|  |
| --- |
|  |
| **EE489 / EE589 Real-Time Embedded Systems Design** |
| Lab #6  *Dianzhi Yu* |
|  |
| *3/7/2018* |

|  |
| --- |
|  |

# Introduction

Rewriting vPrintString() to use a semaphore. A mutex can ensure each task gets mutually exclusive access to the terminal, even when pre-emption occurs, the strings that are displayed will be correct and in no way corrupted.

Rewriting vPrintString() to use a gatekeeper task. Gatekeeper can provide a clean method of avoiding mutual exclusion without the risk of priority inversion or deadlock.

1. **A list of all FreeRTOS API functions being used**

xMutex=xSemaphoreCreateMutex();

First, define the global parameter xMutex. This is used to reference the mutex type semaphore that is used to ensure mutual exclusive access to stdout. Then call the xSemaphoreCreateMutex() API function to create a mutex. The return value is assigned to the newly created variable xMutex.

xTaskCreate(prvPrintTask, "Print1", 240, "Task 1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\r\n", 1, NULL );

xTaskCreate(prvPrintTask, "Print2", 240, "Task 2 ------------------------------------------\r\n", 2, NULL );

Create two tasks, and then starts the scheduler. The two instances of prvPrintTask are created at different priorities.

prvNewPrintString();

Use a mutex to control access to standard out. This function will be called repeatedly by two instances of a task implemented by prvPrintTask.

prvNewPrintString(pcStringToPrint);

Print out the string using the newly defined function.

xSemaphoreTake(xMutex,portMAX\_DELAY);

xSemaphoreGive(xMute);

A mutex is a special type of Semaphore. It must always be returned.

xTaskCreate( prvPrintTask, “Print1”, 240, (void \*)0, 1, NULL);

xTaskCreate( prvPrintTask, “Print2”, 240, (void \*)1, 2, NULL);

xTaskCreate(prvStdioGateKeeperTask, “Gatekeeper”, 240, NULL, 0, NULL);

Assign gate keeper task a lower priority than the print tasks, so messages sent to the gate keeper remain in the queue until both print tasks are in the Blocked state.

xQueueReceive(xPrintQueue, &pcMessageToPrint, portMAX\_DELAY);

Wait for a message to arrive in the gatekeeper task.

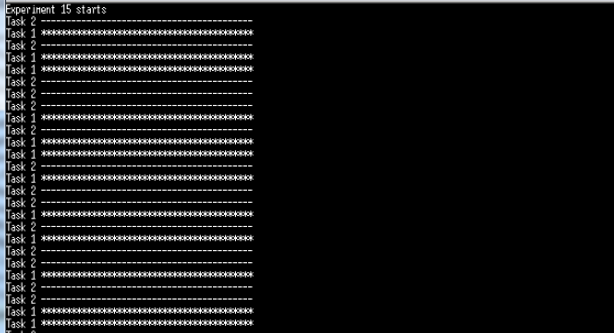
xQueueSendToBack(xPrintQueue, &pcMessageToPrint, portMAX\_DELAY);

Send the string on the queue to the gatekeeper task, rather than written out directly.

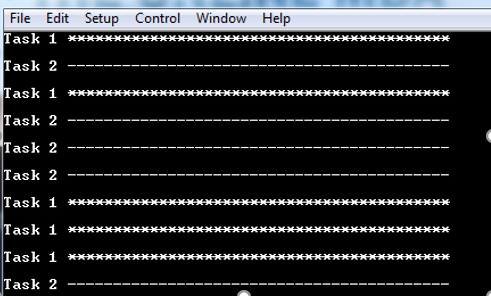
xQueueSendToFrontFromISR(xPrintQueue, &pcStringsToPrint, &xHigherPriorityTaskWoken);

Print out a message every 200 ticks. The message is not written out directly, but sent to the gatekeeper task.

1. **Screen shots of the program execution results or debug windows of Keil µVision**



Task 1 takes the mutex and starts to write out its string. Task 2 has higher priority and attempts to take the mutex, but the mutex is still held by task 1. So task 2 enters the block state to wait task 1 completes the execution until the mutes is gave back.



When a task writes a message to the terminal, it does not call a print function directly but sends the message to the gatekeeper. The gatekeeper task uses a FreeRTOS queue to serialize access to the terminal.So it is the only task permitted to access the terminal directly and the internal implementation of the task does not have to consider mutual exclusion.

