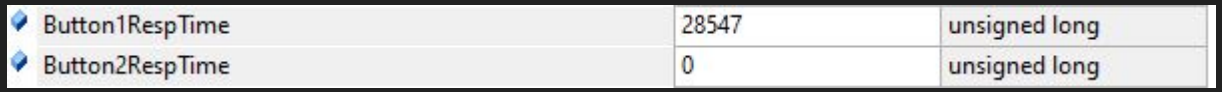
Mini Project 4

Yimin Gao
yg9bq@virginia.edu

Part 1: Experiment 1

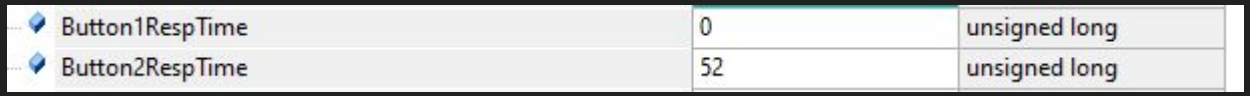
Name	Value	Type
ConsumerCount	1	unsigned long
DisplayCount	2476	unsigned long
tcbs[1].ArriveTime	2544	unsigned int
tcbs[1].WaitTime	0	unsigned int
tcbs[1].ExecCount	7497	unsigned int
tcbs[3].ArriveTime	2544	unsigned int
tcbs[3].WaitTime	2	unsigned int
tcbs[3].ExecCount	7494	unsigned int

Experiment 2



Button1RespTime	28547	unsigned long
Button2RespTime	0	unsigned long

When I pressed button1, it didn’t response until the the last NumSamples of lifetime since the semaphore was occupied by other threads.

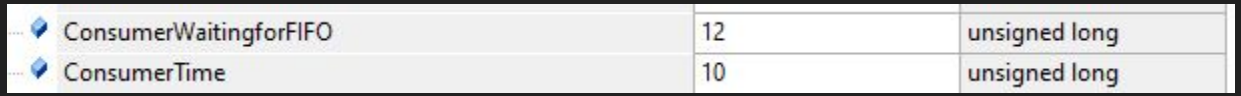


Button1RespTime	0	unsigned long
Button2RespTime	52	unsigned long

When I pressed button2, it responded right away since it reset the NumSamples to RUNLENGTH at first to get both consumer and display thread killed (and perhaps buttonwork1 if created). So it can be executed then since the semaphore was not occupied by any of those threads.

Question 1:

Task 2: Every time I pressed button1, the NumCreated incremented by one but the foreground thread created (ButtonWork) was not executed until the end of lifetime. The reason is that the semaphore was always occupied by the other threads. (Mostly by Display thread)



ConsumerWaitingforFIFO	12	unsigned long
ConsumerTime	10	unsigned long

Task 3: The consumer has the same execution count as display. However, the semaphore is always occupied by display. The consumer count is a lot less than display count. I have added another two variables

ConsumerTime: The time from consumer got the semaphore &LCDFree to the time consumer released the semaphore.

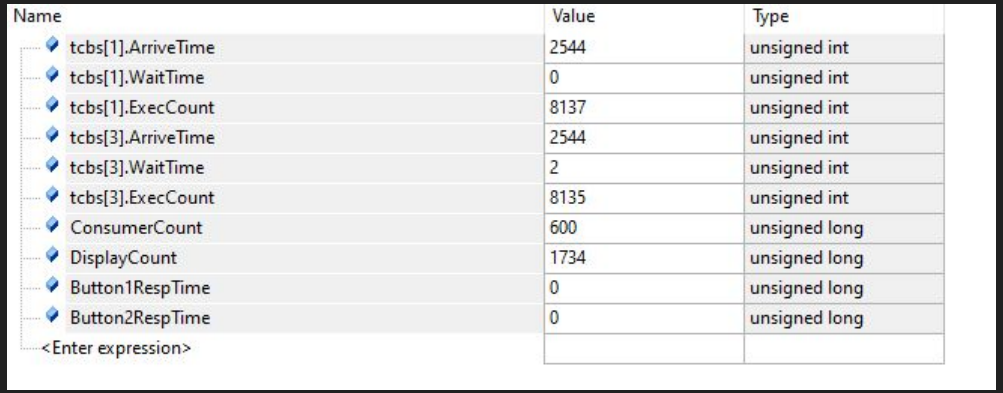
ConsumerWaitingforFIFO: The time from consumer released the semaphore &LCDFree to the time consumer got the semaphore &data (for FIFO)

