Header: Grade: FOR /20 DES /40 EXP /30 ORI /10 / TOT _100
Lab # 2

CS 5390 Summer 2017, Date of Submission: 07/07/17
Students Names: Adeel Malik (amalik@utep.edu)
Instructor: Yadira Jacquez

Section 1: Effort: 12 hours
- Planning and preparation: 2 hours
- Experiment: 9 hours (on simulator)
- Report writing: 1 hours

Section 2: Objectives
The objective of this experiment is to understand how to run multiple tasks and accessing shared data. Understanding of different types of semaphores and their characteristics, synchronization. Learning VxWorks scheduling algorithms.

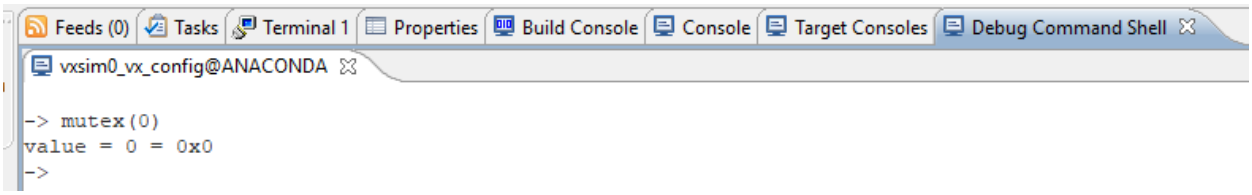
Section 3: Procedures and Results

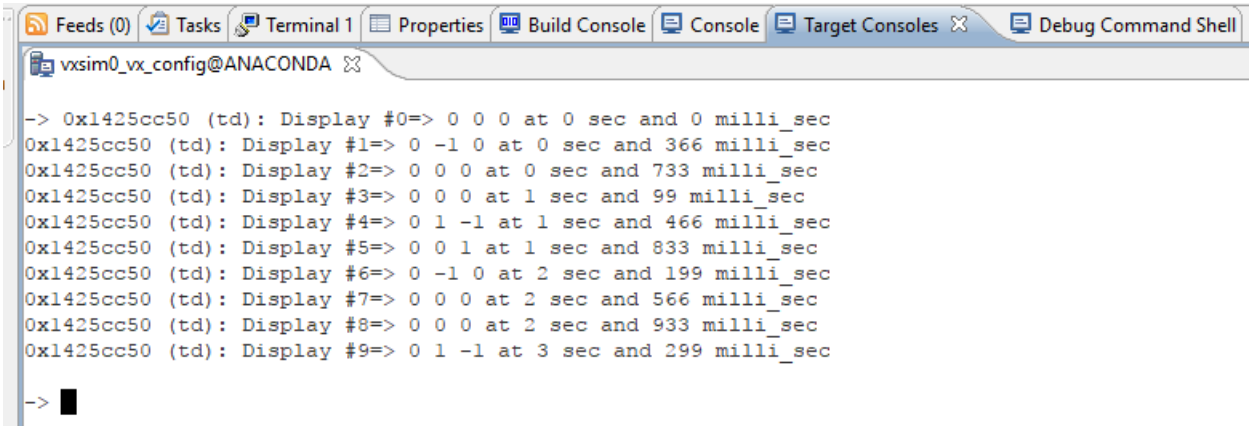
Part A: Mutual Exclusion

A1. Build and download the object file (*mutex.o*) and then execute *mutex* function from the shell both without and with the semaphore protection (argument *protect* 0 or 1 respectively). The function to be used is *mutex* with an argument either zero or one. If the mutex semaphore (*semMtx*) is to be used, we need to create it - either from the shell line or executing the provided function *createM*.

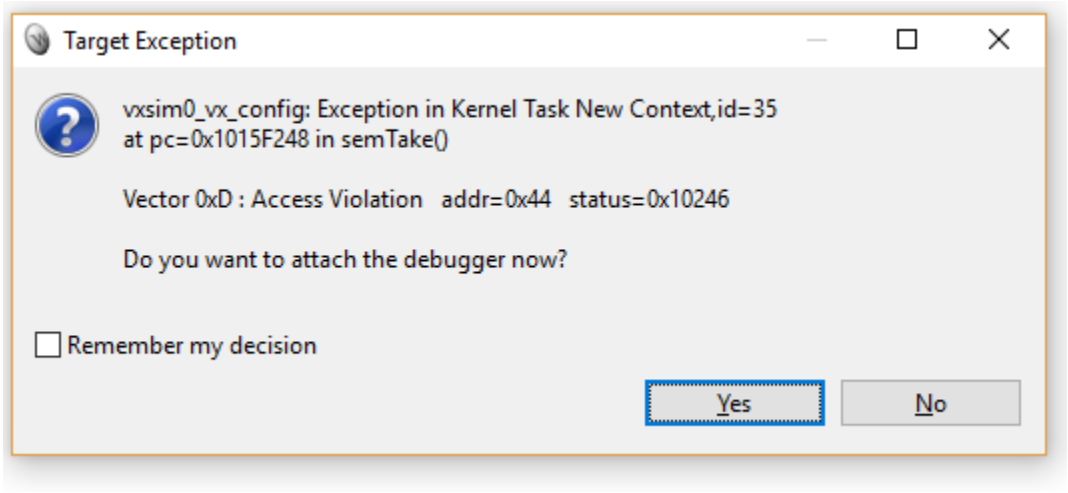
Answer:

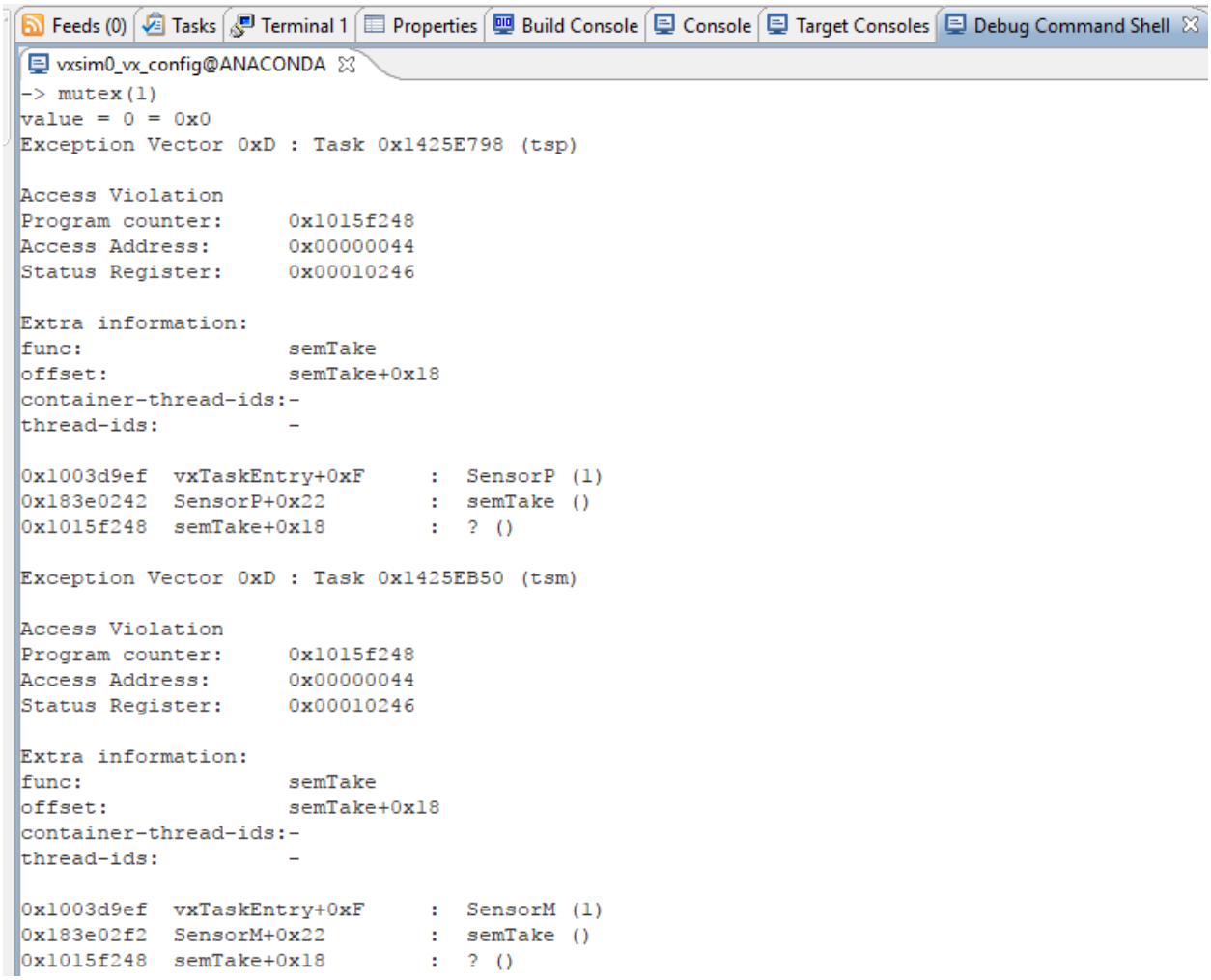
File -> VxWorks DKM Project, select a name for the project.
Right click on the newly created project and select New -> File and name the file mutex.c and copy the code.
Right click on the project and select Build.
Reboot the target with VxWorks image from the vx_config and reconnect the target server and start a host shell and run mutex(0)

