CSC 343 Operating Systems, Spring 2020

Dr. Dale E. Parson, Assignment 2, modeling an atomic spin lock, a mutex, and a condition variable. This assignment is due via **make turnitin** from the criticalSection2020 directory by **11:59 PM on Fridat March 27 (was: Wednesday March 25 (rescheduled from March 18 due to school shutdown))**. Please use the allotted time wisely. Start early. This is a meaty project. There is a 10% per day late penalty, and any assignment turned in after I go over the solution is worth 0%. **You must document any code you write/change with documentation comments. I will deduct 10% for missing documentation, and will deduct points if your name is missing from the top of each of your source files**.

The goal of this assignment is to compare performance characteristics of using an atomic spin lock, a mutex, and finally a condition variable in synchronizing multiple Producer and Consumer threads. In addition to a STM simulation, you will explain measurements for one of the performance metrics of a set of concurrent Producer / Consumer state machines.

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
cd  ./OpSys
cp  ~parson/OpSys/criticalSection2020.problem.zip  criticalSection2020.problem.zip
unzip criticalSection2020.problem.zip
cd ./criticalSection2020
make test
```

1. Understand the workings of Unsafe.stm

The handout includes file **Unsafe.stm** that attempts to solve the critical section problem for a Producer / Consumer simulation, but it has a race condition. Running **make testUnsafe** tries to simulate Unsafe.stm twice, once with 1 thread each for a Producer process and a Consumer process, and again with 10 threads for each process. I have created a race condition in Unsafe.stm by having the Consumer states get ahead of the Producer states in the simulation, consuming messages before they are produced. That situation can arise in real systems when the relative timing of production and consumption are not synchronized. I have forced it with cpu(N) timing on transitions so you can test and get the same results on each run. I considered making the relative timing of Producer and Consumer threads probabilistic as they normally would be, but doing so would lead to a different test run timing each time, making debugging difficult.

Here is how **make testUnsafe** works. First it runs testUnsafe.stm with 1 thread in each of the two processes, based on this variable built into the STM architecture.

```
numThreadsToSpawn = processor.contextCount - 1 ;
```

Field **processor.contextCount** is set here by the makefile's simulation command line.

/bin/bash -c "PYTHONPATH=/home/KUTZTOWN/parson/OpSys:.:.. STMLOGDIR=/tmp time /opt/csw/bin/python Unsafe.py **1** 4 100000 12345 2"
MSG cmd line: ['Unsafe.py', **'1'**, '4', '100000', '12345', '2'], usage USAGE: python THISFILE.py

NUMCONTEXTS NUMFASTIO SIMTIME SEED|None LOGLEVEL

Since the processor's fork() call always starts thread 0 in the forked process, the line above sets numThreadsToSpawn to 0 because thread 0 is the only thread running in the first pass. However, this command line in the second simulation run allocates 10 into processor.contextCount.

/bin/bash -c "PYTHONPATH=/home/KUTZTOWN/parson/OpSys:.:.. STMLOGDIR=/tmp time /opt/csw/bin/python Unsafe.py **10** 4 100000 12345 2"
MSG cmd line: ['Unsafe.py', **'10'**, '4', '100000', '12345', '2'], usage USAGE: python THISFILE.py **NUMCONTEXTS** NUMFASTIO SIMTIME SEED|None LOGLEVEL

Thread 0 spawns 9 additional threads in each of the Producer and Consumer processes (1 process each), for a total of 10 threads in the Producer process, and 10 threads in the Consumer process. This mix of threads runs into the race condition. Here is the test that previously fails on **make testUnsafe**.

$ **diff Unsafe1.msg.out Safe1.msg.ref**
1c1,2
< 97

> 100
> 0
3,6d3
< 3
< None product value CONSUMED
< None product value CONSUMED
< None product value CONSUMED
103a101,103
> 98 product value CONSUMED
> 99 product value CONSUMED
> 100 product value CONSUMED
make: *** [testUnsafe] Error 1

The Producer produces threadcount x 100 consecutive integers that the Consumer reports via **msg** calls. The makefile inspects for the number of valid msg statements, the number of invalid msg statement due to an incorrect value (caused by the race condition), and then it sorts and saves the lines themselves in Unsafe10.msg.out for the failed test. There is one Safe1.msg.ref file that shows the correct output when the Producer and Consumer synchronize their message exchanges. Here is the start of **Safe1.msg.ref**.

