**NICHOLAS CASTEEN**

**CSC 345 002**

**4/25/15**

Output from Part 1(a):

Sum = 500000

What I observed here was one of the more basic aspects of code: arguments passed into a function. Here, we gave an offset to a function, counting\_function.

Output from Part 1(b):

Sum = 50000000

Because offset is 100 instead of 1.

What I observed was that the program took a variable and passed a pointer to it as an argument, rather than the variable itself. The meant that the function had access to the original variable rather than making a copy for its own use. The code then proceeded to dereference the pointer, ie read the value at the address the pointer was pointing at, and copy that value into a function-specific variable anyway.

Output from Part2(a):

Sum = 50000000

What I observed here was the use of thread as a fancy way of calling a function. Create a thread object, then start it up with a given function and the arguments it needs to do its thing. The main function and the thread's counting\_function function ran in tandem...even if the main function just immediately waited for the thread to finish its job. The pthread\_join made the main function wait for the thread to finish, and I said the join function was useful because it allowed you specify which thread the calling function should wait on.

Output from Part 2(b):

Sum = 4295048481

Hmm..hm, call me crazy, but I don't think that's quite what we want.

Here, two threads were spawned, both doing the exact same function, but with different offsets. Allowed to run willy-nilly, all kinds of lower-level producer-consumer problem issues cropped up in one big red hot mess. If only we had some way to control which thread could access what variables at any time so they could take turns.

Output from Part 2(c):

Sum = 0

Oh wait, we do! The mutex acted as a roadblock, allowing only one thread at a time past the mutex\_lock function. Once a function called mutex\_unlock, other threads were allowed to lock the mutex for themselves. As the two threads took turns, one with offset 1 and the other with offset -1, they both ran 500000 times. That means one netted +500000 and the other netted -500000, resulting in 0 overall for the global variable at the end.

Part 3:

Edited code:

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

#define MAXSIZE 100

#define ITERATIONS 1000

int buffer[MAXSIZE]; // buffer. An array of ints. Classic.

int nextp, nextc; // temporary storage. For what? Pointer addresses, I'm willing to bet.

int count=0; //S'right there on the label. A counter.

int ProFin=0; //Is the producer done? That's what this variable will keep track of.

pthread\_mutex\_t mutex=PTHREAD\_MUTEX\_INITIALIZER; //Awh, a semaphore! Calling it mutex1 in case I want to use another gate. I doubt I'll need a second one. EDIT: Got rid of the one.

//Don't worry about this. We only get to this function once the threads are dead anyway.

void printfunction(void \* ptr)

{

int count = \*(int \*) ptr; //End goal. count here copies count global.

if (count==0) //If count is zero, then the producer has nothing else to give the consumer.

{

printf("All items produced are consumed by the consumer \n"); //This is what we want. After all, we can't have a producer that makes stuff the consumer misses.

}

else

{

for (int i=0; i<=count; i=i+1)

{

printf("%d, \t",buffer[i]); //Here, if SOMEHOW the consumer ends without taking all the producer stuff, we'll see all the left over stuff.

}

printf("\n");

}

}

void \*producer(void \*ptr)

{

int item, flag=0; //2 integers, on initialized as 0. I guess this'll be an on/off flag. Roit.

int in = \*(int \*) ptr; //In...okay, so we copy from count's address.

do

{

pthread\_mutex\_lock(&mutex);

//CRITICAL SECTION. We'll edit variables for the Consumer. PRODUCER /MUST/ GO FIRST. Why? Well, the consumer needs stuff to...you know. Consume.

item = (rand()%7)%10; //Okay, make a random number. Fine enough. I do wonder if this'll be critical section, since this has the same name as consumer's item. Might be an issue. //Though, it seems item gets a new value every iteration, so it shouldn't matter so long as the consumer and producer are taking turns.

flag=flag+1; //Literally just a counter SPECIFICALLY for the producer.

nextp=item; //nextp? Oh, that's a global int. Yup, we're in critical section. So we save our completely random variable in nextp. Dunno why there's an intermediate nextp value, but whatevs.

buffer[in]=nextp; //Count starts at 0, so will 'in'. In BASICALLY increments by 1 every cycle, rolling over to zero at 100.

in=((in+1)%MAXSIZE); //Next one in the array. I guess the mod MAXSIZE makes it so, if we go over 100, we just roll over back to 0. Clever.

count=count+1; //Add one to count. Count says how many thing we have to consume. It's a global variable, so Consumer can see it too.