1. **Conclusion**

It is important to maintain data consistency at all time when the access to a resource is shared between tasks or between tasks and interrupts. So, mutex is considered as a token associated with the resource being shared in mutual exclusion scenario. But it stills has some potential pitfall of using mutex exclusion, like priority inversion and deadlock. To solve the problems, creating a gatekeeper task is an alternative choice. The gatekeeper task spends most of its time in the blocked state, waiting for messages to arrive on the queue. When a message arrives, the gatekeeper writes the message to the standard out, before returning to the Blocked state to wait for the next message. The internal implementation of the task does not have to consider mutual exclusion because it is the only task permitted to access the terminal directly.

1. **Appendix: The source code (main.c) with sufficient comments.**

/\*\*ex15

\* \mainpage User Application template doxygen documentation

\*

\* \par Empty user application template

\*

\* Bare minimum empty user application template

\*

\* \par Content

\*

\* -# Include the ASF header files (through asf.h)

\* -# "Insert system clock initialization code here" comment

\* -# Minimal main function that starts with a call to board\_init()

\* -# "Insert application code here" comment

\*

\*/

Ex16

/\*

\* Include header files for all drivers that have been imported from

\* Atmel Software Framework (ASF).

\*/

#include <asf.h>

#include "semphr.h"

#include "task.h"

/\* Dimensions the buffer into which messages destined for stdout are placed. \*/

#define mainMAX\_MSG\_LEN ( 80 )

/\* The task to be created. Two instances of this task are created. \*/

static void prvPrintTask( void \*pvParameters );

/\* The function that uses a mutex to control access to standard out. \*/

static void prvNewPrintString( const portCHAR \*pcString );

/\* To-do: Declare a variable named as xMutex with the type xSemaphoreHandle.

Declare a variable of type xSemaphoreHandle. This is used to reference the

mutex type semaphore that is used to ensure mutual exclusive access to stdout. \*/

xSemaphoreHandle xMutex;

void vApplicationIdleHook(void)

{

while(1);

}

/\*\* UART module for debug. \*/

static struct usart\_module cdc\_uart\_module;

/\*\*

\* \brief Configure UART console.

\*/

static void configure\_console(void)

{

struct usart\_config usart\_conf;

usart\_get\_config\_defaults(&usart\_conf);

usart\_conf.mux\_setting = EDBG\_CDC\_SERCOM\_MUX\_SETTING;

usart\_conf.pinmux\_pad0 = EDBG\_CDC\_SERCOM\_PINMUX\_PAD0;

usart\_conf.pinmux\_pad1 = EDBG\_CDC\_SERCOM\_PINMUX\_PAD1;

usart\_conf.pinmux\_pad2 = EDBG\_CDC\_SERCOM\_PINMUX\_PAD2;

usart\_conf.pinmux\_pad3 = EDBG\_CDC\_SERCOM\_PINMUX\_PAD3;

usart\_conf.baudrate = 115200;

stdio\_serial\_init(&cdc\_uart\_module, EDBG\_CDC\_MODULE, &usart\_conf);

usart\_enable(&cdc\_uart\_module);

}

/\* Used as a loop counter to create a very crude delay. \*/

#define mainDELAY\_LOOP\_COUNT ( 0x11ffff )

/\*-----------------------------------------------------------\*/

int main( void )

{

system\_init();

configure\_console();

printf("Experiment 15 starts\r\n");

//------------------------------------------------------

/\* To-do: Call the xSemaphoreCreateMutex() API function to create a mutex.

The return value is assigned to the newly created variable xMutex.

Before a semaphore is used it must be explicitly created. In this example

a mutex type semaphore is created. \*/

xMutex=xSemaphoreCreateMutex();

/\* The tasks are going to use a pseudo random delay, seed the random number

generator. \*/

srand( 567 );

/\* To-do: Check if xMutex is created successfully by comparing it with NULL.

If it is not NULL, we created two tasks and start the scheduler.

Only create the tasks if the semaphore was created successfully. \*/

if(xMutex != NULL)

{

/\* Create two instances of the tasks that attempt to write stdout. The

string they attempt to write is passed in as the task parameter. The tasks

are created at different priorities so some pre-emption will occur. \*/

xTaskCreate( prvPrintTask, "Print1", 240, "Task 1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\r\n", 1, NULL );

xTaskCreate( prvPrintTask, "Print2", 240, "Task 2 ------------------------------------------\r\n", 2, NULL );

/\* Start the scheduler so the created tasks start executing. \*/

vTaskStartScheduler();

}

//------------------------------------------------------

/\* If all is well we will never reach here as the scheduler will now be

running. If we do reach here then it is likely that there was insufficient

heap available for the idle task to be created. \*/

for( ;; );

}

//------------------------------------------------------------

static void prvNewPrintString( const portCHAR \*pcString )

{

static char cBuffer[ mainMAX\_MSG\_LEN ];

/\* To-do: Call the xSemaphoreTake() API function to take the mutex.