$ **head Safe1.msg.ref**
100
0
0
1 product value CONSUMED
2 product value CONSUMED
3 product value CONSUMED
4 product value CONSUMED
5 product value CONSUMED
6 product value CONSUMED
7 product value CONSUMED

There is a grand total of 100 consecutive values in order. Here is the start of **Unsafe1.msg.out** in the

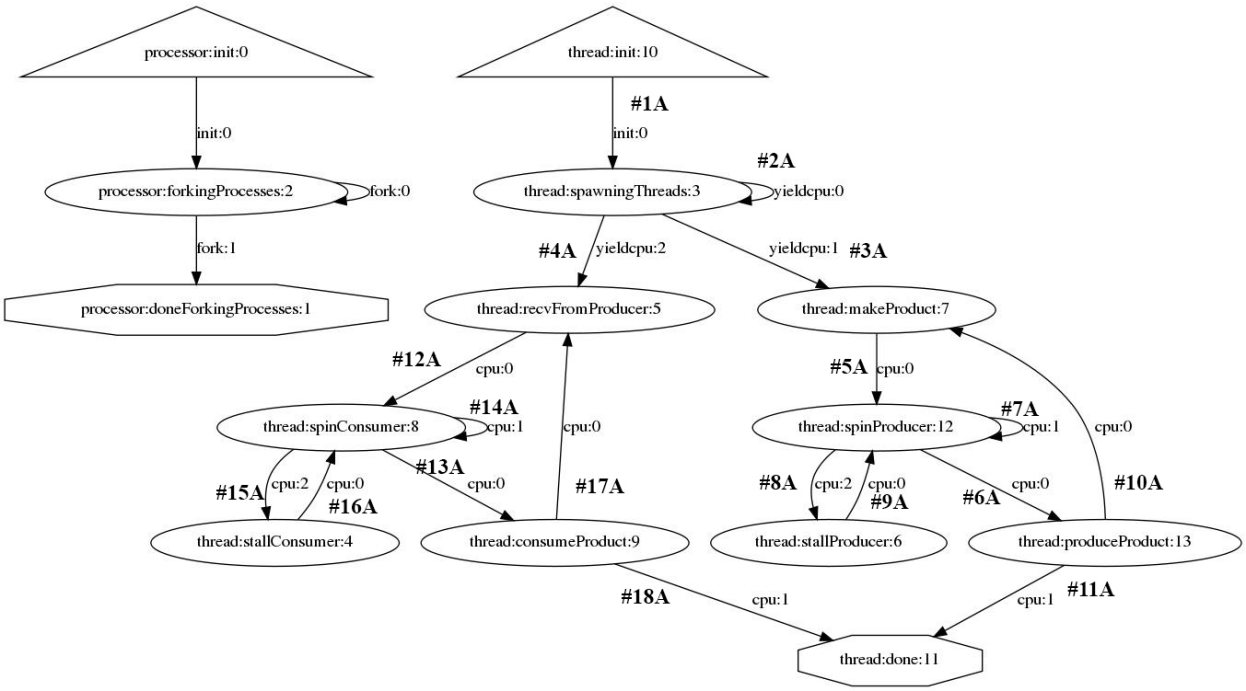
failed test.

$ **head Unsafe1.msg.out**

97 **THIS IS THE COUNT OF CORRECT VALUES**
0 **COUNTS ERRORS IN RE-ASSIGNING THE "product"**
3 **COUNTS None VALUES CAUSED BY RACE COND.**
None product value CONSUMED
None product value CONSUMED
None product value CONSUMED
1 product value CONSUMED
2 product value CONSUMED
3 product value CONSUMED
4 product value CONSUMED

The None values are Python's equivalent of NULL pointers in C++. The race condition that we will solve writes NULL values because the Consumer threads do not test the validity of data passed from the Producer threads via a shared variable **processor.messageBuffer** in the processor object. Your respective solutions will use an atomic spin lock, a mutex lock, and a condition variable to synchronize access to this shared processor.messageBuffer.

Testing also checks state times within very loose tolerances, 20% and 100%. If any test fails, its log file will be in the directory, which you can find using **ls *.log**. Testing deletes the lengthy log file when a test passes or when you run **make clean** or **make turnitin**. If you get an error on one of these crunchlog % differences, check with me to see if we should loosen the tolerances. **The diff of the msg.out to msg.ref file as seen at the top of this page, on the other hand, must be an exactly match**.



Unsafe.stm is the handout state machine. We will go over its working in class.
See http://acad.kutztown.edu/~parson/Unsafe.jpg

I have labeled the transitions in the generated diagram above with labels #1A through #18A to correspond with comments in Unsafe.stm as an aid to explanation. Here is what each of the transitions in Unsafe.stm does.