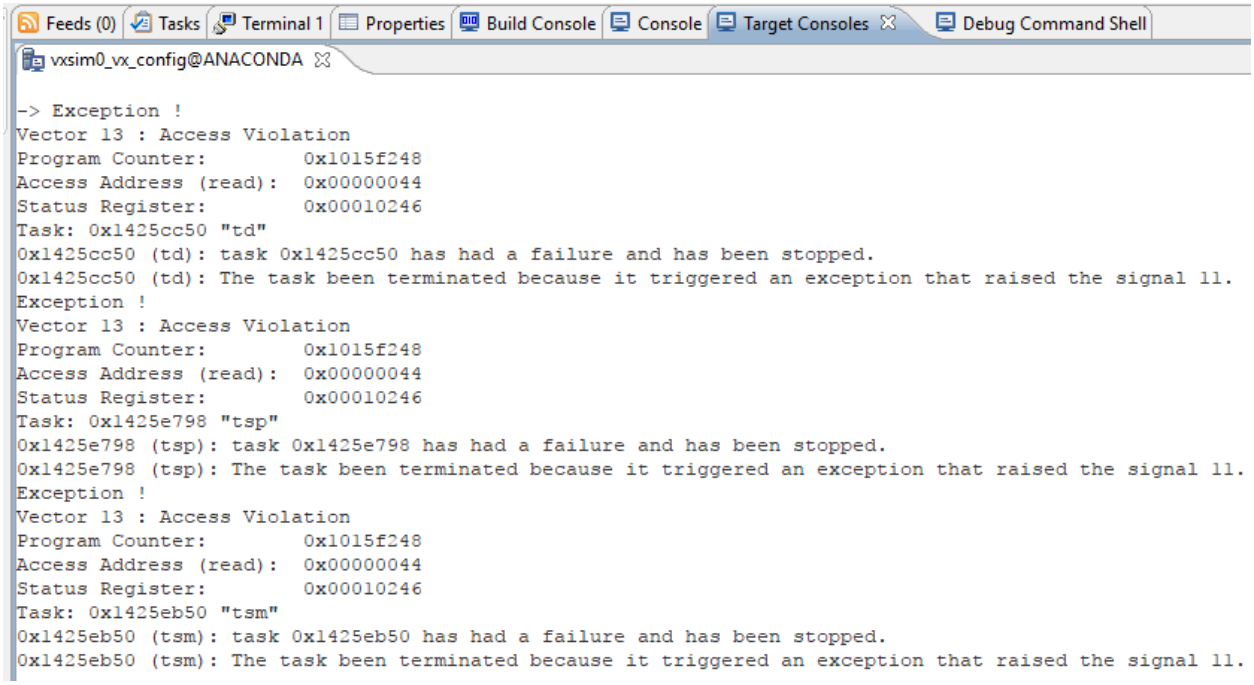




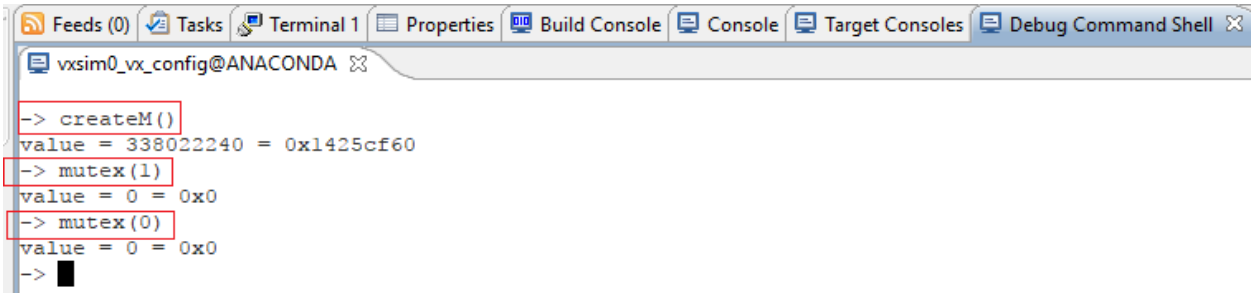
Running mutex(1) generates an error

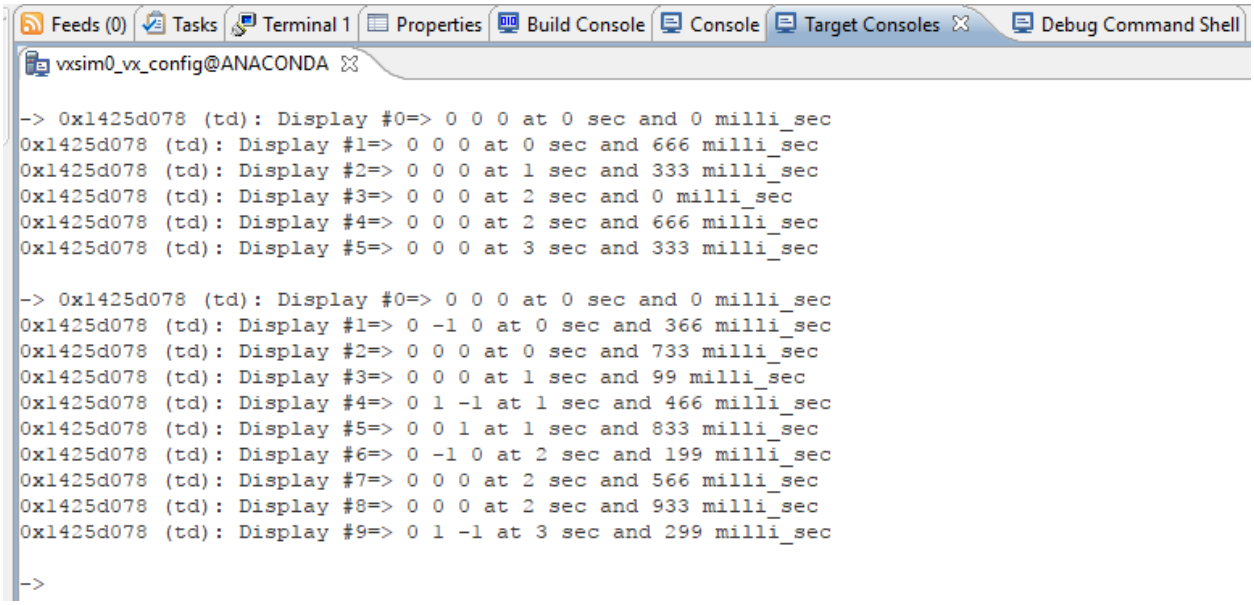






After creating mutex, there is no error.

