Nicholas Garrett

Professor Fouda

CS 4461

4/2/2022

Homework 6

Chapter 26

26.1

operation	dx
sub $1,%dx	0
test $0,%dx	-1
jgte .top	-1
halt	-1

.

26.2

The scripts run one after another with plenty of space and time to spare, so there is not a race condition.

operation	dx	thread
	3	0
sub $1,%dx	2	0
test $0,%dx	2	0
jgte .top	2	0
sub $1,%dx	1	0
test $0,%dx	1	0
jgte .top	1	0
sub $1,%dx	0	0
test $0,%dx	0	0
jgte .top	0	0
sub $1,%dx	-1	0
test $0,%dx	-1	0
jgte .top	-1	0
halt	-1	0

	3	1
sub $1,%dx	2	1
test $0,%dx	2	1
jgte .top	2	1
sub $1,%dx	1	1
test $0,%dx	1	1
jgte .top	1	1
sub $1,%dx	0	1
test $0,%dx	0	1
jgte .top	0	1
sub $1,%dx	-1	1
test $0,%dx	-1	1
jgte .top	-1	1
halt	-1	1

.

26.3

The much smaller interrupt frequency causes many more interrupts and switches between threads than when the frequency was every 100 operations.

26.4

operation	value at memory address 2000	thread
	0	0
mox 2000, %ax	0	0
add $1, %ax	0	0
mov %ax, 2000	1	0
sub $1, %bx	1	0
test $0, %bx	1	0
jgt .top	1	0
halt	1	0

.

26.5

operation	value at memory address 2000	thread
	0	0
mox 2000, %ax	0	0
add $1, %ax	0	0
mov %ax, 2000	1	0
sub $1, %bx	1	0
test $0, %bx	1	0
jgt .top	1	0
mox 2000, %ax	1	0
add $1, %ax	1	0
mov %ax, 2000	2	0
sub $1, %bx	2	0
test $0, %bx	2	0
jgt .top	2	0
mox 2000, %ax	2	0
add $1, %ax	2	0
mov %ax, 2000	3	0
sub $1, %bx	3	0
test $0, %bx	3	0
jgt .top	3	0
halt	3	0

operation	value at memory address 2000	thread
	3	1
mox 2000, %ax	3	1
add $1, %ax	3	1
mov %ax, 2000	4	1
sub $1, %bx	4	1
test $0, %bx	4	1
jgt .top	4	1
mox 2000, %ax	4	1
add $1, %ax	4	1
mov %ax, 2000	5	1
sub $1, %bx	5	1
test $0, %bx	5	1
jgt .top	2	1
mox 2000, %ax	5	1
add $1, %ax	5	1
mov %ax, 2000	6	1
sub $1, %bx	6	1
test $0, %bx	6	1
jgt .top	6	1
halt	6	1

The final value of Value is 6. And the code loops 3 times because the value bx is set to 3 initially. Each iteration subtracts 1 from this value until it reaches 0.

26.6

For seed 0, it looks like the two threads operate the add 1 command only once; and since they share a variable space, that would mean that the final value for Value is $value_0 + 2$, or 2.

However, for seed 1, the second thread increments the Value variable, but since thread 0 does not move that updated value into its local register, it increments the local value and moves it into the shared variable. Therefore, it is equivalent to the value being incremented only once.

The timing of the interrupt does matter, as if the interrupt occurs between either thread moves the value of an address space into a local register, it could cause problems where there are shared variables. This area is the critical section: where the value at 2000 is moved into a variable address, is incremented, and then updated.

26.7

For i = 1, the outcome is practically similar to the code run in seed 1 in problem 26.6. The two threads have their critical sections split up, so the effect is that only a single thread's incrementation is written. Technically, they are both written, but Thread 1 overwrites the result given in Thread 0.

The same outcome occurs when i =2.

However, when the interrupt interval is 3, the critical secions–as defined in problem 26.6–are able to run atomically, so the outcome of the shared variable is 2.

26.8

For the code, the best outcome is when both threads run their critical sections, so when the value of Value is 200 on completion.

Through my experimentation, I found that this was the outcome when I set the interrupt interval to values that were multiples of 3. For example, 3, 6, 9,12, 15, 18, etc.

26.9

Using the code's thread trace as a reference, this code should check that the variable ax is 1, move it into the shared variable space, and halt, or simply loop if it is not. There is no incrementation code, so thread 1 must initially have the value of its variable be 1, or else the program will loop.

The value at 2000 could almost be described as an address space holding data that is sent between the two threads, as thread 0 sets it to 1, and that value is read by thread 1–thus causing a certain outcome. At the end, the final value of 2000 should be 1.

26.10

I did a bit of experimentation with this on problem 26.9, and the outcome was briefly described therein. Under this configuration, the program goes through the code for thread 0, then goes to thread 1's code. Since thread 1 must run for the conditions of thread 0 to allow for a halt outcome, this is an inefficient use of CPU resources.

Smaller increment counters cause the two threads to flip-flop between one another more quickly, making this script a bit more efficient than otherwise. However, when it is set to a large value, thread 0 runs for a while before the halt condition caused by thread 1 can occur.

Chapter 28

28.1

Looking at the code, it looks like it runs by using a flag variable location and using the value at that location to determine whether another process or thread is using it. When the value at that address is 0, the process moves a 1 into it, effectively setting the flag to "in use". It then runs the critical section of incrementing the count variable, and then releases the flag by setting it to 0.

The process then checks if it is expected to continue running through the variable bx, and loops accordingly.

28.2

With the default values, it looks like it works correctly, though the processes run fully before switching.

I would predict that since thread 1 executes last and sets the value of flag to 0, the logical outcome is that flag will be 0.

28.3

This new addition to the script sets the initial values of bx in threads 0 and 1 to 2. Because of this, two threads will run the code three times. With this new addition, I do not think it will change the outcome of the flag variable will be different.

28.4

Testing indicated that as there are 11 operations, the lowest, most efficient value to set for the interrupt frequency is 11. Beyond that, I could not determine a mathematical relation to describe the most efficient values. Values below 11 lead to undesirable outcomes that are below the best return.

28.5

Looking at the code, it looks to work along the lines that the variable is swapped with a 1, and is checked to see if it was a 1 already. If it was, then the lock is active and no change has occurred to that state, but if it was not already locked, the step of swapping it by nature locks it. Then, to release the lock, a 0 is simply written into it.

28.6

In answer to this question, I partly included it in my response for 28.4. To explain further, initial testing implied that $11x : x \in R$ represented the set of efficient values for i. However, I discovered that 15 and 21 were also values that had perfect outcomes, which led to an implied equation of $11x + (4 + x)y : x \in R and y \in W$.

However, further testing indicated that this relationship still did not describe the sequence properly, as values in this equation, when tested, (45) returned sub-optimum results. So, I do not know exactly how to describe this mathematical relation.

A way to quantify efficiency could be through iterations run without variables being changed and the presence or existence of race conditions.

Chapter 29

29.1

As the code for gettimeofday() provides functionality for microseconds, that would logically be the accuracy of this function.

29.2