Two parameters are needed. The first one is the mutex variable you created

previously; The second is the block time specified as portMAX\_DELAY.

The semaphore is created before the scheduler is started so already

exists by the time this task executes.

Attempt to take the semaphore, blocking indefinitely if the mutex is not

available immediately. The call to xSemaphoreTake() will only return when

the semaphore has been successfully obtained so there is no need to check the

return value. If any other delay period was used then the code must check

that xSemaphoreTake() returns pdTRUE before accessing the resource (in this

case standard out. \*/

xSemaphoreTake(xMutex,portMAX\_DELAY);

{

/\* The following line will only execute once the semaphore has been

successfully obtained - so standard out can be accessed freely. \*/

sprintf( cBuffer, "%s", pcString );

printf( cBuffer );

}

/\* To-do: Call the xSemaphoreGive() API function to take the mutex.

One parameter is needed. It is the mutex variable you created previously.

\*/

xSemaphoreGive(xMutex);

}

static void prvPrintTask( void \*pvParameters )

{

char \*pcStringToPrint;

/\* Two instances of this task are created so the string the task will send

to prvNewPrintString() is passed in the task parameter. Cast this to the

required type. \*/

pcStringToPrint = ( char \* ) pvParameters;

for( ;; )

{

/\* To-do: Call the prvNewPrintString() function by passing the variable

pcStringToPrint as the parameter.

Print out the string using the newly defined function. \*/

prvNewPrintString(pcStringToPrint);

/\* Wait a pseudo random time. Note that rand() is not necessarily

re-entrant, but in this case it does not really matter as the code does

not care what value is returned. In a more secure application a version

of rand() that is known to be re-entrant should be used - or calls to

rand() should be protected using a critical section. \*/

vTaskDelay( ( rand() & 0x1FF ) );

}

}

/\*\*

\* \mainpage User Application template doxygen documentation

\*

\* \par Empty user application template

\*

\* Bare minimum empty user application template

\*

\* \par Content

\*

\* -# Include the ASF header files (through asf.h)

\* -# "Insert system clock initialization code here" comment

\* -# Minimal main function that starts with a call to board\_init()

\* -# "Insert application code here" comment

\*

\*/

/\*

\* Include header files for all drivers that have been imported from

\* Atmel Software Framework (ASF).

\*/

#include <asf.h>

#include "semphr.h"

#include "task.h"

/\* Dimensions the buffer into which messages destined for stdout are placed. \*/

#define mainMAX\_MSG\_LEN ( 80 )

/\* The task to be created. Two instances of this task are created. \*/

static void prvPrintTask( void \*pvParameters );

/\* The gatekeeper task itself. \*/

static void prvStdioGatekeeperTask( void \*pvParameters );

/\* Define the strings that the tasks and interrupt will print out via the gatekeeper. \*/

static char \*pcStringsToPrint[] =

{

"Task 1 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\r\n",

"Task 2 ----------------------------------\r\n",

"Message printed from tick hook interrupt #\r\n"

};

/\*-----------------------------------------------------------\*/

/\* To-do: Declare a variable xPrintQueue of the type xQueueHandle.

This is used to send messages from the print tasks to the gatekeeper task. \*/

xQueueHandle xPrintQueue;

void vApplicationIdleHook(void)

{

while(1);

}

/\*\* UART module for debug. \*/

static struct usart\_module cdc\_uart\_module;

/\*\*

\* \brief Configure UART console.

\*/

static void configure\_console(void)

{

struct usart\_config usart\_conf;

usart\_get\_config\_defaults(&usart\_conf);

usart\_conf.mux\_setting = EDBG\_CDC\_SERCOM\_MUX\_SETTING;

usart\_conf.pinmux\_pad0 = EDBG\_CDC\_SERCOM\_PINMUX\_PAD0;

usart\_conf.pinmux\_pad1 = EDBG\_CDC\_SERCOM\_PINMUX\_PAD1;

usart\_conf.pinmux\_pad2 = EDBG\_CDC\_SERCOM\_PINMUX\_PAD2;

usart\_conf.pinmux\_pad3 = EDBG\_CDC\_SERCOM\_PINMUX\_PAD3;

usart\_conf.baudrate = 115200;

stdio\_serial\_init(&cdc\_uart\_module, EDBG\_CDC\_MODULE, &usart\_conf);

usart\_enable(&cdc\_uart\_module);

}

/\* Used as a loop counter to create a very crude delay. \*/

#define mainDELAY\_LOOP\_COUNT ( 0x11ffff )

/\*-----------------------------------------------------------\*/

int main( void )

{

system\_init();

configure\_console();

printf("Experiment 16 starts\r\n");

//------------------------------------------------------

/\* To-do: Call the xQueueCreate() API function to create a queue.