#1A Initializes some variables. Each state machine gets its own local variables, even when they share the same name, just like regular threads or separately executing processes. The fields in the processor object model kernel data structures; they are accessible to every thread in every process in the simulation. The fields in the pcb are available to all threads in a process; there is one unique pcb per process; this project does not use the pcb, but it does use the processor object to store the data buffer and synchronization lock and Queue fields. All threads in the distinct Producer and Consumer processes exchange data via global shared memory in the simulated kernel. Run the manual-printing commands **man shmat**, **man shmdt**, and **man shmctl** on acad or mcgonagall to see the C-language system calls for Unix shared memory.

#2A Thread 0 runs in this loopback transition to spawn additional thread.

#3A Sends all threads in an even-numbered process ID (process 0 here) to the makeProduct state as Producers. I have used cpu(10) on this transition to force the Producer to reach its makeProduct state after the Consumer has reached recvFromProducer via #4A.

#4A Sends all threads in an odd-numbered process ID (process 1 here) to the recvFromProducer state as Consumers. I have used cpu(1) on this transition to force the Producer to reach its makeProduct state via #3A after the Consumer has reached recvFromProducer via #4A.

#5A Sends Producer threads to spinProducer after creating a new product.

#6A A Producer stores its product into processor.messageBuffer[0] and cycles back to make another or terminate.

#7A In Unsafe.stm this loopback does nothing and never runs. See its guard condition. In your three solutions it will loop while waiting to acquire a lock and also for other test conditions.

#8A and **#9A** are dummy stubs in Unsafe.stm. They never run. Mutex.stm and Condvar.stm use the stallProducer state for a Producer thread that stalls itself in a Queue in the kernel (processor object) while waiting for a lock (Mutex.stm) or a lock and a condition (Condvar.stm).

#10A and **#11A** make the decision of whether to continue making products (#10A) versus terminating the Producer thread after making numMessagesPerThread (100) products (#11A).

#12A just goes to spinConsumer, which is the Consumer counterpart of spinProducer. In your solutions spinConsumer spins waiting to acquire a lock. In Mutex.stm and Condvar.stm, spinConsumer transitions to stallConsumer for a Consumer thread that stalls itself in a Queue in the kernel (processor object) while waiting for a lock (Mutex.stm) or a lock and a condition (Condvar.stm). Transitions **#14A**, **#15A** and **#16A**, designed for Consumer spinning and stalling, do nothing in Unsafe.stm.

#13A Removes the product sent by a Producer thread via processor.messageBuffer[0], nulls processor.messageBuffer[0] with a None value, reports the product value in an msg() statement that you must not change, and goes on to consumeProduct.

#17A and **#18A** make the decision of whether to continue consuming products (#17A) versus terminating the Consumer thread after consuming numMessagesPerThread (100) products (#18A).

See file **Unsafe.stm** in the handout code. We will go over it in class.

Unsafe.stm is unsafe because it does nothing to synchronize communication of Producer and Consumer threads using the processor.messageBuffer shared data. (NOTE: Always assign into and out of processor.messageBuffer[0], which is a one-element buffer in this assignment. It never gets any bigger.) You will fix that in your parts of the project by accessing shared data processor.messageBuffer only when the thread wanting to access processor.messageBuffer holds the lock in field processor.thelock.

2. 60% of project grade: Write Atomic.stm to use an atomic spin lock to serialize access to data shared across threads.

In the STM programs that follow, your code must maintain the following restrictions. Violating any of these restrictions leads to deductions in points.

- First **cp Unsafe.stm Atomic.stm** and then edit changes into Atomic.stm.

- Update documentation comments at the top with your name, the file name, and an outline of how the algorithm differs from its predecessor.

- Every access to thread-shared data in the processor object must be serialized by setting the lock variable **processor.thelock** from False to True (Python uses capital False and True as boolean values), if and only if **processor.thelock** is False (not locked) at the time a thread STM wants to set it to True. The thread should also set local variable **haveLock** to True at the same time. Every thread STM has a **haveLock** variable, but there is only one **processor.thelock** variable shared by all threads. Every access to critical section data **processor.messageBuffer[0]** (assignment or access to its value) must take place only when the accessing thread has set both **processor.thelock** and **haveLock** to True, and it may set these true only when it detects them both as == False within a guard condition. Violation of the requirements of this paragraph violates the critical section constraints and leads to race conditions that testing may or may not detect. If I find violations while testing, I will insert a msg() statement to verify the fact of accessing **processor.messageBuffer** without holding the lock, and will deduct 10% per violation in the source code. You can treat the actions of a single STM transition as being atomic – a sequence of actions cannot be preempted by another thread – but you must use **processor.thelock** to serialize access to data across multiple transitions as specified in detail below.