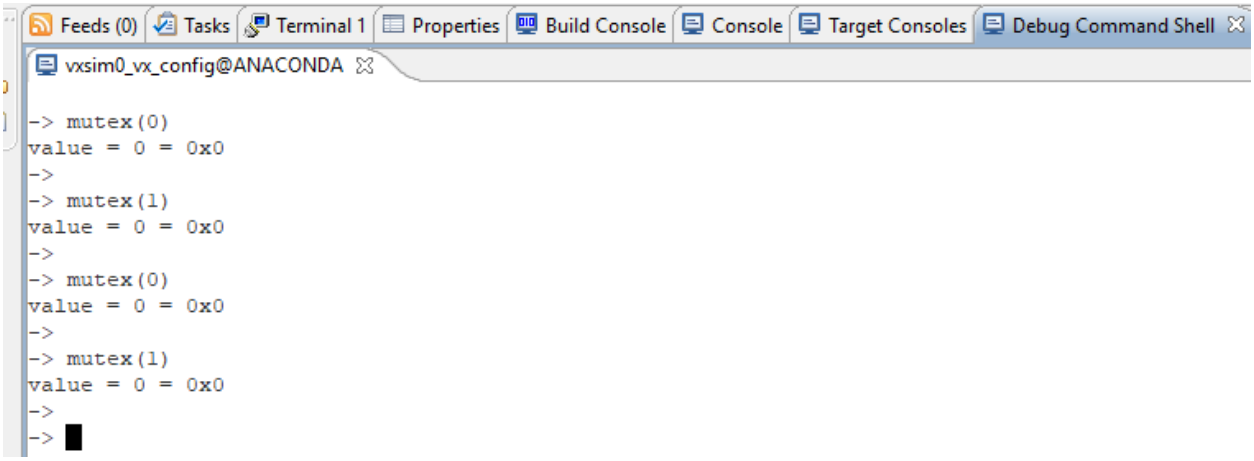




A2. Show, analyze and explain the results of running the function *mutex* a few times with both arguments. **How does it work? Why is creating the semaphore inside the function *mutex* incorrect?**

Answer:

Running the function mutex few times with parameter 0 and 1



Task "td" is created while running the function and function display is called with both parameters 0 and 1.

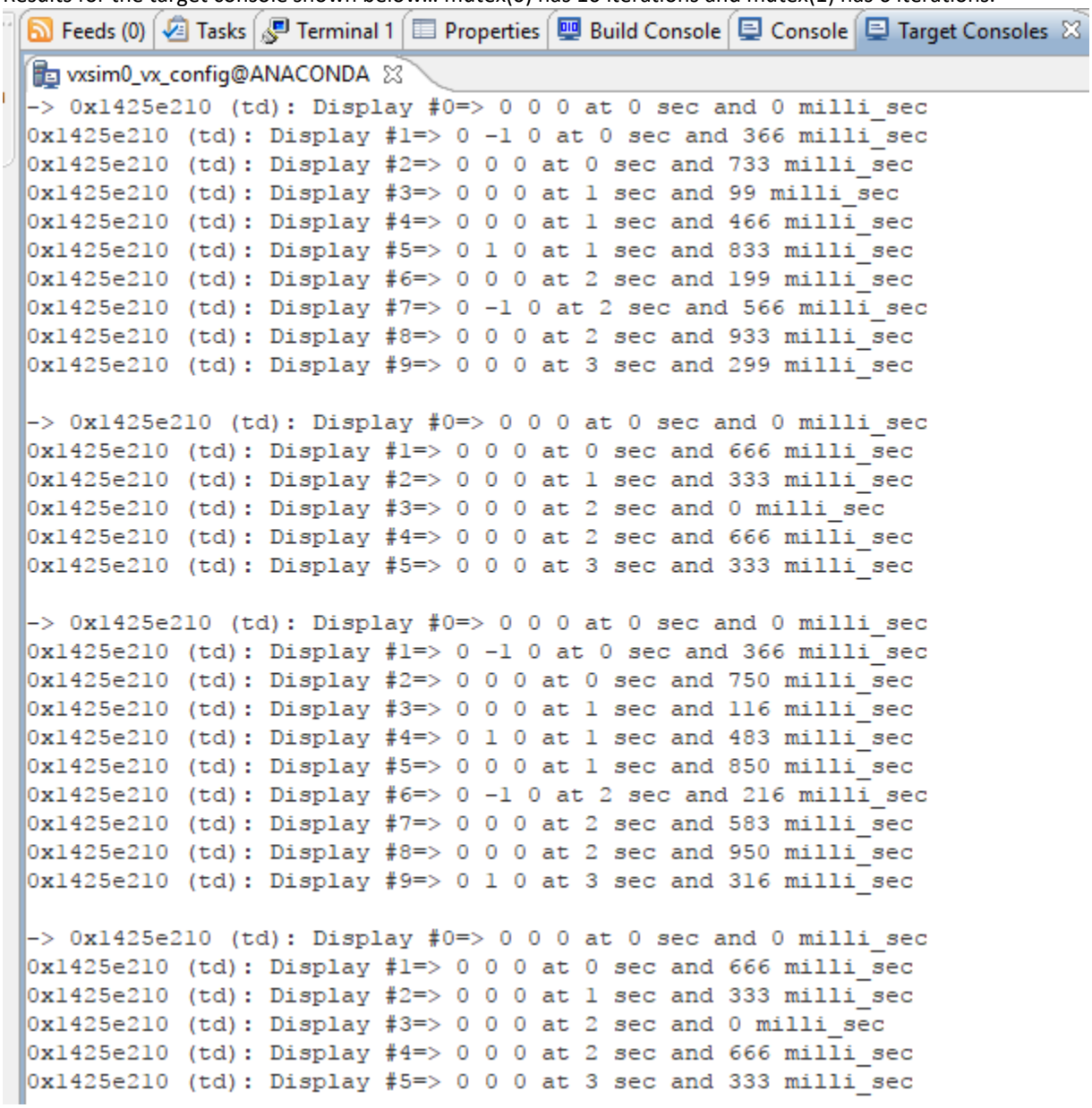
Values of x, y and z change during the argument 0 because the mutex is unlocked, whereas these values remain constant during argument 1 execution because the mutex is locked.

There is indefinite wait to get the semaphore, inside the function display if the argument is 1 and later releasing the semaphore, after the critical section when argument is 1.

20 ticks taskDelay is added in the function allowing other tasks to run at the same time which leads to a value change for x, y and z.

Creating a semaphore inside the function mutex will create a mutex every time the function is called. Function mutex has one integer parameter as well which defines weather it is locked or unlocked while creating a mutex requires 2 parameters.

Results for the target console shown below… mutex(0) has 10 iterations and mutex(1) has 6 iterations.



A3. Modify the source code such that the *Display* is spawned with a priority of 100 and re-run the above experiment - show the necessary code line modification. Observe, show, and explain the behavior of the tasks while executing mutex(1) before and after the modification. **Does a larger value signify higher/lower priority of a task in VxWorks, explain?** After completing this step, change the priority value back to 95.

