Breandan Kerin 1431016 27th September 2018

Advanced Computer Architecture

Tutorial 1 - Question 1

Question 1:

1. **How many ways can the three statements of two identical processes be interleaved?**
    - ➢ 20 different possible interleave variations
2. **How many ways can the four statements of three identical processes be interleaved?**
    - ➢ 34’650 different possible interleave variations
3. **How many ways can the n statements of m identical processes be interleaved?**
    - ➢ For m different identical processes, each with n different statements, the number of possible different interleave variations can be calculated using:
    - ➢ $\#Variations = (mxn)!/(n!)^m$

All code, screenshots and report can be found on my github:
https://github.com/kerinb/CS4521---Advanced-Computer-Architecture

Breandan Kerin   1431016   27th September 2018

Advanced Computer Architecture

Tutorial 1 - Question 2

Question 2:

Using Promela/Spin show that the Bakery lock has the following desirable properties: safety, deadlock free, liveness and starvation free.

The four properties that are required to be proved in this assignment are:

1. Safety/Mutual Exclusion
2. Deadlock free
3. Liveness
4. Starvation Free

Implementation of the Bakery Lock:

The below screenshot of code of the bakery lock is implemented using the Promela programming language. This code does not contain any attempts to try and prove the above properties.

```
int MAX_NUM_THREADS = 2;
byte choosingThread[MAX_NUM_THREADS];
byte ticketNumber[MAX_NUM_THREADS];

proctype customer(){
        byte id = _pid - 1;

        int numIters;
        for(numIters : 0 .. 2){
                choosingThread[id]=1; // Threads request lock

                int i, max=0;

                for(i : 0 .. (MAX_NUM_THREADS - 1)){ // loop across all threads that are active.
                // max is set to 0, if there is a ticketNumber that is greater than max, max = ticketNumber
                        if
                        :: ticketNumber[i] > max -> max = ticketNumber[i];
                        :: else;
                        fi;
                }

                // Take a ticket
                ticketNumber[id] = max + 1;
                choosingThread[id] = 0;

                int j;
                for(j : 0 .. (MAX_NUM_THREADS - 1)){
                        // Wait for our turn to come!
                        // if choosing[j] == 0, then we are on another thread and we must
                        // wait until it is our turn (choosingThread[j] == 1) threads turn to execute
                        do
                        :: (choosingThread[j] == 0) -> break;
                        od;

                        ((ticketNumber[j] == 0) || ((ticketNumber[j] >= ticketNumber[id]) && ((ticketNumber[j] != ticketNumber[id] || (j >= id))));
                }

                // This is the critical section of the code!
                // It will do some - call a function/ update a data stucture ETC

                // Release the lcok by clearing our ticket in ticketNumber
                ticketNumber[id] = 0;
        }
        printf("Process %d has competed %d iterations of this loop", id, numIters);
}

init {
        run customer();
        run customer();
        (_nr_pr == 1);
}
```

Breandan Kerin 1431016 27th September 2018

Advanced Computer Architecture

Tutorial 1 - Question 2

1. **Safety / Mutual Exclusion:**

Safety was determined by adding in an extra variable; *test_in_critical_section*, which was incremented in the critical section of thread execution in order to assert safety - that mutual exclusion is not violated.

```
// This is the critical section of the code!
// It will do some - call a function/ update a data stucture ETC
test_in_critical_section++;
assert(test_in_critical_section == 1);
test_in_critical_section--;
// Release the lcok by clearing our ticket in ticketNumber
```

Since the assertion is not violated - it was concluded that that there was only 1 thread accessing the critical section at any time - and hence that the Safety is ensured and mutual exclusion is no violated in the bakery lock.

Violation of safety in the bakery lock is achievable by removing line 10:

```
8       int numIters;
9       for(numIters : 0 .. 2){
10              // choosingThread[id]=1; // Threads request lock
11
```

Safety was violated here since neither of the threads were able to identify that the other thread had already requested the lock - which was the purpose of *choosingThread[id]* - this resulted in both threads accessing the *test_in_critical_section* variable in the critical section and caused the

assert to fail.

