CS 4390/5372: Specifications and Design of Real-Time Systems
Lab3: Synchronization and Semaphores

Introduction: Semaphores permit multitasking applications to coordinate their activities. The most obvious way for tasks to communicate is via various shared data structures. Because all tasks in VxWorks exist in a single linear address space, shared data structures between tasks is trivial. Global variables, linear buffers, ring buffers, link lists, and pointers can be referenced directly by code running in different context. However, while shared address space simplifies the exchange of data, interlocking access to memory is critical to avoid contention. Many methods exist for obtaining exclusive access to resources, and one of them is implemented using semaphores.

Primary objectives of the experiments:

- To understand how multiple tasks get access to shared data.
- To understand how tasks can synchronize their actions.
- To demonstrate the use of VxWorks semaphores for synchronization and access control from the program and the WindSh command line
- To examine tasks and programming constructs from WindSh.
- To introduce Round-Robin vs. Priority-Based scheduling.

Description:
VxWorks semaphores are highly optimized and provide the fastest intertask communication mechanisms in VxWorks. Semaphores are the primary means for addressing the requirements of both mutual exclusion and task synchronization. There are three types of Wind semaphores, optimized to address different classes of problems:

1. **binary -** the fastest, most general purpose semaphore (optimized for synchronization - can be given and taken by any task)
2. **mutual exclusion -** a special binary semaphore (optimized for mutual exclusion, addressing problems of priority inheritance, deletion safety and recursion - can be only given by task that took it)
3. **counting -** keeping track of the number of times the semaphore is given (optimized for guarding multiple instances of a resource)

Semaphore Operations and Syntax:
VxWorks semaphores provide a single uniform interface for semaphore control. Only the creation routines are specific to the semaphore type:

- *semBCreate*(int options, SEM_B_STATE initialState): Allocate and initialize a binary semaphore.
- *semMCreate*(int options): Allocate and initialize a mutual exclusion semaphore.
- *semCCreate*(int options, int initialCount): Allocate and initialize a counting semaphore.
- *semDelete*(SEM_ID semId): Terminate and free a semaphore.
- *semTake*(SEM_ID semId, int timeout): Take a semaphore.
- *semGive*(SEM_ID semId): Give a semaphore.
- *semFlush*(SEM_ID semId): Unblock all tasks waiting for a semaphore.

Please refer to the VxWorks Reference Manual for valid arguments in the above routines.

Example:

The example program *mutex.c* (attached) uses three tasks implementing functions of two sensors (**SensorP, SensorM)** and a **Display**. The sensor tasks update three data items (x,y,z) to be used by the display task. Obviously, we would like to display the coherent data (reflecting the same update - the values should be identical). To achieve this goal, the "critical section" of the code must be enclosed between the semaphore call (we use a mutual exclusion semaphore here). This program can be executed with a *protect* argument allowing to select the option to either not use (*protect = 0*) or to use (*protect = 1*) the semaphore for data protection. We use *taskDelay* to simulate timing characteristics of the program - different for each sensor. We also introduce *logMsg* for keeping track of the activities (the standard IO *printf* function is not advised to be used in Real-Time programs as it can block).

Requirements:

We shall use Windriver's Workbench and the hardware target. However, you may also experiment with using simulated target environment using vxsim. ***Use one-side paper and single spacing for the report.***

1. Complete Part A and B experiments. Record all pertinent commands that you have executed and their results. Attempt to understand and explain the significance of each step.
2. Prepare lab report using the prescribed format. Include what you performed at each phase of the lab and what results you received. In the report identify by each step of the experiment letter and number (e.g. A7, B3) and respond to all underlined questions. Include descriptions of all procedures/activities, results, and observations, the shell commands and their outcomes.
3. Include in your report only modified or new lines of the source code (highlight/comment the modified sections).

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*.

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?**

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.

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*.**

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?**

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?**

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.**

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

  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 is between FIFO and PRIORITY semaphores?**

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.**

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.**

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.**

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.**

Appendix: *mutex.c*

```
#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

/* function prototypes */
void SensorP(int);
void SensorM(int);
void Display(int);
```

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

SEM_ID semMtx; /* a semaphore supporting mutual exclusion */
               /* only the task "taking" semaphore can "give" it */

int taskSensorP, taskSensorM,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 createM to create "mutex" semaphore - can be also done from the shell line  */
/* queue tasks on FIFO basis and deletion safety                    */
void createM()
{
semMtx = semMCreate(SEM_Q_FIFO | SEM_DELETE_SAFE);
}
```

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

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

```
/* the "Display" routine printing the contents of shared memory */
/* every ten time "ticks"; */
/* the protect argument controls whether or not the semaphore */
/* will be used (1 - used, 0 - not used) */
void Display(int protect)
{
  /* preparation for time computation */
  struct timespec tpstart, tpend;
  int count=0, isec, insec, milli_sec;

  clock_gettime(CLOCK_REALTIME, &tpstart);

  /* loop forever (until the task get killed) */
  while(1)
      {
      /* "critical section" - wait indefinitely for semaphore, if protect = 1 */
      if(protect)  semTake(semMtx,WAIT_FOREVER);

      /* beginning of the "critical section" for printing */
      /* necessary computations to display current time */
      clock_gettime(CLOCK_REALTIME, &tpend);
      isec = tpend.tv_sec - tpstart.tv_sec;
      insec = tpend.tv_nsec - tpstart.tv_nsec;
      if (insec < 0) { insec = insec + NANOS_IN_SEC; isec--; };
           milli_sec=insec/NANOS_PER_MILLI;

      /* we use VxWorks logMsg rather than printf - as printf may block */
           logMsg("Display #%d=> %d %d %d at %d sec and %d milli_sec
                                  \n",count++,data.x,data.y,data.z,isec,milli_sec);

      /* end of the" critical section" give up semaphore, if protect = 1 */
      if(protect) semGive(semMtx);
      /* 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 "sensor Plus" routine increasing the shared memory ITER times; */
/* the protect argument controls whether or not the semaphore will */
/* be used (1 - used, 0 - not used) */
void SensorP(int protect)
{
int i;
for (i=0; i < ITER; i++)
      {
    /* "critical section" - wait indefinitely for semaphore, if protect = 1 */
    if(protect) semTake(semMtx,WAIT_FOREVER);
    /* beginning of the the "critical section" with simulated operation delay */
      data.x++; taskDelay(7*TICK);
      data.y++; taskDelay(1*TICK);
      data.z++; taskDelay(2*TICK);
    /* end of the" critical section" - give up semaphore, if protect = 1 */
    if(protect) semGive(semMtx);
      }
}
```

```
/* the "sensor Minus" routine decreasing the shared memory ITER times; */
/* the protect argument controls whether or not the semaphore will */
/* be used (1 - used, 0 - not used) */
void SensorM(int protect)
{int i;
  for (i=0; i < ITER; i++)
        {
      /* "critical section" - wait indefinitely for semaphore, if protect = 1 */
      if(protect) semTake(semMtx,WAIT_FOREVER);
      /* beginning of the the "critical section" with simulated operation delay */
        data.x--; taskDelay(1*TICK);
        data.y--; taskDelay(6*TICK);  data.z-
        -; taskDelay(3*TICK);
      /*  end of the" critical section" - give up semaphore, if protect = 1 */
      if(protect) semGive(semMtx);
        }
}
```