Answer:

Changing priority to 100 in the code

*mutex.c semaphore.c

```
void mutex(int protect)
{
    /* clear the memory */
    data.x = 0; data.y = 0; data.z = 0;

    /* spawn three tasks */
    taskDisplay = taskSpawn("td", 100,0x100,2000,(FUNCPTR)Display,protect,0,0,0,0,0,0,0,0);
    taskSensorP = taskSpawn("tsp",95,0x100,2000,(FUNCPTR)SensorP,protect,0,0,0,0,0,0,0,0);
    taskSensorM = taskSpawn("tsm",95,0x100,2000,(FUNCPTR)SensorM,protect,0,0,0,0,0,0,0,0);
    taskDelay(220*TICK);    /* delay arbitrary # "ticks" before terminating Display task */
    taskDelete(taskDisplay);
}
```

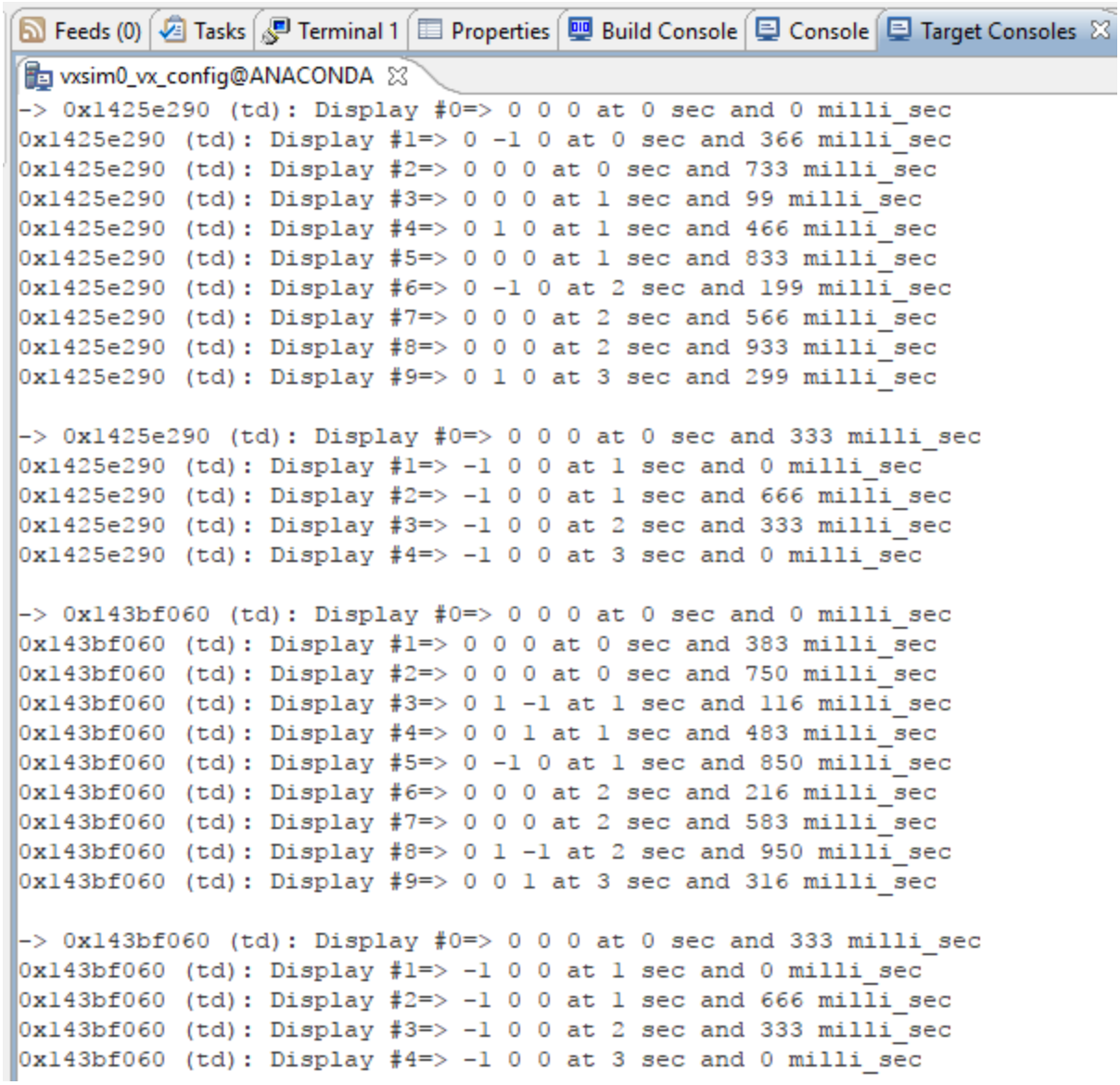
Running the commands from host shell

Feeds (0) Tasks Terminal 1 Properties Build Console Console Target Consoles Debug Command Shell

vxsim0_vx_config@ANACONDA (2)

```
-> createM()
value = 338027024 = 0x1425e210
-> mutex(0)
value = 0 = 0x0
-> mutex(1)
value = 0 = 0x0
-> mutex(0)
value = 0 = 0x0
-> mutex(1)
value = 0 = 0x0
->
```

Results on the target console in the image below:
Mutex(0) has the similar behavior whereas mutex(1) has changed from 6 iterations to 5 and values of x from second iteration is always -1.
Lower the value, higher the priority of the task is in VxWorks. Higher value represents a lower priority task.



A4. Use *show* command to examine the semaphore (*show semMtx* - we use the name of already created semaphore). **Show and explain the results of the *show* command on *semMutex*.**

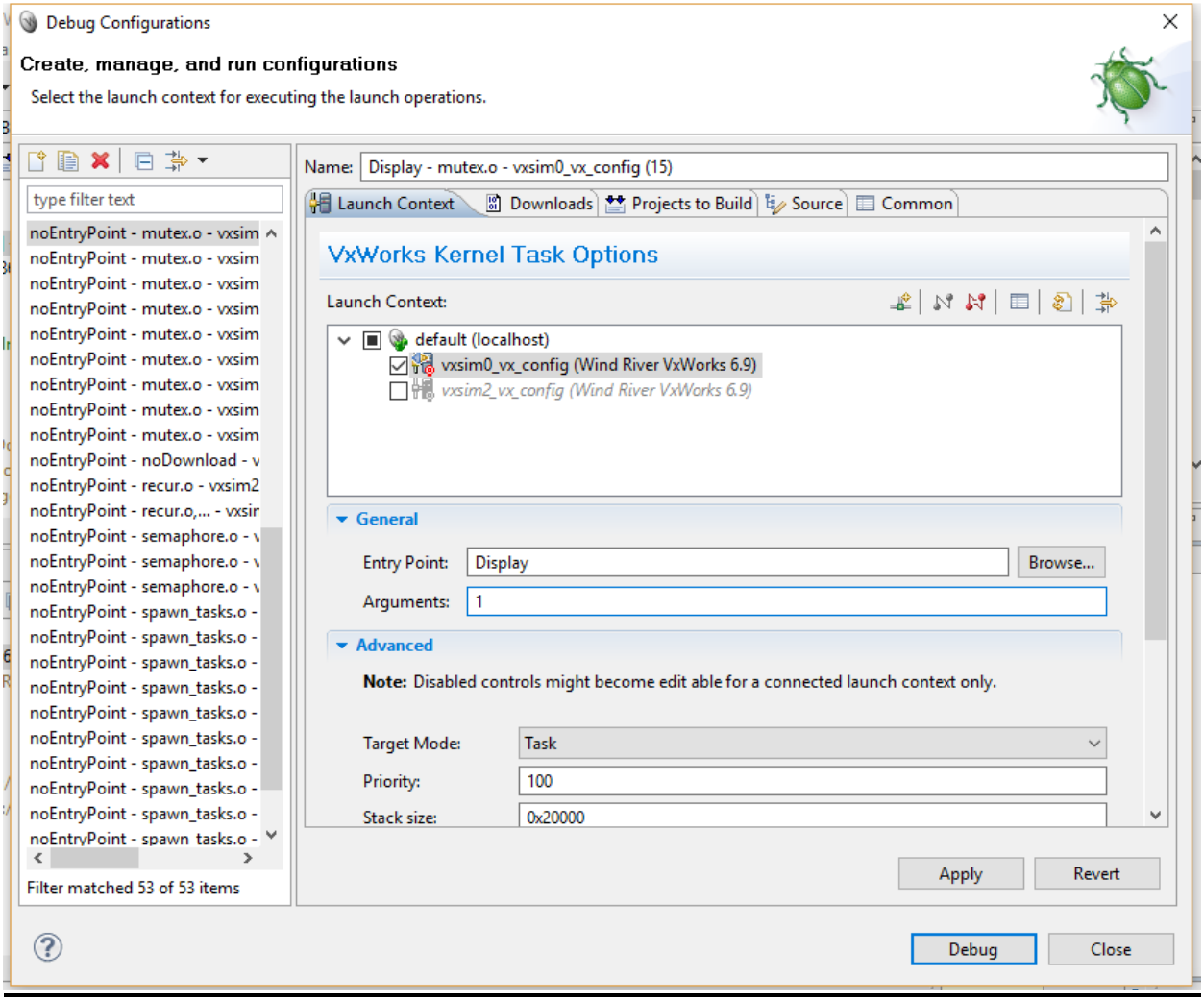
Answer:

