École Centrale de Nantes

M2 - CORO-Embedded Real-Time Systems

Adding A Service Call in Trampoline

Authors:
Ragesh Ramachandran
Gopi Krishnan Regulan

Supervisors:
Mikaël Briday

January 24, 2019

Contents

Chapter 1

Adding a service call in Trampoline

1.1 Objective

The goal of the lab is to implement the system service related the system calls $SemWait()$ to lock the semaphore and $SemPost()$ to unlock a semaphore. Using the result a buffer needs to be protected using semaphores.

1.2 Semaphores

Semaphores are used to protect the shared resources and this mechanism offers three function

- $init()$ to initialize the semaphore.
- $SemWait()$ to test the semaphore.
- $SemPost()$ to increment the semaphore.

A counter is associated with the semaphore

- A call to SemWait() is used to ask for resource access:
  - If the counter is > 0, it is decremented and the resource may be taken.
  - If the counter is $= 0$, the task which called P() is put in the waiting state until the counter became > 0.
  - At that time the task will be awaken and the counter will be decremented again.
- A call to SemPost() is used to release a resource:
  - The counter is incremented and a task which is waiting for the resource may be put in ready state.

1.3 System service

Question 1 *Implement the **SemWait()** service*

The code for **SemWait()** is implemented. If the task is activated more than once then the result is **E_OS_ACCESS** else the service is called. The following actions are performed.

- If there are at least one token, the number of token is decremented and the service call ends.
- If there are no more tokens, the running task should be blocked, and the task id should be saved.

```
FUNC(tpl_status, OS_CODE)..
                .. tpl_sem_wait_service(CONST(SemType, AUTOMATIC) sem_id)
{
  GET_CURRENT_CORE_ID(core_id)
  GET_TPL_KERN_FOR_CORE_ID(core_id, kern)
  VAR(tpl_status, AUTOMATIC) result = E_OK;
  VAR(tpl_task_id, AUTOMATIC) task_id;
  P2CONST(tpl_proc_static, AUTOMATIC, OS_APPL_DATA)  s_task;
  CONSTP2VAR(tpl_semaphore, AUTOMATIC, OS_CONST) sem = tpl_sem_table[sem_id];

  task_id = tpl_kern.running_id;
  s_task = tpl_stat_proc_table[task_id];
  LOCK_KERNEL()
  if(s_task->max_activate_count >= 1){
   result = E_OS_ACCESS;
  }
  else{
   if(sem->token >= 1) {
      sem->token--;
   }
   else{
      tpl_block();
      sem->waiting_tasks[sem->index] = task_id;
      sem->index =(sem->index+1)%TASK_COUNT;
      sem->size++;
   }
  }
  UNLOCK_KERNEL()
  return result;
}
```

Question 2 *Implement the **SemPost()** service*

When the service is called: This service always returns **E_OK**

- if there is at least one task that is blocked by the semaphore, then the oldest task is released (FIFO), and a reschedule is done (as we update the ready list).
- if no task is bloked, then the number of token is incremented.

```
FUNC(tpl_status, OS_CODE) tpl_sem_post_service(CONST(SemType, AUTOMATIC) sem_id)
{
  GET_CURRENT_CORE_ID(core_id)
  GET_TPL_KERN_FOR_CORE_ID(core_id, kern)
  VAR(tpl_task_id, AUTOMATIC) task_id;
  CONSTP2VAR(tpl_semaphore, AUTOMATIC, OS_CONST) sem = tpl_sem_table[sem_id];

  LOCK_KERNEL()
  if(sem->size >= 1)
  {
     VAR(tpl_task_id, AUTOMATIC) ..
      .. task_blocked_id = sem->waiting_tasks[sem->index-sem->size];
          tpl_release(task_blocked_id);
          sem->size--;
  }
  else{
   sem->token++;
  }
  UNLOCK_KERNEL()

  return E_OK;
}
```

1.4 Test application

Question 3 *The buffer should be protected with 2 semaphores (overflow/underflow). Update the provided application to test the semaphores.*

The critical resource *nbItem* is protected using the semaphores which is implemented in the following code.

```
#include "tpl_os.h"
#include "stm32f30x.h"
#include "stm32f30x_rcc.h"
#include "stm32f30x_gpio.h"
#include "mcp23s17.h"
#ifdef __cplusplus
 extern "C" {
#endif /* __cplusplus */

DeclareSemaphore(overflow);
Declaresemaphore(underflow);

int nbItem = 0;

int getBufferSize() {return nbItem;};

void displaySize()
{
   int bufferSize = getBufferSize();
   if((nbItem >=0) && (nbItem <=8))
   ioExt.clearBits(mcp23s17::PORTA,0xff);
   ioExt.setBits(mcp23s17::PORTA,(1<<nbItem)-1);
}

void bufferWrite()
{
   SemWait(overflow);
   nbItem++;
   SemPost(underflow);
   displaySize();
}

void bufferRead()
{
   SemWait(underflow);
   nbItem--;
   SemPost(overflow);
   displaySize();
}
```

```
#ifdef __cplusplus
}
#endif /* __cplusplus */
```

Question 4 *use the application to validate your semaphore implementation. You can use the tft display to print information, or use directly the debugger. Explain your tests scenarios.*

The semaphore implementation is validated using gdb debugger. The interaction of the computer with the micro-controller is enabled using the ST-Link tool. To enable the ST tool we run the command:

```
st-util
```

Once the link is established the debugger *gdb* is started with the binary file *semTest.elf* using the command:

```
arm-none-eabi-gdb -tui semTest.elf
```

The basic commands used are

- **c** -resumes the execution.
- **s** -execute one instruction, step into functions.
- **n** -execute one line, step over functions.

1.5 Test scenarios

The file init.gdb is loaded in the debugger using the command given below

```
source init.gdb
```

The break point is set on *tpl_sem_wait_service* and then the program continues and the result is displayed using

```
display *(tpl_kern->running)
```

The value is analyzed for every button press to check the correctness

Now, the break point is set on *tpl_sem_post_service* and then the program continues and the result is displayed using

```
display *(tpl_kern->running)
```

The value is analyzed for every button press to check the correctness Now the break point is set on bufferWrite and then on bufferRead. The program continues without any error.

The system call for the semaphore is implemented on the trampoline and verified using the gdb debugger.