Breandan Kerin 1431016 27th September 2018

Advanced Computer Architecture

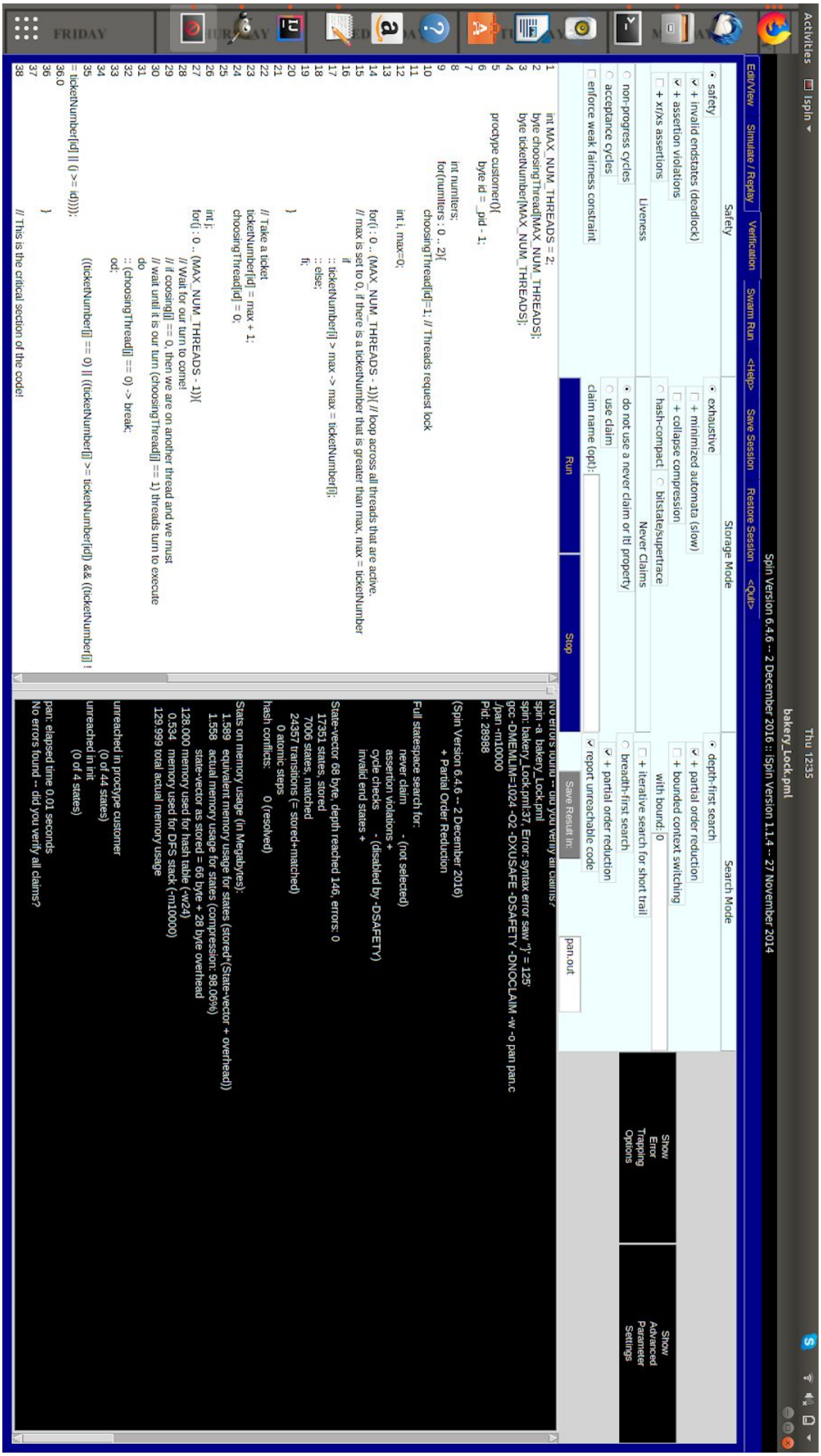
Tutorial 1 - Question 2

```
int MAX_NUM_THREADS = 2;
byte choosingThread[NUM_THREADS];
byte ticketNumber[NUM_THREADS];
int test_in_critical_section;

proctype customer(){
	byte id = _pid - 1;
	int numIters;
	for(numIters : 0 .. 2){
		// Comment the line below in order to break safety/mutual exclusion property
		choosing[id] = 1; // Threads request lock

		int i, max = 0;

		for(i : 0 .. (NUM_THREADS - 1)){ // loop across all threads that are active.
			if			// max is set to 0, if there is a ticketNumber that is greater than max, max = ticketNumber
			:: ticketNumber[i] > max -> max = ticketNumber[i];
			:: else;
			fi;
		}

		// Take a ticket
		ticketNumber[id] = max + 1;
		choosing[id] = 0;

		int j;
		for(j : 0 .. (NUM_THREADS - 1)){
			do
			:: (choosing[j] == 0) -> break;
			od;

			((ticketNumber[j] == 0) || ((ticketNumber[j] >= ticketNumber[id]) && ((ticketNumber[j] != ticketNumber[id] || (j >= id))));
		}

		// This is the critical section of the code!
		// It will do some - call a function/ update a data stucture ETC
		test_in_critical_section++;
		assert(test_in_critical_section);
		test_in_critical_section--;
		// Release the lcok by clearing our ticket in ticketNumber
		ticketNumber[id] = 0;
	}
	printf("Process %d has competed %d iterations of this loop", id, numIters);
}

init{
	run customer();
	run customer();
	(_nr_pr == 1);
}
```