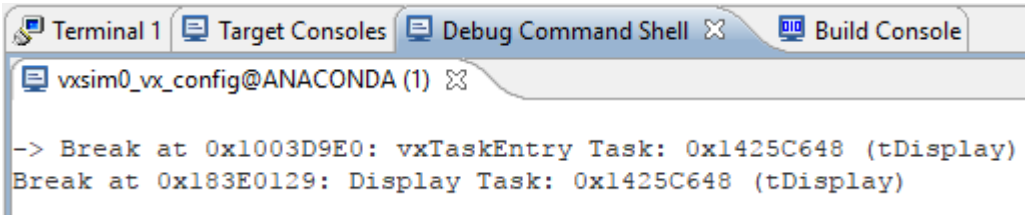
Semaphore information displayed in the image below explains semaphore attributes. The semaphore ID which is 0x143bfe50, type of semaphore which is Mutex, how the tasks are queued i.e. FIFO basis, pended tasks shows how many tasks are pending because of this mutex, Owner shows the owner task and options show what options are selected while creating the semaphore.

```
-> semMtx = semMCreate(4, 0)
0x14210690: value = 339476048 = 0x143bfe50
->
-> show(semMtx)

Semaphore Id          : 0x143bfe50
Semaphore Type        : MUTEX
Task Queueing         : FIFO
Pended Tasks          : 0
Owner                 : NONE
Options               : 0x4       SEM_Q_FIFO
                                  SEM_DELETE_SAFE

VxWorks Events
--------------
Registered Task       : NONE
Event(s) to Send      : N/A
Options               : N/A


value = 0 = 0x0
->
```

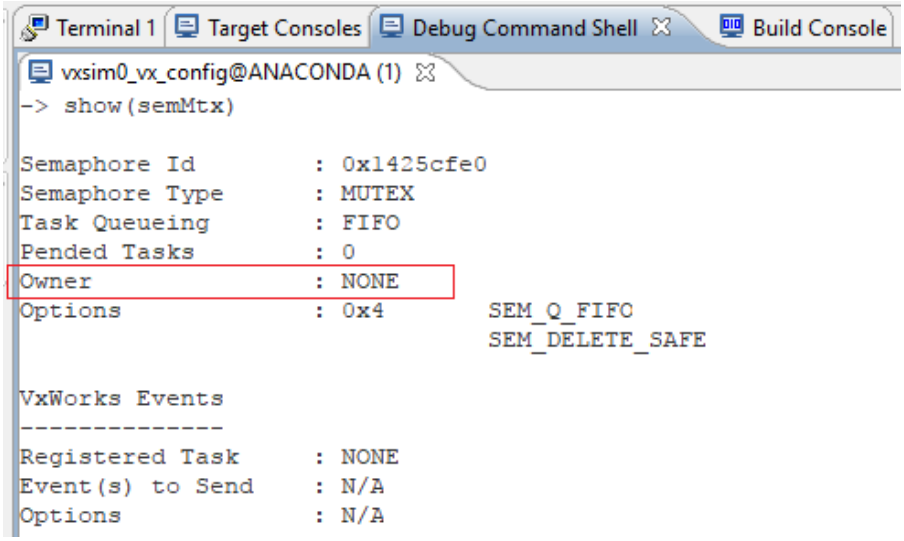
A5. Run the *Display* (with argument equal to 1) from the Debugger. Watch the *semMutex* while you single-step through the routine loop. Explain how *semMutex* changes while in the debugger. **Can you delete the task from the shell window (use *td*) while the *Display* is "inside" the while loop (the mutex is owned by the task)? Explain what you need to do to delete the task?**

Answer:

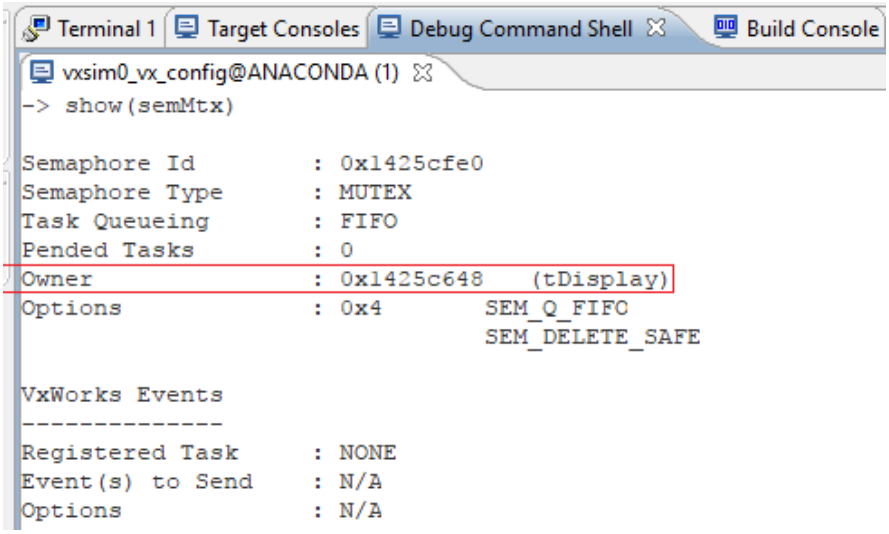




Executing step by step



As soon as the loop is executed, the semaphore is taken by tDisplay.

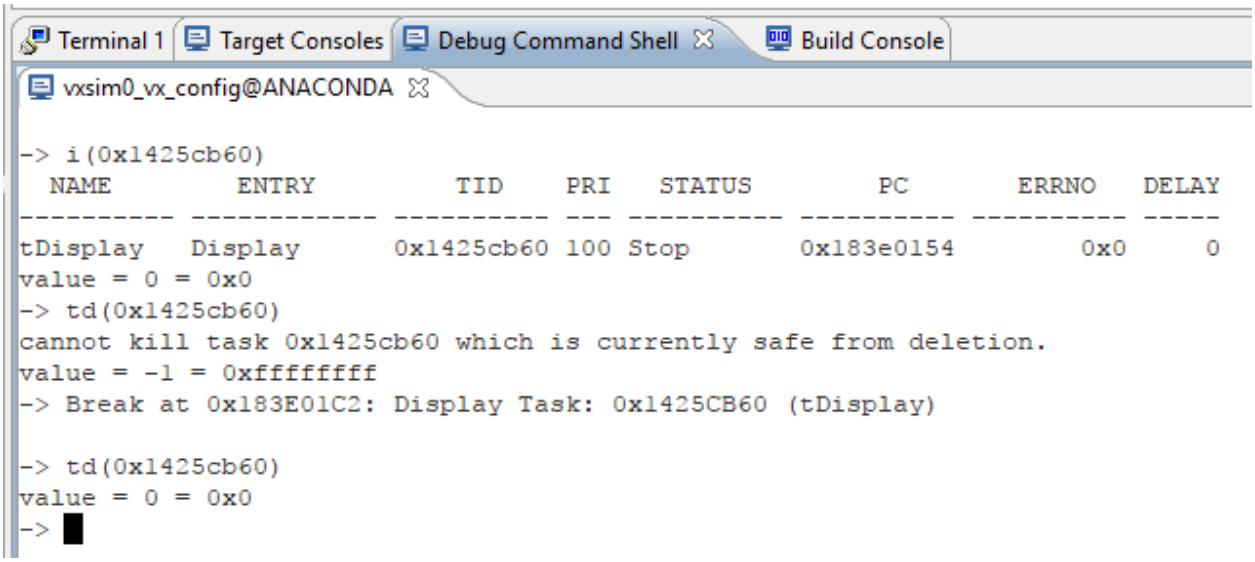


After the semGive, the semaphore is released.

```
-> Break at 0x183E01C2: Display Task: 0x1425C648 (tDisplay)

-> show(semMtx)

Semaphore Id         : 0x1425cfe0
Semaphore Type       : MUTEX
Task Queueing        : FIFO
Pended Tasks         : 0
Owner                : NONE
Options              : 0x4        SEM_Q_FIFO
                                  SEM_DELETE_SAFE

VxWorks Events
--------------
Registered Task      : NONE
Event(s) to Send     : N/A
Options              : N/A
```

The task cannot be deleted while the semaphore is taken, once its released, given back, the task can be deleted.