If code get a run-time error during testing that just reports an index into the Python byte code table like this:

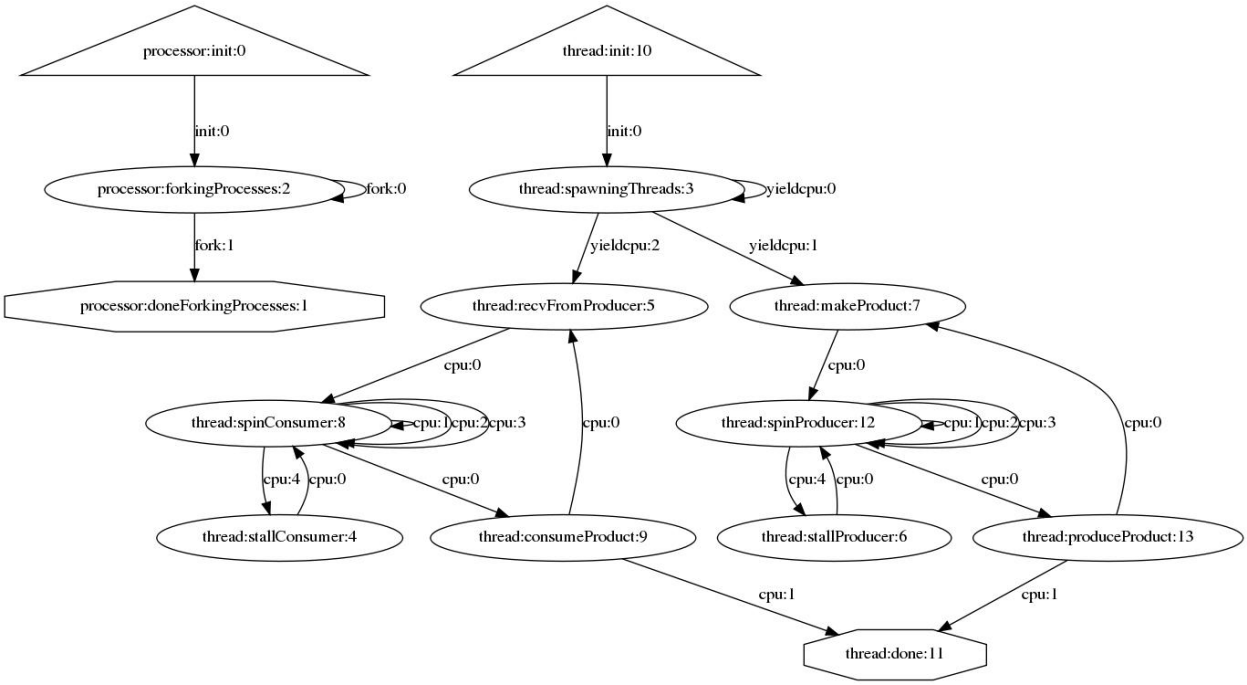
```
Traceback (most recent call last):
  File "Unsafe.py", line 769, in <module>
    main()
  File "Unsafe.py", line 713, in main
    scheduler.__run__()
  File "/home/KUTZTOWN/parson/OpSys/state2codeV15/CSC343Sim.py", line 138, in __run__
    waitingObject.__generator__.next()  # run() the model
  File "Unsafe.py", line 152, in run
    exec(__codeTable__[4],globals,locals)
  File "nofile", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
0.11user 0.05system 0:00.28elapsed 58%CPU (0avgtext+0avgdata 15856maxresident)k
0inputs+8outputs (0major+8813minor)pagefaults 0swaps
```

make: *** [testUnsafe] Error 1

Just run this utility from within your project directory, with the name of the STM's .py file and the reported line number into __codeTable__[N] as the command line arguments, like this:
$ **./decode.py Unsafe.py 4**

__codeTable__[4] = compile('processesToGo -= 1/0','nofile','exec'),

Here are detailed instructions for writing Atomic.stm that simulates using an atomic boolean variable processor.thelock as a spin lock. These are precise changes to make to a copy of Unsafe.stm. The diagram of my Atomic.stm appears below. Yours may differ in incidental details, although if you follow my instructions carefully, they will have the same topology.



Atomic.stm shows my solution.
See http://acad.kutztown.edu/~parson/Atomic.jpg

- At transition **#6A** from spinProducer -> produceProduct, add a guard that allows the transition only when the thread holds the lock (see above) and processor.messageBuffer[0] == None (i.e., processor.messageBuffer[0] is empty). This guard prohibits setting processor.messageBuffer[0] when the lock is not held or when the buffer has contents, thereby avoid over-writing a previous product. Release the lock by setting both haveLock and processor.thelock to False after storing processor.messageBuffer[0] = product.

- At transition **#7A** spinProducer -> spinProducer, make the following changes. I made two additional transitions **#7B** and **#7C** spinProducer -> spinProducer. See the above diagram.