Breandan Kerin 1431016 27th September 2018

Advanced Computer Architecture

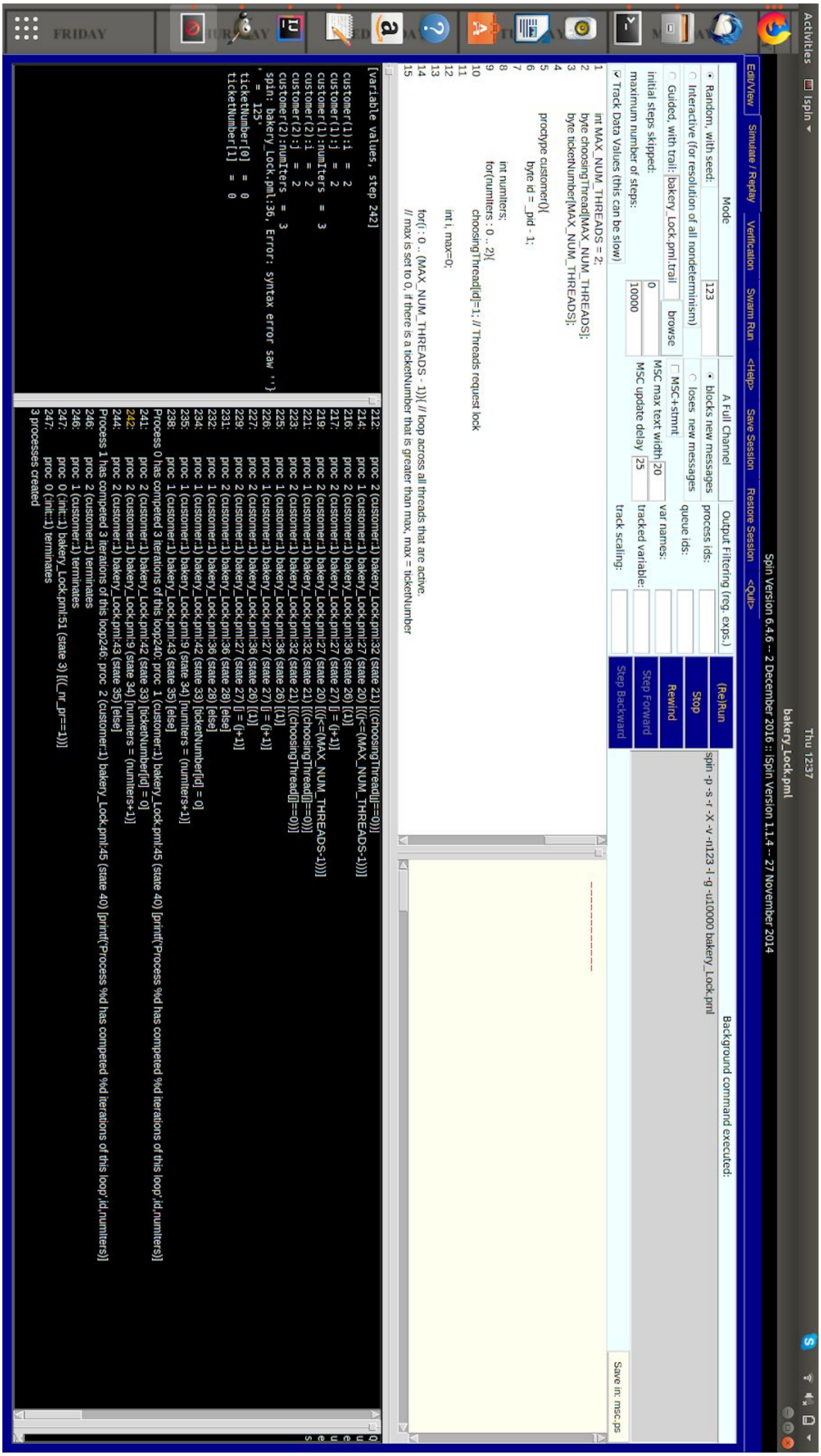
Tutorial 1 - Question 2



Breandan Kerin 1431016 27th September 2018

Advanced Computer Architecture

Tutorial 1 - Question 2



Breandan Kerin 1431016 27th September 2018

Advanced Computer Architecture

Tutorial 1 - Question 2

2. Deadlock free

Due to the fact that both threads were able to execute their instructions 3 times, each to completion without reaching an '*invalid end state*' in SPIN, allows me to conclude that the bakery lock implementation is deadlock free, and deadlock was not violated in the bakery lock implementation shown.