The image below shows the same…



Part B: Counting and Binary Semaphores

B1. Create **binary** FIFO empty semaphore from the shell command line *semBin = semBCreate(a,b).* Use proper numerical values for *a* and *b* rather than symbolic arguments: SEM_Q_FIFO is 0, SEM_Q_PRIORITY is 1, SEM_EMPTY is 0, SEM_FULL is 1. **What were the arguments to the *semBCreate* function? Check the status of the created semaphore object. How did you do it?**

Answer: Create a semaphore and check if it is as per requirement

```
semBin = semBCreate (0, 0)
New symbol "semBin" added to kernel symbol table.
semBin = 0x141ba694: value = 337579400 = 0x141f0d88
-> semShow(semBin)
Semaphore Id         : 0x141f0d88
Semaphore Type       : BINARY
Task Queueing        : FIFO
Pended Tasks         : 0
State                : EMPTY
Options              : 0x0       SEM_Q_FIFO

VxWorks Events
--------------
Registered Task      : NONE
Event(s) to Send     : N/A
Options              : N/A
```

B2. Spawn a task with *semTake* and 500 ticks wait from the shell line: *taskSpawn("x",95,0,1000, semTake, semBin, 500).* ***Observe the status of the task.***

a. Spawn the same task above multiple times. **Observe & explain the information you can gather about the created tasks.**

Answer: Creating a task t1 and spawning it multiple times only changes the task id.

```
t1 = taskSpawn("x",95,0,1000, semTake, semBin, 500)
t1 = 0x141ba674: value = 337579528 = 0x141f0e08
->
-> i(t1)
  NAME        ENTRY       TID      PRI   STATUS      PC        ERRNO    DELAY
---------- ------------ ---------- --- ---------- ---------- ---------- -----
x          semTake      0x141f0e08  95 Pend+T     0x1015dbab        0x0      0
value = 0 = 0x0
->
-> t1 = taskSpawn("x",95,0,1000, semTake, semBin, 500)
t1 = 0x141ba674: value = 339450024 = 0x143b98a8
->
-> i(t1)
  NAME        ENTRY       TID      PRI   STATUS      PC        ERRNO    DELAY
---------- ------------ ---------- --- ---------- ---------- ---------- -----
x          semTake      0x143b98a8  95 Pend+T     0x1015dbab        0x0      0
value = 0 = 0x0
->
-> t1 = taskSpawn("x",95,0,1000, semTake, semBin, 500)
t1 = 0x141ba674: value = 337579528 = 0x141f0e08
->
-> i(t1)
  NAME        ENTRY       TID      PRI   STATUS      PC        ERRNO    DELAY
---------- ------------ ---------- --- ---------- ---------- ---------- -----
x          semTake      0x141f0e08  95 Pend+T     0x1015dbab        0x0      0
value = 0 = 0x0
->
```

b. Execute a few times *semGive(semBin)* from the shell command line while watching the semaphore status. **What is the result?**

Answer: Running semGive(semBin) changes the semaphore state from "Empty" to "Full" and running it again and again it does not change anything, it is either empty or full.

```
-> semGive(semBin)
value = 0 = 0x0
-> show(semBin)

Semaphore Id         : 0x141f0d88
Semaphore Type       : BINARY
Task Queueing        : FIFO
Pended Tasks         : 0
State                : FULL
Options              : 0x0        SEM_Q_FIFO

VxWorks Events
--------------
Registered Task      : NONE
Event(s) to Send     : N/A
Options              : N/A
```

c. Change the priority of the spawned task to observe the effect when working with a PRIORITY semaphore. **Explain and show how you accomplished this. What is the difference between FIFO and PRIORITY semaphores?**

Answer: First we need to create a PRIORITY semaphore semBin2.

```
semBin2 = semBCreate (1, 0)
New symbol "semBin2" added to kernel symbol table.
semBin2 = 0x141ba628: value = 337579528 = 0x141f0e08
-> show(semBin2)

Semaphore Id         : 0x141f0e08
Semaphore Type       : BINARY
Task Queueing        : PRIORITY
Pended Tasks         : 0
State                : EMPTY
Options              : 0x1       SEM_Q_PRIORITY

VxWorks Events
--------------
Registered Task      : NONE
Event(s) to Send     : N/A
Options              : N/A
```

Create 2 tasks x and y with priorities 85 and 90 and see how they show up among tasks.

```
-> t10 = taskSpawn("x",85,0,1000, semTake, semBin, 500)
t10 = 0x141ba618: value = 337580904 = 0x141f1368
-> t20 = taskSpawn("y",90,0,1000, semTake, semBin, 500)
t20 = 0x141ba5fc: value = 337581688 = 0x141f1678
-> i
  NAME        ENTRY          TID     PRI   STATUS        PC        ERRNO    DELAY
---------- ------------ ---------- --- ---------- ---------- ---------- -----
tJobTask   jobTask      0x14217a00   0 Pend       0x1015dbab        0x0     0
tExcTask   excTask      0x101e73e0   0 Pend       0x1015dbab        0x0     0
tLogTask   logTask      0x1421b740   0 Pend       0x1015b8eb        0x0     0
tShell0    shellTask    0x14377e40   1 Pend       0x1015dbab        0x0     0
tWdbTask   wdbTask      0x184a8fd0   3 Ready      0x1015dbab    0xb30008    0
ipcom_tick ipcom_tickd  0x1437d970  20 Pend       0x1015dbab        0x0     0
tVxdbgTask vxdbgEventTa 0x184a86d0  25 Pend       0x1015dbab        0x0     0
tAioIoTask aioIoTask    0x1421fbe0  50 Pend       0x1015e446        0x0     0
tAioIoTask aioIoTask    0x14238750  50 Pend       0x1015e446        0x0     0
tNet0      ipcomNetTask 0x1423cdf0  50 Pend       0x1015dbab    0x3d0001    0
ipcom_sysl ipcom_syslog 0x1423f4a0  50 Pend       0x1015e446        0x0     0
tNetConf   ipnet_config 0x1429f900  50 Pend       0x1015dbab        0x0     0
tAioWait   aioWaitTask  0x1421f728  51 Pend       0x1015dbab        0x0     0
x          semTake      0x141f1368  85 Pend+T     0x1015dbab        0x0     0
y          semTake      0x141f1678  90 Pend+T     0x1015dbab        0x0     0
tsp        SensorP      0x141f4378  95 Stop       0x101654a7        0x0     0
tsm        SensorM      0x141f7698  95 Stop       0x101654a7        0x0     0
tsp        SensorP      0x143b3020  95 Stop       0x101654a7        0x0     0
tsm        SensorM      0x143b6368  95 Stop       0x101654a7        0x0     0
value = 0 = 0x0
```