- #7A loops when haveLock is False but processor.thelock is True, meaning that a different thread holds the lock. This is spinning on the spin lock. There is no change in the action.

- #7B loops when no thread holds the lock (haveLock and processor.thelock are False). Its action grabs the lock by setting both to True, and loops back with cpu(1). Any cpu() call you add to the STM must be cpu(1) for 1 tick to maintain reference file timing comparisons.

- #7C loops when this thread holds the lock (haveLock is True) but processor.messageBuffer[0] != None, meaning that processor.messageBuffer[0] has contents that it must not over-write. Its action releases the lock (haveLock and processor.thelock) and loops back with cpu(1).

- #13A has changes that are Consumer counterparts to #6a above. Add a guard that allows the transition only when the thread holds the lock and processor.messageBuffer[0] != None (i.e., processor.messageBuffer[0] has contents to consume). Release the lock by setting both haveLock and processor.thelock to False immediately after fetching the product and then setting processor.messageBuffer[0] = None. Leave my msg statement and the remaining code as it is in order to avoid a formatting diff on output.

- #14A extends spinConsumer -> spinConsumer into three transitions similar to the extension of #7A above. See the diagram.

- #14A loops when haveLock is False but processor.thelock is True, meaning that a different thread holds the lock. This is spinning on the spin lock. There is no change in the action.

- #14B loops when no thread holds the lock. Its action grabs the lock by setting both to True, and loops back with cpu(1).

- #14C loops when this thread holds the lock but processor.messageBuffer[0] == None, meaning that processor.messageBuffer[0] is empty, with nothing to consume. Its action releases the lock and loops back with cpu(1).

- After making any edit changes, run **make clean testAtomic** to test this part of the project. The 1-thread test takes 0.17 (previously 1.4 on harry and an older version of the STM) seconds to complete and the 10-thread test takes 3.15 (previously 43.4) seconds in my preparation tests. Anything running over 30 seconds likely indicates a bug. Hit control-C and debug the log file. I found searching for the CONSUMED messages useful. If your test simulation gets stuck in a loop, look near the bottom of the log file for a fruitless loop through states. Run **ls *.log** to see the log file name.

3. <u>15% of project grade: Write Mutex.stm to simulate a mutex for serializing access to data shared across threads. The mutex uses a queue to hold threads that are waiting for lock access.</u>

- After **make testAtomic** is working correctly, **cp Atomic.stm Mutex.stm** then edit Mutex.stm, making the following changes.

- Update the documentation comments at the top of Mutex.stm as before. Feel free to add comments at transitions if they help you to keep track of the logic.

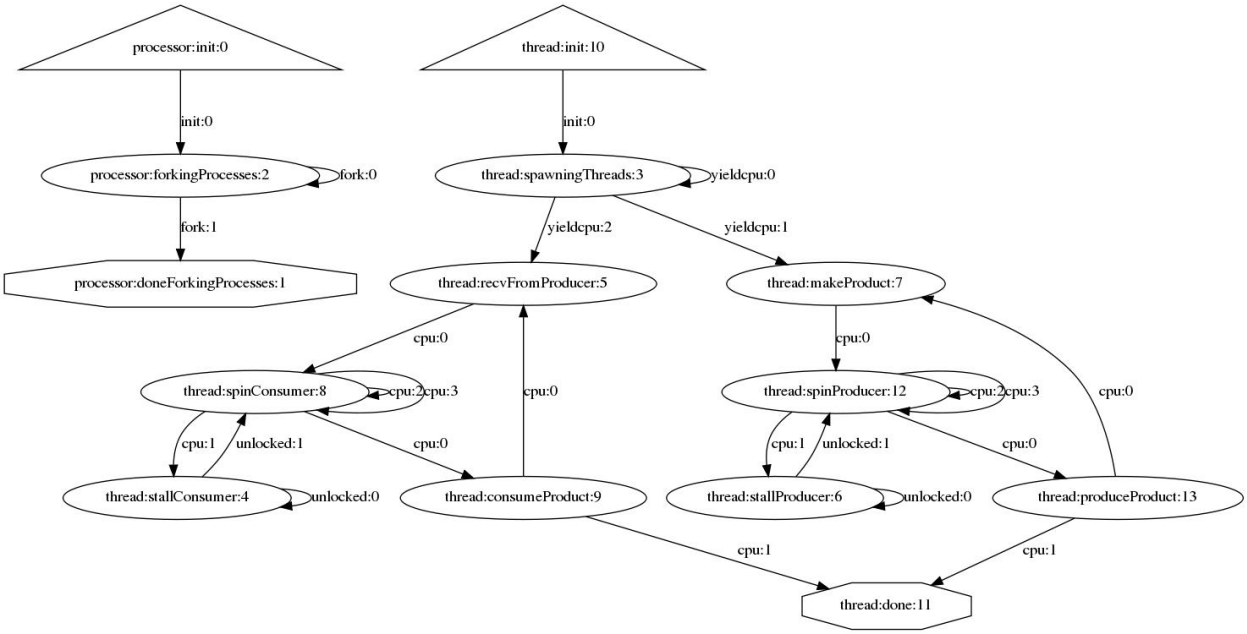
- **<u>EVERY</u>** transition that releases the lock must add this line of code immediately after setting **haveLock** and **processor.thelock** from True to False. There are more than 1 such transitions.

