Introduction

So far, all work has been done using self-written code. For small systems this can be sufficient, and the question remains whether preemptive scheduling is a necessity for these systems at all! However, for bigger systems with lots of different tasks, a real-time operating system might be better suitable. In this assignment you will use the open-source and free Real-Time Operating System called FreeRTOS[1], which is managed by Amazon and which is supported by STM32CubeIDE. There is a standard POSIX[2] API[3] available for FreeRTOS which can be used to define tasks and inter task communication.

Assignments week 5 – FreeRTOS and pthreads

This week you will learn how to:

- analyze the producer-consumer synchronization problem solved with semaphores in the context of a Real Time Operating System (FreeRTOS);
- use a message queue to solve the producer-consumer synchronization problem;
- use a mutex to replace a binary semaphore.

In this assignment you will use FreeRTOS, a free Real-Time Operating System. There is a standard POSIX API available which can be used to define tasks and inter task communication. In this assignment you will be introduced to threads (tasks) and semaphores (an inter task communication device) by analyzing an existing multithreaded program. We will analyze the properties of the real-time scheduler used in FreeRTOS in particular. In the last parts of this assignment you will use a POSIX `mqueue` (message queue) and `mutex` (a synchronization device which provides mutual exclusivity) to simplify the program.

The problem we are going to analyze is the so-called producer-consumer synchronization problem. See Figure 1.

The main file of the example program `main.c` creates of three threads: two producers and one consumer. Each producer produces one data element at a time and puts this into a shared buffer. The consumer reads one data element at a time from this buffer and consumes the data.

[1] https://www.freertos.org/

[2] The Portable Operating System Interface (POSIX) is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems.

[3] https://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_POSIX/index.html.

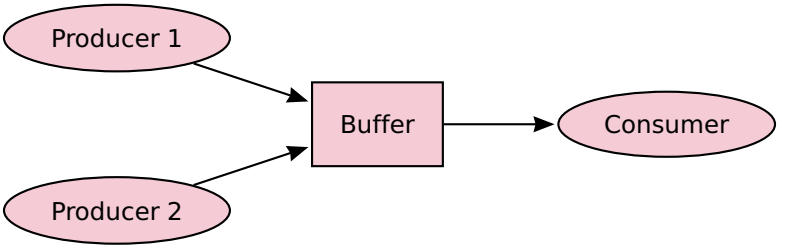

Figure 1: Two producers and one consumer communicating via a buffer.

Obviously, it must be ensured that the producers and the consumer synchronize:

- The consumer has to wait when the buffer is empty.
- A producer has to wait when the buffer is full.
- The consumer and producers have to wait on each other when using shared variables (used to implement the buffer).

This synchronization can be realized with the help of three semaphores:

- The semaphore `semEmpty` counts the number of empty places. This semaphore is initialized with the number of places in the buffer. A producer calls a `sem_wait` on this semaphore before placing a data element into the buffer. As a result, the semaphore is decremented by 1 every time an element is placed into the buffer. If the semaphore `semEmpty` is 0, the producer must wait (there are no more empty places, so the buffer is full). After the consumer has taken a data element from the buffer, a `sem_post` is executed on the semaphore `semEmpty`. As a result, the semaphore is incremented by 1 each time a place has been emptied.
- The semaphore `semFilled` counts the number of filled places. This semaphore is initialized with 0. The consumer calls a `sem_wait` on this semaphore before a data element is taken from the buffer. As a result, the semaphore is decremented by 1 every time one place is emptied in the buffer. If the semaphore `semFilled` is 0, the consumer must wait (there are no more filled places, so the buffer is empty). After a producer has put a data element into the buffer, a `sem_post` is executed on the semaphore `semFull`. As a result, the semaphore is incremented by 1 each time one element is put into the buffer.
- The semaphore `semMutualExclusive` ensures mutual exclusion. This semaphore is initialized with 1. The consumer and producers call a `sem_wait` on this semaphore before using the shared variables. When they have finished using the shared variables,

a sem_post is executed on this semaphore. This ensures that only one thread at a time can use the shared variables.

