want to turn in, just run **make turnitin** again, which over-writes the previous submission. There is a 10% per-day penalty for late assignments, and I will not accept and assignment after I go over my solution in class. Note we are not using the turnin script in my courses. Also, please run **make clean** whenever you end a work session on this project, since the log files actually reside in ~parson/tmp in order to avoid overloading your file space limits.

See **STM.doc.txt** in the project directory for documentation for the simulation library functions.