It showed that the consumer spends more than half of the time waiting for FIFO semaphore. However, the display thread doesn't have such time waiting for other semaphore. That is why display always occupied the semaphore (LCDFree) while consumer didn't.

Task 6: The foreground thread display doesn't have any other semaphore limitation. It only needs the semaphore (LCDFree) and periodically updates the PseudoCount to LCD display. Thus it always occupies the semaphore.

Task 7: Every time I pushed the button2, the screen showed up restarting and all threads were reset. The reason that the created foreground thread Restart is able to get the semaphore is that before asking for the semaphore, it reset the value of NumSamples to RUNLENGTH to stop both both consumer and display thread killed (and perhaps buttonwork1 if created) threads. Once the other threads occupied the semaphore were killed, the thread Restart is able to run.

Question 2:



Name	Value	Type
tcbs[1].ArriveTime	2544	unsigned int
tcbs[1].WaitTime	0	unsigned int
tcbs[1].ExecCount	8137	unsigned int
tcbs[3].ArriveTime	2544	unsigned int
tcbs[3].WaitTime	2	unsigned int
tcbs[3].ExecCount	8135	unsigned int
ConsumerCount	600	unsigned long
DisplayCount	1734	unsigned long
Button1RespTime	0	unsigned long
Button2RespTime	0	unsigned long
<Enter expression>		

After adding the suspend in display thread, the thread automatically jumped to the next one after releasing the semaphore. Thus consumer got to execute a lot more this time. However, the consumer spent so much time waiting for the FIFO semaphore to be ready (ConsumerWaitingforFIFO = 40ms).Every time the consumer didn't have the FIFO semaphore ready, it automatically called OS_Suspend to go to the next thread, causing that the displaycount was still a lot greater than consumercount.

Question 3

Name	Value	Type
tcbs[1].ArriveTime	2544	unsigned int
tcbs[1].WaitTime	0	unsigned int
tcbs[1].ExecCount	18822	unsigned int
tcbs[3].ArriveTime	2544	unsigned int
tcbs[3].WaitTime	0	unsigned int
tcbs[3].ExecCount	18819	unsigned int
ConsumerCount	600	unsigned long
DisplayCount	4405	unsigned long
Button1RespTime	0	unsigned long
Button2RespTime	0	unsigned long
<Enter expression>		

The display count this time was a lot greater than the round robin case in question 2. The sleep function in question 2 in Display thread works the same as the OS_Supend(), since they all go to the next thread and they are ready before the next time Display should run.

The consumer thread waited for the FIFO semaphore most of the time. The wait function calls OS_Suspend, so consumer function works almost the same as cooperative scheduling in question2 as well.

However, with cooperative scheduling implemented, the other two threads(tcbs[0], tcbs[2]) runs a lot less, causing the number of displaycounted increasing a lot. Since the FIFO only got fixed amount of data in fixed time, the ConsumerCount is the same.

Part 2

Experiment 1

SignalCount1	20000	unsigned long
SignalCount2	20000	unsigned long
SignalCount3	1960000	unsigned long
WaitCount1	1983960	unsigned long
WaitCount2	11563	unsigned long
WaitCount3	4477	unsigned long