//All that guff with adding to count a lot until it equaled MAXSIZE was removed. I have no idea what it was there for, logically. Would just waist time and not be very helpful, as I want //to use count as, literally, a count of all the things the producer has, at any point, for the consumer to consume.

//CRITICAL SECTION END

pthread\_mutex\_unlock(&mutex); //AAAAAND unluock

}

while (flag<=ITERATIONS); //So flag has to go up to ITERATIONS...or, 1000.

ProFin=1; //Once we're out of the loop, change the ProFin flag to 1. Consumer only reads this, so it's okay.

pthread\_exit(NULL); //Well, whatever happened, we're done once flag's done with its iterations.

}

void \*consumer(void \*ptr)

{

//Okay, this bit is just reading CS variables. No trouble.

int item, flag=ITERATIONS; //These might be okay, because they're defined as their own variables for this function. Let's see if we can suss out what this part does. Consumer consumes, right?

int out = \*(int \*) ptr; //Okay, so, 'out', like 'in', takes count, the global variable.

do

{

pthread\_mutex\_lock(&mutex);

//CRITICAL SECTION. Reads Count which can be edited by Producer at the same time.

while (count >0) //count being above zero means there's things to consume!

{

nextc = buffer[out]; //next c...???? That's a global as well, only used by this one.

out=(out+1)%MAXSIZE; //What is this? Rolling over, just like 'in' in the producer.

printf("\t\tCount = %d in consumer at Iteration = %d\n", count, flag); //So, we read the buffer at the next index into nextc. That's nice.

count = count-1; //That's one less thing to consume.

flag=flag-1; //One less flag. Just used for the flair up above, but has nothing to do with the while loop.

}

if (count <= 0 && ProFin == 0) //If count is equal to zero, producer ain't made anything yet. Buuuut, if ProFin is 1, then we're not faster than producer...it's just not doing anything.

{

printf("consumer made to wait...faster than producer.\n");

}

pthread\_mutex\_unlock(&mutex);

//CRITICAL SECTION END. Flag is Cosumer's OWN variable.

}

while (ProFin == 0 || count > 0); //While the producer isn't done stuffing my fat face, AMIRITE????

pthread\_exit(NULL);

}

int main(void) //So, first, let's figure out what we're wanting to happen here.

{

int in=0, out=0; //pointers

pthread\_t pro, con; //Two threads. Producer and Consumer

int rc1;

int rc2; //I guess there are error things

// Spawn threads

rc1 = pthread\_create(&pro, NULL, producer, &count); //So, start one, with count

rc2 = pthread\_create(&con, NULL, consumer, &count); //Start the other, also with count

//These creation functions will return integers into rc1 and rc2, respecitively. If there's no trouble, then both rc1 and 2 should be zero.

//Now, the magic happens in producer and consumer

if (rc1) //If this is true?

{

printf("ERROR; return code from pthread\_create() is %d\n", rc1); //I guess there was an error

exit(-1);

}

if (rc2) //Same here. Error checker...?

{

printf("ERROR; return code from pthread\_create() is %d\n", rc2);

exit(-1);

}

// Wait for the threads to finish

// Otherwise main might run to the end

// and kill the entire process when it exits.

pthread\_join(pro, NULL);

pthread\_join(con, NULL);

printfunction(&count); //Here's where we end it.

}

OUTPUT (Tail-End):