The buffer is globally defined as follows:

```
#define SIZE 8
char buffer[SIZE]; // buffer which can store SIZE elements of type
 ↪ char
int indexGet = 0; // index where the next element will be read
int indexPut = 0; // index where the next element will be written
sem_t semMutualExclusive; // binary semaphore: used for mutual
 ↪ exclusive use of the buffer
sem_t semEmpty; // counting semaphore: counts the number of empty
 ↪ places
sem_t semFilled; // counting semaphore: count the number of filled
 ↪ places
```

The use of global variables is of course "not done". All these variables can be places in a structure and passed to the threads by using the `void *` argument. But to begin with, the use of global variables is simpler.

The type sem_t is defined in the header file semaphore.h. The link refers to the POSIX standard (IEEE Std 1003.1) documentation.

A character can be written into the buffer using the function put:

```
void put(char c)
{
    check_errno( sem_wait(&semEmpty) ); // lower the number of
 ↪ empty places, WAIT if there are no free places left!
    check_errno( sem_wait(&semMutualExclusive) ); // enter
 ↪ critical region
    buffer[indexPut] = c;
    indexPut++;
    if (indexPut == SIZE)
    {
        indexPut = 0;
    }
    check_errno( sem_post(&semMutualExclusive) ); // leave
 ↪ critical region
    check_errno( sem_post(&semFilled) ); // increase the number of
 ↪ filled places
}
```

The sem_wait and sem_post functions return the value 0 if no error has occurred. In case of an error, -1 is returned and the standard global variable errno is filled with the error number. In the example program the function check_errno is used to check this return value. In case of an error, this function prints an appropriate error message using the standard function perror and enters an infinite loop.

```
void check_errno(int error)
{
    if (error < 0)
    {
        perror("Error");
        while (1);
    }
}
```

A character can be read from the buffer using the function get:

```
char get(void)
{
    check_errno( sem_wait(&semFilled) ); // lower the number of ↩
 ↪ filled places, WAIT if there are no filled places left!
    check_errno( sem_wait(&semMutualExclusive) ); // enter ↩
 ↪ critical region
    char c = buffer[indexGet];
    indexGet++;
    if (indexGet == SIZE)
    {
        indexGet = 0;
    }
    check_errno( sem_post(&semMutualExclusive) ); // leave ↩
 ↪ critical region
    check_errno( sem_post(&semEmpty) ); // increase the number of ↩
 ↪ empty places
    return c;
}
```

The semaphores are initialized in the main_thread function using the function sem_init:

```
    check_errno( sem_init(&semMutualExclusive, 0, 1) ); // allow ↩
 ↪ one thread exclusively in critical section
    check_errno( sem_init(&semEmpty, 0, SIZE) ); // there are SIZE ↩
 ↪ empty places
```

```
    check_errno( sem_init(&semFilled, 0, 0) ); // there are 0 ↩
 ↪ filled places
```

The two producer threads execute the same code:

```
void *producer(void *arg) // function for producer thread
{
    char c = *(char *)arg;
    check_errno( sem_wait(&semPrintf) );
    check_errno( printf("Thread: %p with argument: %c starts\n", ↩
 ↪ pthread_self(), c) );
    check_errno( sem_post(&semPrintf) );
    for (int i = 0; i < 100; ++i)
    {
        put(c);
    }
    check_errno( sem_wait(&semPrintf) );
    check_errno( printf("Thread: %p stops\n", pthread_self()) );
    check_errno( sem_post(&semPrintf) );
    return NULL;
}
```

The character to be produced is passed on as an argument (when starting the thread). 100 characters are placed in the buffer using the function `put`. Using `printf`, the starting and stopping of the thread is reported. The function `pthread_self` is used to retrieve the thread ID. An extra semaphore `semPrintf` has been defined which ensures that the output is not alternated with the output of other threads.