Experiment 1

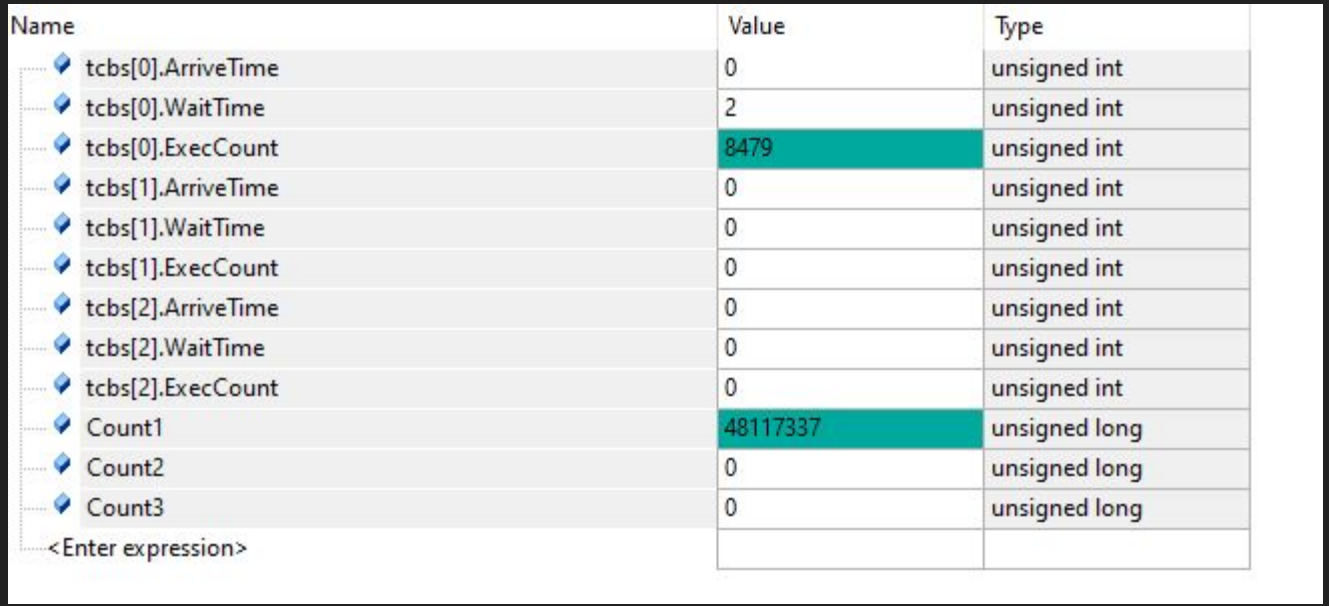
SignalCount1	20000	unsigned long
SignalCount2	20000	unsigned long
SignalCount3	1960000	unsigned long
WaitCount1	1999869	unsigned long
WaitCount2	71	unsigned long
WaitCount3	60	unsigned long

The cooperative spinlock semaphore cannot dynamically assign the status of each thread. Thus, thread one will not be disabled whenever it waits for the semaphore. Each time the semaphore is ready, thread one will execute first. Then if the semaphore is more than one before wait1 executed, wait2 and wait3 can then be executed.

However, for experiment1 each time a thread waited for the semaphore, it would be blocked. The blocking semaphore placed blocked threads on a list, sorted by the order they got blocked. Thus Wait2 and Wait3 got more chances to run.

In all two experiments, the sum of signal count = the sum of wait count = 2000000

Part 3: Experiment 1



Name	Value	Type
tcbs[0].ArriveTime	0	unsigned int
tcbs[0].WaitTime	2	unsigned int
tcbs[0].ExecCount	8479	unsigned int
tcbs[1].ArriveTime	0	unsigned int
tcbs[1].WaitTime	0	unsigned int
tcbs[1].ExecCount	0	unsigned int
tcbs[2].ArriveTime	0	unsigned int
tcbs[2].WaitTime	0	unsigned int
tcbs[2].ExecCount	0	unsigned int
Count1	48117337	unsigned long
Count2	0	unsigned long
Count3	0	unsigned long
<Enter expression>		

Part 3: Experiment 2

(a)

Name	Value	Type
tcbs[1].ArriveTime	2546	unsigned int
tcbs[1].WaitTime	0	unsigned int
tcbs[1].ExecCount	2401	unsigned int
tcbs[3].ArriveTime	2546	unsigned int
tcbs[3].WaitTime	2	unsigned int
tcbs[3].ExecCount	6735	unsigned int
DisplayCount	2045	unsigned long
ConsumerCount	600	unsigned long
<Enter expression>		