It can be noted that deadlock can be induced on the program by removing line 23:

```
22              // Take a ticket
23              ticketNumber[id] = max + 1;
24              // choosingThread[id] = 0;
25
26              int j;
27              for(j : 0 .. (MAX_NUM_THREADS - 1)){
28                      // Wait for our turn to come!
29                      // if choosing[j] == 0, then we are on another thread and we must
30                      // wait until it is our turn (choosingThread[j] == 1) threads turn to execute
31                      do
32                      :: (choosingThread[j] == 0) -> break;
33                      od;
34
```

This means that when the code enters the *do* section of the program - on lines 29-31 as per above, each of the two threads run would eventually enter an infinite loop - and hence break this expression. This is due to the fact that neither thread will be able to progress beyond this point until it knows the *choosingThread[id]* flag for the rest of the live threads has been reset back to 0; *choosingThread[id] = 0.*

Since the threads were not reset to 0, only one thread could enter the critical section and as a result none of the threads will be able to progress beyond this point since they are waiting on each other to no longer require the lock.

It is clear that, since the code had to be altered; removing line 23 - *choosingThread[id] = 0* - in order to cause deadlock to occur, it can be concluded that the bakery lock is deadlock free. From the screen shots below - it can be seen that SPIN detected deadlock due to the - '**invalid end state**' shown below.