Change the priorities of x and y to 40 and 30

```
-> t10 = taskSpawn("x",40,0,1000, semTake, semBin, 500)
t10 = 0x141ba618: value = 337580904 = 0x141f1368
-> t20 = taskSpawn("y",30,0,1000, semTake, semBin, 500)
t20 = 0x141ba5fc: value = 337581688 = 0x141f1678
-> i
  NAME        ENTRY          TID     PRI   STATUS        PC        ERRNO    DELAY
---------- ------------ ---------- --- ---------- ---------- ---------- -----
tJobTask   jobTask      0x14217a00   0 Pend       0x1015dbab        0x0      0
tExcTask   excTask      0x101e73e0   0 Pend       0x1015dbab        0x0      0
tLogTask   logTask      0x1421b740   0 Pend       0x1015b8eb        0x0      0
tShell0    shellTask    0x14377e40   1 Pend       0x1015dbab        0x0      0
tWdbTask   wdbTask      0x184a8fd0   3 Ready      0x1015dbab    0xb30008     0
ipcom_tick ipcom_tickd  0x1437d970  20 Pend       0x1015dbab        0x0      0
tVxdbgTask vxdbgEventTa 0x184a86d0  25 Pend       0x1015dbab        0x0      0
y          semTake      0x141f1678  30 Pend+T     0x1015dbab        0x0      0
x          semTake      0x141f1368  40 Pend+T     0x1015dbab        0x0      0
tAioIoTask aioIoTask    0x1421fbe0  50 Pend       0x1015e446        0x0      0
tAioIoTask aioIoTask    0x14238750  50 Pend       0x1015e446        0x0      0
tNet0      ipcomNetTask 0x1423cdf0  50 Pend       0x1015dbab    0x3d0001     0
ipcom_sysl ipcom_syslog 0x1423f4a0  50 Pend       0x1015e446        0x0      0
tNetConf   ipnet_config 0x1429f900  50 Pend       0x1015dbab        0x0      0
tAioWait   aioWaitTask  0x1421f728  51 Pend       0x1015dbab        0x0      0
tsp        SensorP      0x141f4378  95 Stop       0x101654a7        0x0      0
tsm        SensorM      0x141f7698  95 Stop       0x101654a7        0x0      0
tsp        SensorP      0x143b3020  95 Stop       0x101654a7        0x0      0
tsm        SensorM      0x143b6368  95 Stop       0x101654a7        0x0      0
value = 0 = 0x0
```

FIFO semaphores work on first in first out basis, task making the first request is served first and PRIORITY semaphores entertains tasks with higher priorities first.

B3. Create new semaphore with different characteristics (empty/full, priority/FIFO) and the same identifier (*semBin*). **Are there in fact two semaphores or only one? Prove your answer showing shell commands and the system responses. Explain.**

Answer: Every time we change characteristics and try to create a new semaphore with the same name, it changes the attributes of existing semaphore, that includes (id, task queueing, state and options).

```
show(semBin)

Semaphore Id          : 0x141f0e88
Semaphore Type        : BINARY
Task Queueing         : PRIORITY
Pended Tasks          : 0
State                 : EMPTY
Options               : 0x1       SEM_Q_PRIORITY

VxWorks Events
--------------
Registered Task       : NONE
Event(s) to Send      : N/A
Options               : N/A


value = 0 = 0x0
-> semBin = semBCreate(0, 0)
semBin = 0x141ba694: value = 337579784 = 0x141f0f08
-> show(semBin)

Semaphore Id          : 0x141f0f08
Semaphore Type        : BINARY
Task Queueing         : FIFO
Pended Tasks          : 0
State                 : EMPTY
Options               : 0x0       SEM_Q_FIFO

VxWorks Events
--------------
Registered Task       : NONE
Event(s) to Send      : N/A
Options               : N/A

value = 0 = 0x0
-> semBin = semBCreate(1, 1)
semBin = 0x141ba694: value = 337579912 = 0x141f0f88
-> show(semBin)

Semaphore Id          : 0x141f0f88
Semaphore Type        : BINARY
Task Queueing         : PRIORITY
Pended Tasks          : 0
State                 : FULL
Options               : 0x1       SEM_Q_PRIORITY

VxWorks Events
--------------
Registered Task       : NONE
Event(s) to Send      : N/A
Options               : N/A


value = 0 = 0x0
```

B4. Experiment with a **counting** semaphore similar to the points above. **What are the commands you must execute from the shell? Show and explain your results**.

Answer: Working with COUNTING semaphores also does the same thing…

```
-> semCn = semCCreate(0, 1)
New symbol "semCn" added to kernel symbol table.
semCn = 0x141ba5e0: value = 337580040 = 0x141f1008
-> show(semCn)

Semaphore Id         : 0x141f1008
Semaphore Type       : COUNTING
Task Queueing        : FIFO
Pended Tasks         : 0
Count                : 1
Options              : 0x0       SEM_Q_FIFO

VxWorks Events
--------------
Registered Task      : NONE
Event(s) to Send     : N/A
Options              : N/A


value = 0 = 0x0
-> semCn = semCCreate(1, 1)
semCn = 0x141ba5e0: value = 337901488 = 0x1423f7b0
-> show(semCn)

Semaphore Id         : 0x1423f7b0
Semaphore Type       : COUNTING
Task Queueing        : PRIORITY
Pended Tasks         : 0
Count                : 1
Options              : 0x1       SEM_Q_PRIORITY

VxWorks Events
--------------
Registered Task      : NONE
Event(s) to Send     : N/A
Options              : N/A


value = 0 = 0x0
-> semCn = semCCreate(0, 0)
semCn = 0x141ba5e0: value = 337901616 = 0x1423f830
-> show(semCn)

Semaphore Id         : 0x1423f830
Semaphore Type       : COUNTING
Task Queueing        : FIFO
Pended Tasks         : 0
Count                : 0
Options              : 0x0       SEM_Q_FIFO

VxWorks Events
--------------
Registered Task      : NONE
Event(s) to Send     : N/A
Options              : N/A
```

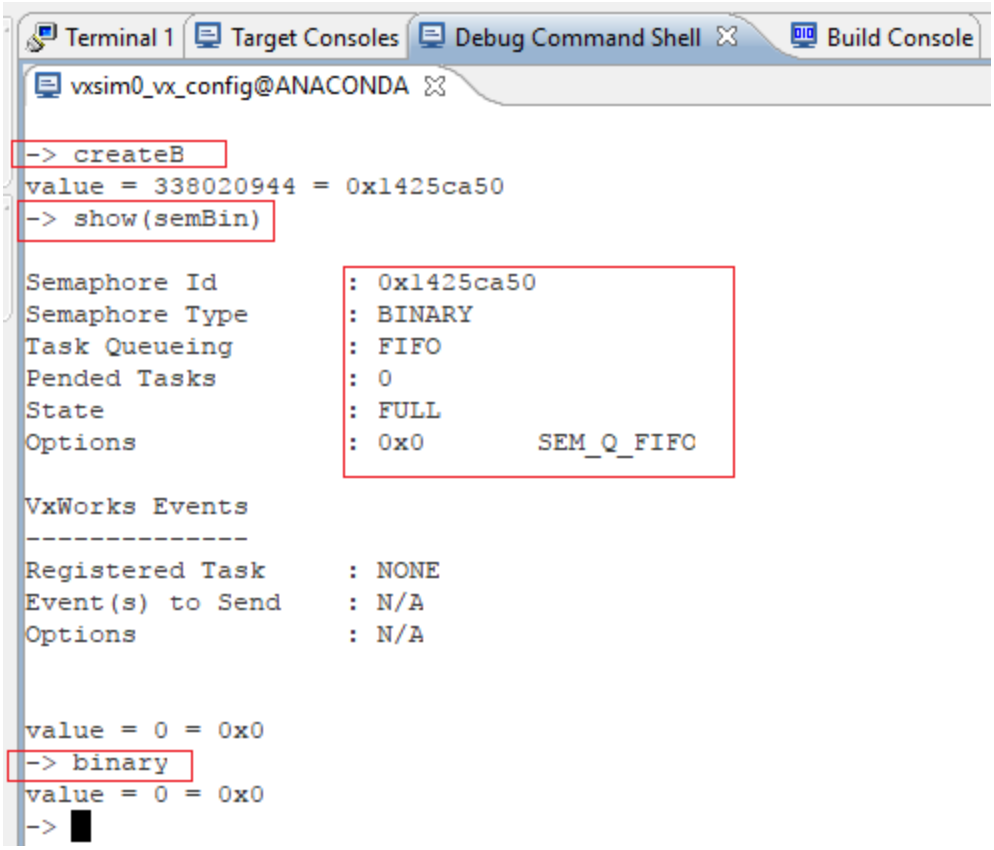
B5. Write a new program ***semaphore.c*** to have only two tasks: **Sensor** (increasing the data by one - an equivalent to the **SensorP** from the demo program) and **Display** (displaying the data - but with time stamp expressed as VxWorks time ***tick***, rather than seconds and nanoseconds)**.** The two tasks shall synchronize their action, i.e. the Display task must wait for the Sensor to update *x,y,z* and only then log the message - rather than loging the data periodically as in the demo. As the result of this modification the message is displayed after each update and thus *x,y,z* values displayed will be always 1, 1, 1. {HINT: we need to use binary semaphore *semBin* for synchronization - rather than mutex semaphore for mutual exclusion. Change the name of the program main function (to e.g. *binary*); create and properly initialize the semaphore. The *Display* should take the semaphore before logging message, while the *Sensor* should give the semaphore after completion of updating.} **Show the created source code with comments and explain the results of executing your program.**