(b)

Name	Value	Type
tcbs[1].ArriveTime	2546	unsigned int
tcbs[1].WaitTime	0	unsigned int
tcbs[1].ExecCount	2401	unsigned int
tcbs[3].ArriveTime	2546	unsigned int
tcbs[3].WaitTime	2	unsigned int
tcbs[3].ExecCount	6732	unsigned int
DisplayCount	2044	unsigned long
ConsumerCount	600	unsigned long
Button1RespTime	0	unsigned long
Button2RespTime	0	unsigned long
<Enter expression>		

Question 1:

In mini project 2, the count of three threads are almost the same. However, with fixed priority scheduler, the scheduler always chose the thread with the highest priority. That is the reason why count2 and count3 are zero.

Question 2:

The difference is that the thread Consumer got to run a lot more, while both buttons didn't respond.

The reason thread Consumer executed a lot more is that with fixed priority scheduling, Consumer has the highest priority. However, it didn't hold the semaphore the whole time since sometimes it had to wait for the FIFO semaphore. The wait function calls OS_Supsend to the next thread.

The reason both buttons didn't work is that the foreground threads they added has the lowest priority. So with fixed priority scheduling, they will never be executed until all other threads are done executing

.

Part4: Experiment 1:

Name	Value	Type
tcbs[1].ArriveTime	0	unsigned int
tcbs[1].WaitTime	10	unsigned int
tcbs[1].ExecCount	1860	unsigned int
Count1	41587731	unsigned long
Count2	10526436	unsigned long
Count3	5784330	unsigned long
tcbs[0].ArriveTime	0	unsigned int
tcbs[0].WaitTime	2	unsigned int
tcbs[0].ExecCount	7347	unsigned int
tcbs[2].ArriveTime	0	unsigned int
tcbs[2].WaitTime	18	unsigned int
tcbs[2].ExecCount	1022	unsigned int
tcbs[1].age	0	unsigned int
<Enter expression>		

Part4: Experiment 2 (a)

Name	Value	Type
tcbs[1].ArriveTime	2548	unsigned int
tcbs[1].WaitTime	0	unsigned int
tcbs[1].ExecCount	2401	unsigned int
tcbs[3].ArriveTime	2548	unsigned int
tcbs[3].WaitTime	2	unsigned int
tcbs[3].ExecCount	6891	unsigned int
ConsumerCount	600	unsigned long
DisplayCount	2097	unsigned long

Part4: Experiment 2 (b)

Button1RespTime	19	unsigned long
Button2RespTime	59	unsigned long

Question 1:

Count2 and count3 got to increment a lot more than before. The foreground thread with the highest priority (Thread1b) didn’t run all the time. Instead, with dynamic priority scheduling, the tasks with lower priority got to upgrade their WorkPriority once they waited for 8ms. The result is that the ranking of the count is exactly the same as they should be (same as the ranking of priority).

Question 2:

When using dynamic priority scheduling and blocking semaphore, the Consumer ran a lot more than experiment 1 in part 1. The reason is that we set the Consumer thread the highest priority.

Also, the response of button1 and button2 is really fast this time. That is because of the aging scheduler. Once the thread didn’t run in the past 8 ms, they will upgrade their priority(decrease by 1) and eventually got executed.

Question 3

In part 3 experiment b, both buttons didn’t respond. However, in part 4 experiment b, both buttons responded really fast. The reason is that in part3 with fixed priority scheduling, the threads that both button added had the lowest priority. The semaphore is always occupied by the Consumer thread and the display thread.(When consumer waited for FIFO semaphore, the LCD semaphore will be occupied by display thread.) Thus the threads created by the button push will never execute.

However, with dynamic priority scheduling, the threads (ButtonWork and Restart) created by button push dynamically updated their priority and got executed every time I pushed the button

Thank you for your feedback on Mini Project 4.