Count = 105 in consumer at Iteration = 104

Count = 104 in consumer at Iteration = 103

Count = 103 in consumer at Iteration = 102

Count = 102 in consumer at Iteration = 101

Count = 101 in consumer at Iteration = 100

Count = 100 in consumer at Iteration = 99

Count = 99 in consumer at Iteration = 98

Count = 98 in consumer at Iteration = 97

Count = 97 in consumer at Iteration = 96

Count = 96 in consumer at Iteration = 95

Count = 95 in consumer at Iteration = 94

Count = 94 in consumer at Iteration = 93

Count = 93 in consumer at Iteration = 92

Count = 92 in consumer at Iteration = 91

Count = 91 in consumer at Iteration = 90

Count = 90 in consumer at Iteration = 89

Count = 89 in consumer at Iteration = 88

Count = 88 in consumer at Iteration = 87

Count = 87 in consumer at Iteration = 86

Count = 86 in consumer at Iteration = 85

Count = 85 in consumer at Iteration = 84

Count = 84 in consumer at Iteration = 83

Count = 83 in consumer at Iteration = 82

Count = 82 in consumer at Iteration = 81

Count = 81 in consumer at Iteration = 80

Count = 80 in consumer at Iteration = 79

Count = 79 in consumer at Iteration = 78

Count = 78 in consumer at Iteration = 77

Count = 77 in consumer at Iteration = 76

Count = 76 in consumer at Iteration = 75

Count = 75 in consumer at Iteration = 74

Count = 74 in consumer at Iteration = 73

Count = 73 in consumer at Iteration = 72

Count = 72 in consumer at Iteration = 71

Count = 71 in consumer at Iteration = 70

Count = 70 in consumer at Iteration = 69

Count = 69 in consumer at Iteration = 68

Count = 68 in consumer at Iteration = 67

Count = 67 in consumer at Iteration = 66

Count = 66 in consumer at Iteration = 65

Count = 65 in consumer at Iteration = 64

Count = 64 in consumer at Iteration = 63

Count = 63 in consumer at Iteration = 62

Count = 62 in consumer at Iteration = 61

Count = 61 in consumer at Iteration = 60

Count = 60 in consumer at Iteration = 59

Count = 59 in consumer at Iteration = 58

Count = 58 in consumer at Iteration = 57

Count = 57 in consumer at Iteration = 56

Count = 56 in consumer at Iteration = 55

Count = 55 in consumer at Iteration = 54

Count = 54 in consumer at Iteration = 53

Count = 53 in consumer at Iteration = 52

Count = 52 in consumer at Iteration = 51

Count = 51 in consumer at Iteration = 50

Count = 50 in consumer at Iteration = 49

Count = 49 in consumer at Iteration = 48

Count = 48 in consumer at Iteration = 47

Count = 47 in consumer at Iteration = 46

Count = 46 in consumer at Iteration = 45

Count = 45 in consumer at Iteration = 44

Count = 44 in consumer at Iteration = 43

Count = 43 in consumer at Iteration = 42

Count = 42 in consumer at Iteration = 41

Count = 41 in consumer at Iteration = 40

Count = 40 in consumer at Iteration = 39

Count = 39 in consumer at Iteration = 38

Count = 38 in consumer at Iteration = 37

Count = 37 in consumer at Iteration = 36

Count = 36 in consumer at Iteration = 35

Count = 35 in consumer at Iteration = 34

Count = 34 in consumer at Iteration = 33

Count = 33 in consumer at Iteration = 32

Count = 32 in consumer at Iteration = 31

Count = 31 in consumer at Iteration = 30

Count = 30 in consumer at Iteration = 29

Count = 29 in consumer at Iteration = 28

Count = 28 in consumer at Iteration = 27

Count = 27 in consumer at Iteration = 26

Count = 26 in consumer at Iteration = 25

Count = 25 in consumer at Iteration = 24

Count = 24 in consumer at Iteration = 23

Count = 23 in consumer at Iteration = 22

Count = 22 in consumer at Iteration = 21

Count = 21 in consumer at Iteration = 20

Count = 20 in consumer at Iteration = 19

Count = 19 in consumer at Iteration = 18

Count = 18 in consumer at Iteration = 17

Count = 17 in consumer at Iteration = 16

Count = 16 in consumer at Iteration = 15

Count = 15 in consumer at Iteration = 14

Count = 14 in consumer at Iteration = 13

Count = 13 in consumer at Iteration = 12

Count = 12 in consumer at Iteration = 11

Count = 11 in consumer at Iteration = 10

Count = 10 in consumer at Iteration = 9

Count = 9 in consumer at Iteration = 8

Count = 8 in consumer at Iteration = 7

Count = 7 in consumer at Iteration = 6

Count = 6 in consumer at Iteration = 5

Count = 5 in consumer at Iteration = 4

Count = 4 in consumer at Iteration = 3

Count = 3 in consumer at Iteration = 2

Count = 2 in consumer at Iteration = 1

Count = 1 in consumer at Iteration = 0

All items produced are consumed by the consumer

What I saw here was a producer that created random variables and put them into an array for the consumer to read. The consumer had to wait for the array to have unread items before it could proceed. The producer made use of a count and an int called ProFin. Count was a global integer that could let the consumer know how many items were left to consume, if any. ProFin only existed to let the consumer know the producer had no more items to give.

In the end, all the items are always consumed by the consumer, as the consumer is forced to deplete the count before completion, taking every item into nextc. I assume nextc would be used for other, hypothetical purposes, in a real producer-consumer scenario.