The first parameter is set to 5 which specifies the maximum number of items.

The second parameter is the size of char\*.

The return value is assigned to the newly created variable xPrintQueue.\*/

xPrintQueue = xQueueCreate(5,sizeof(char\*));

/\*Before a queue is used it must be explicitly created. The queue is created

to hold a maximum of 5 character pointers. \*/

/\* The tasks are going to use a pseudo random delay, seed the random number

generator. \*/

srand( 567 );

/\* Check the queue was created successfully. \*/

if( xPrintQueue != NULL )

{

/\* Create two instances of the tasks that send messages to the gatekeeper.

The index to the string they attempt to write is passed in as the task

parameter (4th parameter to xTaskCreate()). The tasks are created at

different priorities so some pre-emption will occur. \*/

xTaskCreate( prvPrintTask, "Print1", 240, ( void \* ) 0, 1, NULL );

xTaskCreate( prvPrintTask, "Print2", 240, ( void \* ) 1, 2, NULL );

/\* Create the gatekeeper task. This is the only task that is permitted

to access standard out. \*/

xTaskCreate( prvStdioGatekeeperTask, "Gatekeeper", 240, NULL, 0, NULL );

/\* Start the scheduler so the created tasks start executing. \*/

vTaskStartScheduler();

}

//------------------------------------------------------

/\* If all is well we will never reach here as the scheduler will now be

running. If we do reach here then it is likely that there was insufficient

heap available for the idle task to be created. \*/

for( ;; );

}

/\*-----------------------------------------------------------\*/

static void prvStdioGatekeeperTask( void \*pvParameters )

{

char \*pcMessageToPrint;

static char cBuffer [mainMAX\_MSG\_LEN];

/\* This is the only task that is allowed to write to the terminal output.

Any other task wanting to write to the output does not access the terminal

directly, but instead sends the output to this task. As only one task

writes to standard out there are no mutual exclusion or serialization issues

to consider within this task itself. \*/

for( ;; )

{

/\* To-do: Call xQueueReceive() API.

The first parameter is the queue variable xPrintQueue you declared at the beginning.

The second parameter is the reference to the variable pcMessageToPrint.

The last parameter is portMAX\_DELAY. \*/

/\* Wait for a message to arrive. \*/

xQueueReceive(xPrintQueue, &pcMessageToPrint, portMAX\_DELAY);

/\* There is no need to check the return value as the task will block

indefinitely and only run again when a message has arrived. When the

next line is executed there will be a message to be output. \*/

sprintf(cBuffer, "%s", pcMessageToPrint);

printf( cBuffer );

/\* Now simply go back to wait for the next message. \*/

}

}

/\*-----------------------------------------------------------\*/

void vApplicationTickHook( void )

{

static int iCount = 0;

portBASE\_TYPE xHigherPriorityTaskWoken = pdFALSE;

/\* Print out a message every 200 ticks. The message is not written out

directly, but sent to the gatekeeper task. \*/

iCount++;

if( iCount >= 200 )

{

/\* To-do: Call xQueueSendToFrontFromISR() API.

The first parameter is the queue variable xPrintQueue you declared at the beginning.

The second parameter is the reference to the last element of

the array variable pcStringsToPrint.

The last parameter is the reference to the variable xHigherPriorityTaskWoken. \*/

xQueueSendToFrontFromISR(xPrintQueue, &pcStringsToPrint, &xHigherPriorityTaskWoken);

/\* In this case the last parameter (xHigherPriorityTaskWoken) is not

actually used but must still be supplied. \*/

/\* Reset the count ready to print out the string again in 200 ticks time. \*/

iCount = 0;

}

}

static void prvPrintTask( void \*pvParameters )

{

int iIndexToString;

/\* Two instances of this task are created so the string the task will send

to prvNewPrintString() is passed in the task parameter. Cast this to the

required type. \*/

iIndexToString = ( int ) pvParameters;

for( ;; )

{

/\* Print out the string, not directly but by passing the string to the

gatekeeper task on the queue. The queue is created before the scheduler is

started so will already exist by the time this task executes. A block time

is not specified as there should always be space in the queue. \*/

xQueueSendToBack( xPrintQueue, &( pcStringsToPrint[ iIndexToString ] ), 0 );

/\* Wait a pseudo random time. Note that rand() is not necessarily

re-entrant, but in this case it does not really matter as the code does

not care what value is returned. In a more secure application a version

of rand() that is known to be re-entrant should be used - or calls to

rand() should be protected using a critical section. \*/

vTaskDelay( ( rand() & 0x1FF ) );

}

}