Breandan Kerin   1431016   27th September 2018

Advanced Computer Architecture

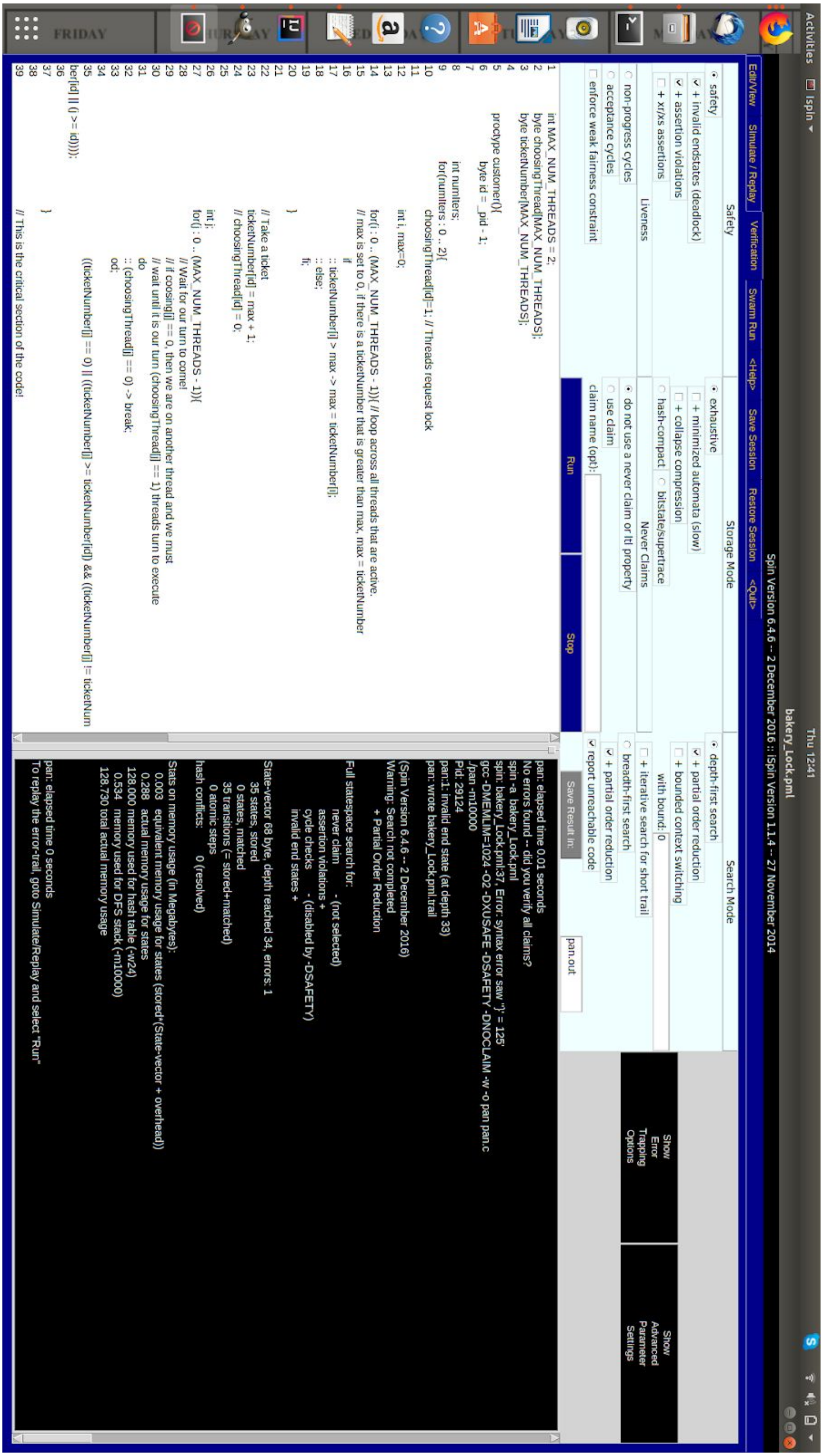
Tutorial 1 - Question 2

```
int MAX_NUM_THREADS = 2;
byte choosingThread[NUM_THREADS];
byte ticketNumber[NUM_THREADS];

proctype customer(){
        byte id = _pid - 1;

        int numIters;
        for(numIters : 0 .. 2){
                choosing[id] = 1; // Threads request lock

                int i, max = 0;

                for(i : 0 .. (NUM_THREADS - 1)){ // loop across all threads that are active.
                        if                       // max is set to 0, if there is a ticketNumber that is greater than max, max = ticketNumber
                        :: ticketNumber[i] > max -> max = ticketNumber[i];
                        :: else;
                        fi;
                }

                // Take a ticket
                ticketNumber[id] = max + 1;
                // choosing[id] = 0; // To remove deadlock from program - remove comment from this line!

                int j;
                for(j : 0 .. (NUM_THREADS - 1)){
                        // Wait for our turn to come!
                        // if choosing[j] == 0, then we are on another thread and we must wait until it is our turn (choosing[j] == 1) threads turn to execute
                        do
                        :: (choosing[j] == 0) -> break;
                        od;

                        ((ticketNumber[j] == 0) || ((ticketNumber[j] >= ticketNumber[id]) && ((ticketNumber[j] != ticketNumber[id] || (j >= id))));
                }

                // This is the critical section of the code!
                // It will do some - call a function/ update a data stucture ETC

                // Release the lcok by clearing our ticket in ticketNumber
                ticketNumber[id] = 0;
        }
        printf("Process %d has competed %d iterations of this loop", id, numIters);
}

init{
        run customer();
        run customer();
        (_nr_pr == 1);
}
```