  if len(processor.mqueue): signalEvent(processor.mqueue.deq(), 'unlocked') ;

  That line checks to see whether a stalled thread is waiting in Queue **processor.mqueue** to run. If

there is at least one thread, that line dequeues it (see STM.doc.txt for the Queue operations) and sends it an *unlocked* event. Python's "if" is available for conditionally running a single command; do not place blocking function calls within an "if" statement. We are extending the simulation framework with an *unlocked* event type by using **waitForEvent** and **signalEvent**. STM.doc.txt documents those functions. The declaration of an empty FIFO queue mqueue is in processor.

processor.mqueue = Queue(False) ; NOTE – False for a FIFO queue, True for priority Queue.



Mutex.stm shows my solution. Note the spin states cut back to two self-transitions each.
See http://acad.kutztown.edu/~parson/Mutex.jpg

- Replace #7A that transits spinProducer -> spinProducer with a transition spinProducer -> **stallProducer** with an identical guard condition (haveLock is False and processor.thelock is True) and action that sequence as follows.

  processor.mqueue.enq(thread); **NOTE – This is now #8A spinProducer -> stallProducer**
  waitForEvent('unlocked', True)@

  When a thread enters either stall state, it enqueues itself into processor.mqueue (the mutex queue), then blocks until another thread signals it via signalEvent as given above.

  Note that #7A is deleted from my Mutex.stm diagram.

- The replacement of #7A in the previous bullet covers the case of the spinProducer -> stallProducer transition #8A. Next, there is a **new stallProducer -> stallProducer** *unlocked*() transition in the diagram with the following guard and actions.

  The guard tests that haveLock is False and processor.thelock is True. It covers a race condition in which a thread awakened by a signalEvent cannot get the lock because another thread has come in and taken the lock after this thread was signaled, but before this transition got a change to run.

The actions are as follows.

processor.mqueue.enq(thread);
waitForEvent('unlocked', True)@

The stalled thread simply goes back into processor.mqueue and waits.

- #9A is now **stallProducer -> spinProducer** *unlocked*(). Note the change in event type from cpu() to unlocked(). Its **guard is on haveLock and processor.thelock being False** (the lock is available), and its **action is cpu(0).** Do not change the tick count from 0. I had a race condition in Condvar.stm caused by the fact that a thread releasing another thread from the stalled state would reacquire the lock faster than the unstalled thread could acquire it, leading to starvation of unstalled Producer threads. Using cpu(0) ensures the unstalled thread moves to the front of the simulated scheduling queue in the simulator. My first solution was to use cpu(2) for the thread sending the signal to the unstalled thread. That worked, but it threw the tick counts off from the preceding solutions.

- Replace #14A that transits spinConsumer -> spinConsumer with a transition spinConsumer -> **stallConsumer** with an identical guard condition (haveLock is False and processor.thelock is True) and action that sequence as follows.

  processor.mqueue.enq(thread); **NOTE – This is now #8A spinConsumer -> stallConsumer**
  waitForEvent('unlocked', True)@

  When a thread enters either stall state, it enqueues itself into processor.mqueue (the mutex queue), then blocks until another thread signals it via signalEvent as given above.

  Note that #14A is deleted from my Mutex.stm diagram.

- The replacement of #14A in the previous bullet covers the case of the spinConsumer -> stallConsumer transition #15A. Next, there is a **new stallConsumer -> stallConsumer** *unlocked*() transition in the diagram with the following guard and actions.

  The guard tests that haveLock is False and processor.thelock is True. It covers a race condition in which a thread awakened by a signalEvent cannot get the lock because another thread has come in and taken the lock after this thread was signaled, but before this transition got a change to run.

  The actions are as follows.

  processor.mqueue.enq(thread);
  waitForEvent('unlocked', True)@

  The stalled thread simply goes back into processor.mqueue and waits.

- #16A is now **stallConsumer -> spinConsumer** *unlocked*(). Note the change in event type from cpu() to unlocked(). Its **guard is on haveLock and processor.thelock being False** (the lock is available), and its **action is cpu(0).** It is symmetric to the Producer #9A transition in Mutex.stm.

