Experiment #3
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 crucial to avoid contention. Many methods exist for obtaining exclusive access to resources, and one of them is semaphores.

Objectives

The following are the primary objectives of this experiment:

- To demonstrate the use of VxWorks semaphores.

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.
The are three types of Wind semaphores, optimized to address different classes of problems:
binary

- The fastest, most general purpose semaphore. Optimized for synchronization and can also be used for mutual exclusion.

mutual exclusion

- A special binary semaphore optimized for problems inherent in mutual exclusion: priority inheritance, deletion safety and recursion.

counting

- Like the binary semaphore, but keeps track of the number of times the semaphore is given. Optimized for guarding multiple instances of a resource.

1. Semaphore Control

Wind 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 intialize a binary semaphore.*
- *semMCreate(int options): Allocate and intialize a mutual exclusion semaphore.*
- *semCCreate(int options, int initialCount): Allocate and intialize 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): Ublock all tasks waiting for a semaphore.*

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

2. Example: Binary Semaphore

A binary semaphore can be viewed as a flag that is available or unavailable. When a task takes a binary semaphore, using *semTake()*, the outcome depends on whether the semaphore is available or unavailable at the time of the call. If the semaphore is

available, then the semaphore becomes unavailable and then the task continues executing immediately. If the semaphore is unavailable, the task is put on a queue of blocked tasks and enters a state of pending on the availability of the semaphore.

When a task gives a binary semaphore, using *semGive()*, the outcome depends on whether the semaphore is available or unavailable at the time of the call.If the semaphore is already available, giving the semaphore has no effect at all. If the semaphore is unavailable and no task is waiting to take it, then the semaphore becomes available. If the semaphore is unavailable and one or more tasks are pending on its availablity, then the first task in the queue of pending tasks is unblocked, and the semaphore is left unavailable.

In the example below, two tasks(taskOne and taskTwo), are competing to update the value of a global variable, called "global." The objective of the program is to toggle the value of the global variable(1s and 0s). taskOne changes the value of "global" to 1 and taskTwo changes the value back to 0.

Without the use of the semaphore, the value of "global" would be random and the value of "global" would be corrupted.

```
-------------------------------------------------------------------------------
/* includes */
#include "vxWorks.h"
#include "taskLib.h"
#include "semLib.h"
#include "stdio.h"

/* function prototypes */
void taskOne(void);
void taskTwo(void);

/* globals */
#define ITER 10
SEM_ID semBinary;
int global = 0;

void binary(void)
{
int taskIdOne, taskIdTwo;

/* create semaphore with semaphore available and queue tasks on FIFO basis */
semBinary = semBCreate(SEM_Q_FIFO, SEM_FULL);

/* Note 1: lock the semaphore for scheduling purposes */
semTake(semBinary,WAIT_FOREVER);

/* spawn the two tasks */
taskIdOne = taskSpawn("t1",90,0x100,2000,(FUNCPTR)taskOne,0,0,0,0,0,0,0,0,0,0);
taskIdTwo = taskSpawn("t2",90,0x100,2000,(FUNCPTR)taskTwo,0,0,0,0,0,0,0,0,0,0);
}
```

```
void taskOne(void)
{
int i;
for (i=0; i < ITER; i++)
          {
          semTake(semBinary,WAIT_FOREVER); /* wait indefinitely for semaphore */
          printf("I am taskOne and global = %d.....................\n", ++global);
          semGive(semBinary); /* give up semaphore */
          }
}

void taskTwo(void)
{
int i;
semGive(semBinary); /* Note 2: give up semaphore(a scheduling fix) */
for (i=0; i < ITER; i++)
          {
          semTake(semBinary,WAIT_FOREVER); /* wait indefinitely for semaphore */
          printf("I am taskTwo and global = %d----------------------\n", --global);
          semGive(semBinary); /* give up semaphore */
          }
}
--------------------------------------------------------------------------------
```

Procedures

1. Copy the source code in the example and compile it.
2. Load the object file onto the target machine.
3. Run the examples by executing the main routine("binary",etc.) of the example on WindSh terminal.

Note: Make sure you have redirected I/O, otherwise you won't see the results of the printf commands.

Follow On Experiment

Experiment 1. In the binary semaphore example, remove the source code associated with comments Notes 1 and Notes 2. Compile and run the program. Is there a difference in the output? Is so, explain the reasons why.

Experiment 2. Write a program that toggles the value of "global" between 1 and 0 using

counting semaphores. Just edit the example code to make appropriate modifications.

Additional Information

Refer to VxWorks User's Manual and Reference Manual.