Breandan Kerin 1431016 27th September 2018

Advanced Computer Architecture

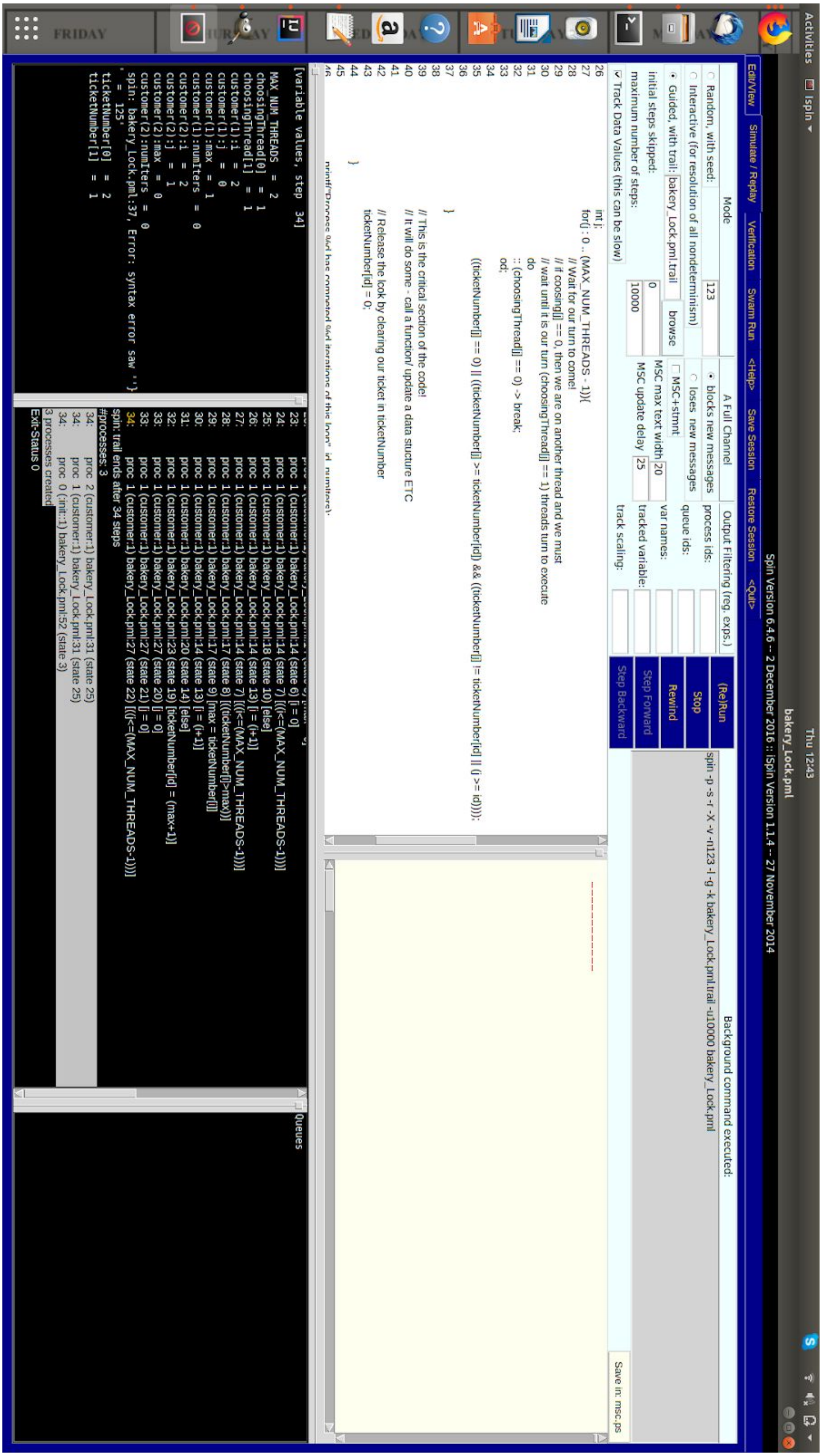
Tutorial 1 - Question 2



Breandan Kerin 1431016 27th September 2018

Advanced Computer Architecture

Tutorial 1 - Question 2



Breandan Kerin 1431016 27th September 2018

Advanced Computer Architecture

Tutorial 1 - Question 2

3. **Starvation free:**

The starvation free property was verified by using an assert in the promela code and by enforcing the weak fairness constraint.

Assert code:

assert(threadReachedCriticalSection[0] == eventually(threadReachedCriticalSection[1] == 3)

This is an assertion that will, for when the end state is reached, each of the two threads will be tested to ensure that they are equal to each other and 3.