The consumer thread executes the following code:

```
void *consumer(void *arg) // function for consumer thread
{
    check_errno( sem_wait(&semPrintf) );
    check_errno( printf("Thread: %p starts\n", pthread_self()) );
    check_errno( sem_post(&semPrintf) );
    for (int i = 0; i < 200; ++i)
    {
        char c = get();
        check_errno( sem_wait(&semPrintf) );
        check_errno( printf("%c", c) );
        check_errno( fflush(stdout) );
        check_errno( sem_post(&semPrintf) );
```

```
    }
    check_errno( sem_wait(&semPrintf) );
    check_errno( printf("Thread: %p stops\n", pthread_self()) );
    check_errno( sem_post(&semPrintf) );
    return NULL;
}
```

200 characters are read from the buffer using the function `get`. Each character is printed using the standard function `printf`. Also, the starting and stopping of the thread is reported.

The `main_thread` is given a high priority in the main function. The `main_thread` must have the highest priority because this thread starts the other threads, and we want to study the mutual interaction of these threads.

Setting the priority of a thread is fairly complicated. First the scheduling parameters have to be retrieved from the pthread attributes using the function `pthread_attr_getschedparam`. These scheduling parameters are of the type `struct sched_param`. The data field `sched_priority` in this **`struct`** can be used to set the priority. Finally, the changed scheduling parameters must be passed to the thread attributes again using the function `pthread_attr_setschedparam`.

```
    struct sched_param sp;
    check( pthread_attr_getschedparam(&pta, &sp) );
    sp.sched_priority = 15;
    check( pthread_attr_setschedparam(&pta, &sp) );
```

The `pthread_xxx` functions have a return value of 0 if no error has occurred. In case of an error, the error number is returned. The default global variable `errno` is *not* filled with the error number. In the example program the function `check` is used to check the return value. In case of an error, this function prints an appropriate error message using the standard function `strerror` and enters an infinite loop.

```
void check(int error)
{
    if (error != 0)
    {
        printf("Error: %s\n", strerror(error));
        while (1);
    }
}
```

In the example program, one producer "bakes" *frikandellen*[4] (represented by the letter F) and the other producer "bakes" *kroketten*[5] (represented by the letter K). The consumer "eats" the frikandellen and kroketten.

Figure 2: Cora from Mora is showing a box filled with king-size frikandellen.

When starting the program, the priorities of the consumer, the frikandel producer and the kroket producer can be entered in the console. These priorities are stored in the variables `prioc`, `priop1`, and `priop2`.

When starting a thread, the scheduling priority can be specified via a variable of the type `pthread_attr_t`. A variable of the type `pthread_attr_t` can be initialized with the default values using the function `pthread_attr_init`:

```
pthread_attr_t ptac, ptap1, ptap2;
check( pthread_attr_init(&ptac) );
check( pthread_attr_init(&ptap1) );
check( pthread_attr_init(&ptap2) );
```

Then the stack size can be set by using the function `pthread_attr_setstacksize`:

```
check( pthread_attr_setstacksize(&ptac, 1024) );
check( pthread_attr_setstacksize(&ptap1, 1024) );
check( pthread_attr_setstacksize(&ptap2, 1024) );
```

Specifying the priority with which a thread has to be started is cumbersome, as discussed before. First the scheduling parameters must be retrieved (from the thread attributes) using the function `pthread_attr_getschedparam`. These scheduling parameters are of the type `struct sched_param`. The data field `sched_priority` in this `struct` can be used to set the

[4] A frikandel (plural frikandellen) is a traditional Dutch snack, a sausage-shaped meatball, see Figure 2.

[5] A kroket (plural kroketten) is also a traditional Dutch snack, meat ragout covered in breadcrumbs.

priority. Finally, the scheduling parameters must be stored in the thread parameters again using the function pthread_attr_setschedparam.

```
struct sched_param spc, spp1, spp2;
check( pthread_attr_getschedparam(&ptac, &spc) );
check( pthread_attr_getschedparam(&ptap1, &spp1) );
check( pthread_attr_getschedparam(&ptap2, &spp2) );

spc.sched_priority = prioc;
spp1.sched_priority = priop1;
spp2.sched_priority = priop2;

check( pthread_attr_setschedparam(&ptac, &spc) );
check( pthread_attr_setschedparam(&ptap1, &spp1) );
check( pthread_attr_setschedparam(&ptap2, &spp2) );
```

After the thread attributes are set, the threads can be started using the function pthread_create:

```
pthread_t ptc, ptp1, ptp2;
char frikandel = 'F', kroket = 'K';
check( pthread_create(&ptc, &ptac, consumer, NULL) );
check( pthread_create(&ptp1, &ptap1, producer, &frikandel) );
check( pthread_create(&ptp2, &ptap2, producer, &kroket) );
```

The thread IDs of these threads are of the type pthread_t and are stored in the variables ptc, ptp1, and ptp2.