Answer: The modified code:

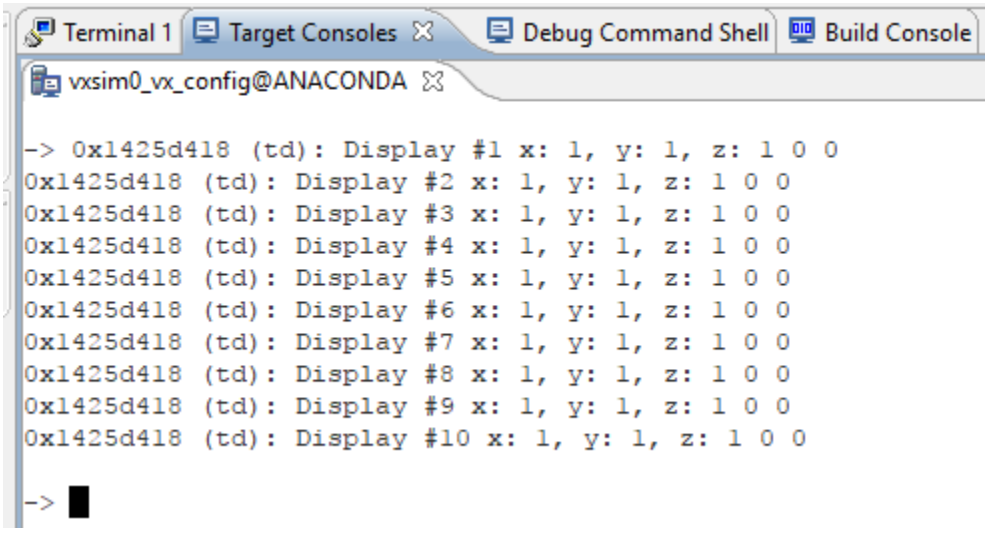
```
#include <vxWorks.h>      /* Always include this as the first thing in every program */
#include <time.h>         /* we use clock_gettime */
#include <taskLib.h>      /* we use tasks */
#include <sysLib.h>       /* we use sysClk... */
#include <semLib.h>       /* we use semaphores */
#include <logLib.h>       /* we use logMsg rather than printf */

/* define useful constants for timing */
#define NANOS_IN_SEC 1000000000
#define NANOS_PER_MILLI 1000000
#define TICK sysClkRateGet()/60

/* globals */
#define ITER 22   /* arbitrary number of iterations – can be changed */

/* function prototypes */
void Sensor();
void Display();

SEM_ID semBin;  /* a semaphore supporting mutual exclusion */
/* only the task "taking" semaphore can "give" it */
int taskSensor,taskDisplay;          /* task references */
/* our "shared memory" area: three data to be kept coherent */
/* i.e. they need to show the same values when printing */
struct mem{
        int x; int y; int z;
} data;

/* a routine createB to create a binary semaphore - can be also done from the shell line  */
/* queue tasks on FIFO basis and deletion safety       */
void createB(){
        semBin = semBCreate(0, 1);
}

/* the main program named mutex creating semaphore and spawning three working tasks */
void binary(){
        /* clear the memory */
        data.x = 0; data.y = 0; data.z = 0;
```

```
	/* spawn three tasks */
	taskSensor = taskSpawn("ts",95,0x100,2000,(FUNCPTR)Sensor,0,0,0,0,0,0,0,0,0,0);
	taskDisplay = taskSpawn("td", 95,0x100,2000,(FUNCPTR)Display,0,0,0,0,0,0,0,0,0,0);
	taskDelay(220*TICK);   /* delay arbitrary # "ticks" before terminating Display task */
	taskDelete(taskDisplay);
}

/* the "sensor" routine */
void Sensor(){
	int i;
	for (i=0; i < ITER; i++){
		/* "critical section" - wait indefinitely for semaphore */
		semTake(semBin, WAIT_FOREVER);
		/* beginning of the the "critical section" with simulated operation delay */
		data.x++; data.y++; data.z++;
		/* end of the" critical section" - give up semaphore, if protect = 1 */
		semGive(semBin);
		taskDelay(22*TICK); /* delay arbitrary # ticks - periodic task */
	}
}

/* the "Display" routine */
void Display(int protect){
	int i=1;
	/* loop forever (until the task get killed) */
	while(1){
		/* "critical section" - wait indefinitely for semaphore, if protect = 1 */
		semTake(semBin,WAIT_FOREVER);
		/* beginning of the "critical section" for printing */
		/* we use VxWorks logMsg rather than printf - as printf may block */
		logMsg("Display #%d x: %d, y: %d, z: %d %d %d\n", i++, data.x, data.y, data.z, 0, 0);
		semGive(semBin);
		/* clear the memory for the next printing */
		data.x = 0; data.y = 0; data.z = 0;
		taskDelay(22*TICK); /* delay arbitrary # ticks - periodic task */
	}
}
```

The host shell commands after compiling and downloading the new code…



Results on the target console…



B6. **What default scheduling algorithm is used by VxWorks? What line of code must be added/changed to change the scheduling algorithm? Experiment with the demo program after these changes. Show & explain your results.**

Answer:

VxWorks uses preemptive priority-based scheduling as default. A task of a specified priority can only preempted by a higher priority task, another task of the same priority will only run when this task is blocked or willingly goes to sleep. This means that if a single task is never blocked, it never gives a change to another equal-priority task to run.

Round-robin scheduling solves this problem. In VxWorks, this is the default scheduling algorithm. Round-robin scheduling Like preemptive priority-based scheduling but it also attempts the share the CPU fairly among all tasks of the same priority using the so called time-slicing technique. In time-slicing each task can run freely until its preempted by a higher priority task or its time-slice has ended. In the latter case, another equal priority task is scheduled to run. Thus, the equal-priority task rotate, each executing for an equal interval of time.

In VxWorks one can activate round-robin scheduling with the function kernelTimeSlice() with the specified timeslice.

Section 4: Observations, Comments, and Lessons Learned

I have learned more about WindRiver functionality. During this exercise, I have discovered new challenges while working on this lab. This time I feel more confident and less helpless like last time. Still I believe more practice and exercise is required in order to gain more experience and knowledge.

While working on this lab, I was able to understand about semaphores and their characteristics., running multiple tasks, setting task priorities, working on shared data and the issues involved. How and when the use of semaphores is useful.

I have used the following websites for information related to the lab.
http://www.vxdev.com/
http://www.cs.ru.nl/lab/vxworks/vxworksOS.html