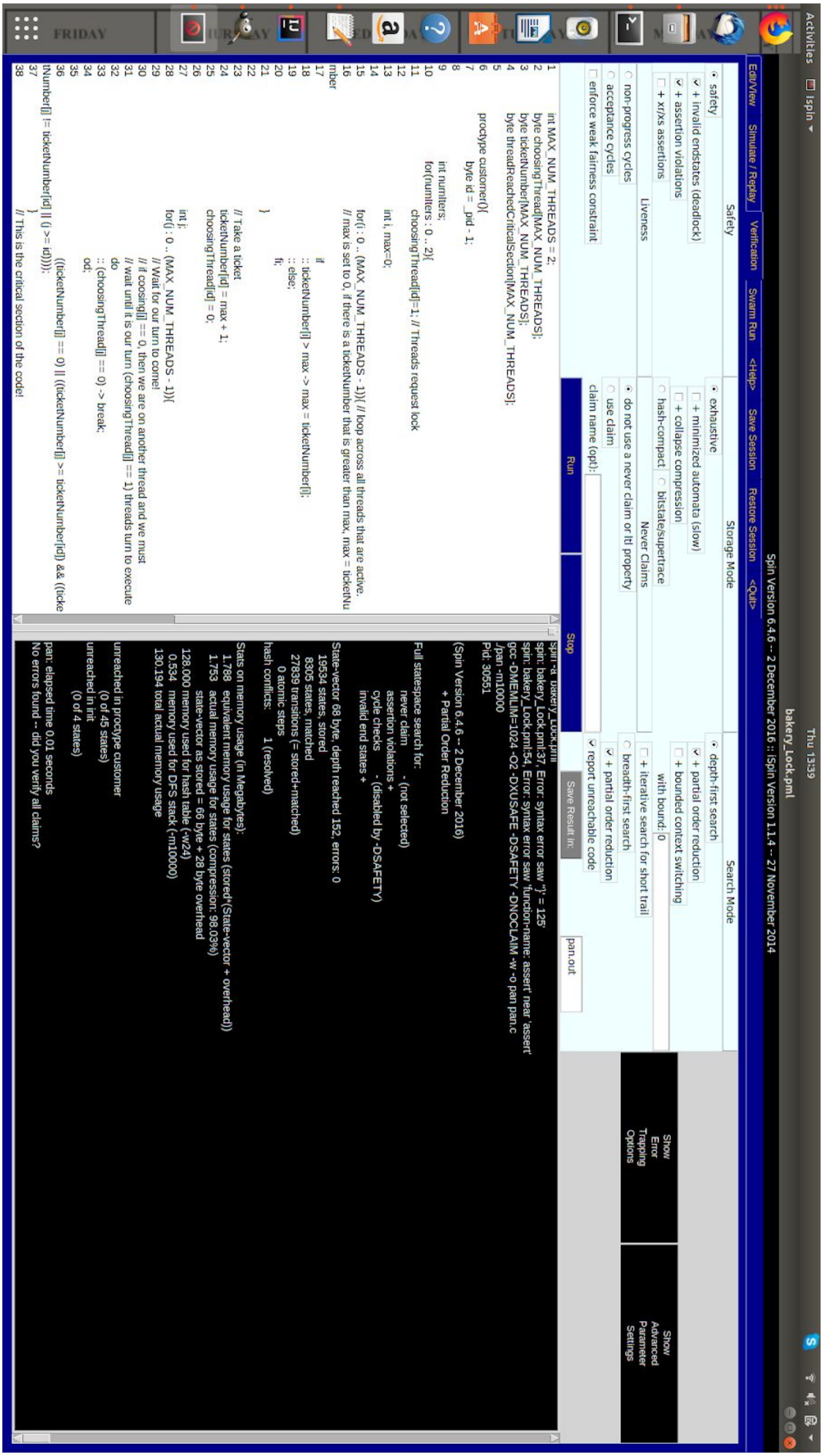
Since the claim didnt throw an assertion error when it was executed, this means that each thread was eventually able to reach the critical section and update the variable to go from 0 to 3. Hence, the starvation free property isn't violated in the bakery lock implementation.

```
int MAX_NUM_THREADS = 2;
byte choosingThread[MAX_NUM_THREADS];
byte ticketNumber[MAX_NUM_THREADS];
byte threadReachedCriticalSection[MAX_NUM_THREADS];

proctype customer(){
        byte id = _pid - 1;

        int numIters;
        for(numIters : 0 .. 2){
                choosingThread[id]=1; // Threads request lock

                int i, max=0;

                for(i : 0 .. (MAX_NUM_THREADS - 1)){ // loop across all threads that are active.
                // max is set to 0, if there is a ticketNumber that is greater than max, max = ticketNumber
                        if
                        :: ticketNumber[i] > max -> max = ticketNumber[i];
                        :: else;
                        fi;
                }

                // Take a ticket
                ticketNumber[id] = max + 1;
                choosingThread[id] = 0;

                int j;
                for(j : 0 .. (MAX_NUM_THREADS - 1)){
                        // Wait for our turn to come!
                        // wait until it is our turn (choosingThread[j] == 1) threads turn to execute
                        do
                        :: (choosingThread[j] == 0) -> break;
                        od;

                        ((ticketNumber[j] == 0) || ((ticketNumber[j] >= ticketNumber[id]) && ((ticketNumber[j] != ticketNumber[id] || (j >= id))));
                }
                // This is the critical section of the code!
                // It will do some - call a function/ update a data stucture ETC
                threadReachedCriticalSection[id]++;
                // Release the lcok by clearing our ticket in ticketNumber
                ticketNumber[id] = 0;
        }

        printf("Process %d has competed %d iterations of this loop", id, numIters);
}

init {
        run customer();
        run customer();
        (_nr_pr == 1);
}
```