- After making any edit changes, run **make clean testMutex** to test this part of the project. The 1-thread test takes 0.17 (previously) 1.6 seconds to complete and the 10-thread test takes 0.64 (previously 41.8) seconds in my preparation tests. Anything running over 30 seconds likely indicates

a bug. Hit control-C and debug the log file. I found searching for the CONSUMED messages useful. If you test simulation gets stuck in a loop, look near the bottom of the log file for a fruitless loop through states. Run **ls *.log** to see the log file name.

4. **10% of project grade: Write Condvar.stm to simulate a condition variable for signaling Producer or Consumer threads only at the appropriate time for their jobs. The condition variable uses two distinct queues to hold threads that are waiting for lock access.**

- First **cp Mutex.stm Condvar.stm** after **make testMutex** works without errors. Your changes go into Condvar.stm, starting with documentation comments as usual. Update the documentation comments at the top of Condvar.stm as before. Feel free to add comments at transitions if they help you to keep track of the logic.

- Remove the declaration of **processor.mqueue** from the processor state machine and replace it with two new FIFO queues, **processor.pqueue** and **processor.cqueue**.

- In Condvar.stm, always enqueue a Producer thread into **processor.pqueue**, and always enqueue a Consumer thread into **processor.cqueue**. Simply change all existing mqueue.enq() calls to use the appropriate queue.

- Producer threads always signal Consumer threads like this.

  if len(processor.cqueue): signalEvent(processor.cqueue.deq(), 'unlocked') ;

  These are not new calls. They replace prior calls using processor.mqueue with processor.cqueue.

- Consumer threads always signal Producer threads like this.

  if len(processor.pqueue): signalEvent(processor.pqueue.deq(), 'unlocked') ;

  These are not new calls. They replace prior calls using processor.mqueue with processor.pqueue.

- There are two additional transitions as follows. See the diagram below.

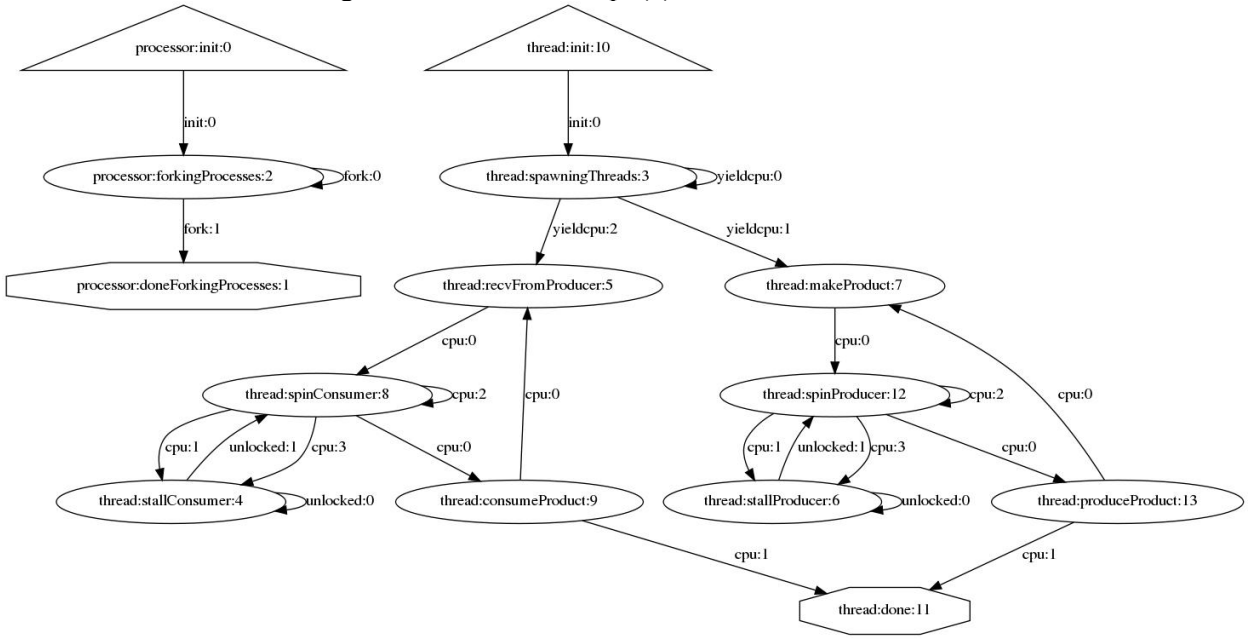
```
spinProducer -> stallProducer cpu()[@haveLock == True
        and processor.messageBuffer[0] != None@]/@
    processor.thelock = False ;
    haveLock = False ;
    if len(processor.cqueue): signalEvent(processor.cqueue.deq(), 'unlocked') ;
    processor.pqueue.enq(thread);
    waitForEvent('unlocked', True)@

spinConsumer -> stallConsumer cpu()[@haveLock == True
        and processor.messageBuffer[0] == None@]/@
    processor.thelock = False ;
    haveLock = False ;
    if len(processor.pqueue): signalEvent(processor.pqueue.deq(), 'unlocked') ;
    processor.cqueue.enq(thread);
    waitForEvent('unlocked', True)@
```

These stall their respective threads when they have the lock, but cannot proceed because of the state of the

message buffer.