Breandan Kerin 1431016 27th September 2018

Advanced Computer Architecture

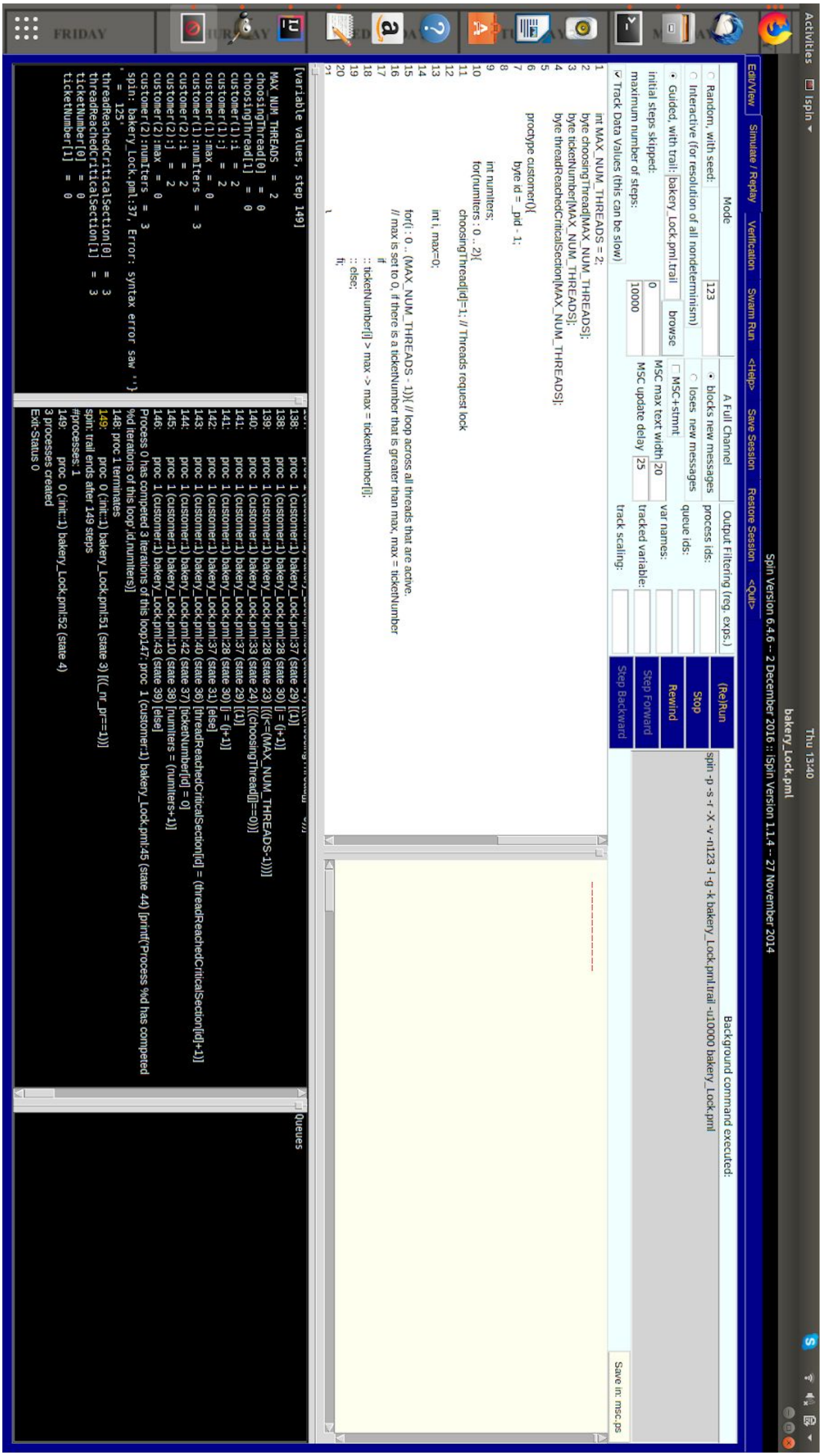
Tutorial 1 - Question 2



Breandan Kerin 1431016 27th September 2018

Advanced Computer Architecture

Tutorial 1 - Question 2



4. Liveness:

Due to the fact that the bakery lock implementation is starvation free, the implementation is proven to uphold the liveness property.
This is because, for the implementation to have liveness, at least one thread must be able to enter the critical section continually. For a program to be starvation free, all threads present must be able to enter the critical section eventually.

From this, the starvation free property of the implementation proven above shows that the liveness of this implementation is true.

The code and screenshots for the liveness property are seen above in the starvation free section of this report.