NOTE that the above two transitions replace the spinProducer -> spinProducer and spinConsumer -> spinConsumer transitions with identical guards from Mutex.stm. There is only one spinProducer -> spinProducer transition and one spinConsumer -> spinConsumer transition in the Condvar.stm state diagram. This is the remaining spin where the haveLock and processor.thelock are both available (False). The action in each case is to grab the lock and run cpu(1).



Condvar.stm shows my solution. Note the spin states cut back to one self-transition each.
See http://acad.kutztown.edu/~parson/Condvar.jpg

- After making any edit changes, run **make clean testCondvar** to test this part of the project. The 1-thread test takes 0.17 (previously 1.6) seconds to complete and the 10-thread test takes 0.57 (previously 17.2) seconds in my preparation tests. This test runs faster than its predecessors because only threads in the appropriate queue (pqueue or cqueue) awaken to test their conditions at a time when the correct condition has just been satisfied. Anything running over 30 seconds likely indicates a bug. Hit control-C and debug the log file. I found searching for the CONSUMED messages useful. If you test simulation gets stuck in a loop, look near the bottom of the log file for a fruitless loop through states. Run **ls *.log** to see the log file name.

5. **15% of project grade: Answer the 3 questions in the README.txt file in the handout directory.** Make sure to have your answer README.txt file in the directory when you **make turnitin**.

README.txt
STUDENT NAME:
Answer the following 3 questions (5% each for the assignment grade)
in one short paragraph each about the following
simulated test times, taken from the test runs.
See criticalSection2020.time.txt for the full set of reference simulation times.

Q1: In going from 1 Producer-Consumer pair to 10 Producer-Consumer pairs
using a single global buffer in processor shared memory, why does this

maximum slow-down of 11 times slower going from Unsafe (synchronized)
to an atomic spin lock:
Unsafe1.crunch.ref:MAX_spinConsumer=1
Unsafe1.crunch.ref:MAX_spinProducer=1
Atomic1.crunch.ref:MAX_spinConsumer=11
Atomic1.crunch.ref:MAX_spinProducer=2

change to a max slow-down of over 11,000 times slower when there are 10 pairs:

Unsafe10.crunch.ref:MAX_spinConsumer=1
Unsafe10.crunch.ref:MAX_spinProducer=1
Atomic10.crunch.ref:MAX_spinConsumer=11706
Atomic10.crunch.ref:MAX_spinProducer=11705

Given the fact that there are only 10X Producer/Consumer thread pairs,
why is there a 11706/11 = a greater than 1000X slow-down?
YOUR ANSWER GOES HERE:

Q2: In contrast, why is the spinning time for the Mutex with either
1 or 10 Producer-Consumer pairs only marginally longer than the
Unsafe pairs?

Unsafe1.crunch.ref:MAX_spinConsumer=1
Unsafe1.crunch.ref:MAX_spinProducer=1
Mutex1.crunch.ref:MAX_spinConsumer=9
Mutex1.crunch.ref:MAX_spinProducer=2

Unsafe10.crunch.ref:MAX_spinConsumer=1
Unsafe10.crunch.ref:MAX_spinProducer=1
Mutex10.crunch.ref:MAX_spinConsumer=3
Mutex10.crunch.ref:MAX_spinProducer=2
YOUR ANSWER GOES HERE:

Q3: Why would be expect the spinning times for the Condvar (condition
variable) test to be somewhat less than the Mutex times, as shown here?

Mutex1.crunch.ref:MAX_spinConsumer=9
Mutex1.crunch.ref:MAX_spinProducer=2
Condvar1.crunch.ref:MAX_spinConsumer=2
Condvar1.crunch.ref:MAX_spinProducer=2

Mutex10.crunch.ref:MAX_spinConsumer=3
Mutex10.crunch.ref:MAX_spinProducer=2
Condvar10.crunch.ref:MAX_spinConsumer=2
Condvar10.crunch.ref:MAX_spinProducer=1
YOUR ANSWER GOES HERE:

Run **make turnitin** by the due date. There is a 10% per day late penalty, and any assignment turned in after I go over the solution is worth 0%.