Then the main_thread must wait until the other threads have ended by using function pthread_join:

```
check( pthread_join(ptc, NULL) );
check( pthread_join(ptp1, NULL) );
check( pthread_join(ptp2, NULL) );
```

When all threads are finished, the created semaphores and thread attributes are destroyed by using the functions sem_destroy and pthread_attr_destroy:

```
check_errno( sem_destroy(&semMutualExclusive) );
check_errno( sem_destroy(&semEmpty) );
check_errno( sem_destroy(&semFilled) );

check( pthread_attr_destroy(&ptac) );
check( pthread_attr_destroy(&ptap1) );
```

```
check( pthread_attr_destroy(&ptap2) );
```

5.1 You will now create a project to run the example program `main.c` on the STM32F411-E-DISCO development board.

A Download the project buffer.zip.

B Import this project into STM32CubeIDE with the menu-option File > Import..., General > Projects from Folder or Archive, Next, Archive, select the file `buffer.zip`, Open and Finish. Build and debug the project. There should appear some text in the console window. Enter the following priorities: Consumer = 3, Frikandel Producer = 2, and Kroket Producer = 1. You may want to enable the Word Wrap option in the Console window, see Figure 3.

Explain the output of the program. Be precise in your explanation. For example, explain why first all frikandellen are baked followed by all kroketten. Explain why no frikandellen are consumed after the frikandellen producer has stopped. How many snacks are stored in the buffer before the first snack is consumed?

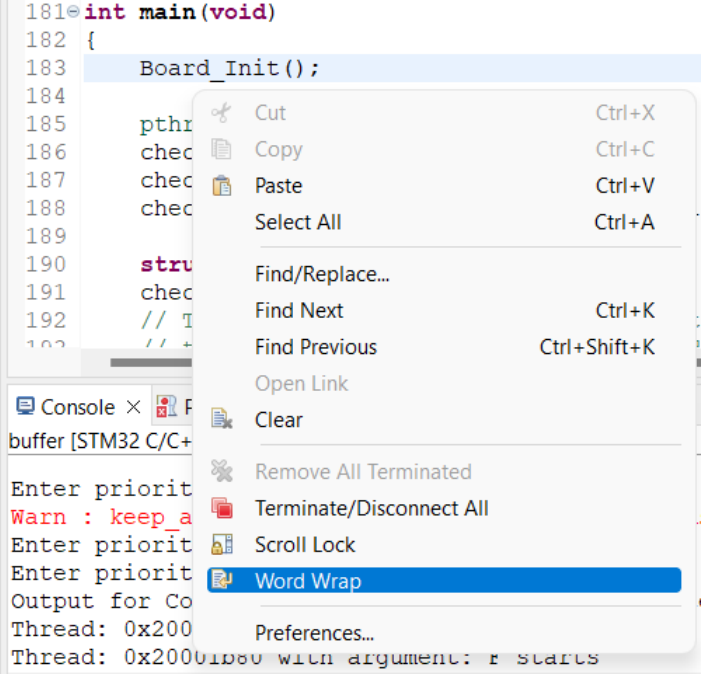

Figure 3: Enable the Word Wrap option in the Console window.

Make notes, so the teacher can easily give you feedback on this assignment.

You can use RTOS-aware debugging[6] provided by STM32CubeIDE to analyze the program. Set a breakpoint at line 58 `indexGet++;` and restart the debugger. You can now inspect the FreeRTOS objects such as tasks and semaphores. Select the menu Window > Show View > FreeRTOS > FreeRTOS Task List, see Figure 5. The FreeRTOS Task List window opens and is empty because the program is not started yet. Select the menu Window > Show View > FreeRTOS > FreeRTOS Semaphores. The FreeRTOS Semaphores window opens and is also empty because the program is not started yet.

Now run the program with priorities 3, 2, and 1. When the breakpoint is reached:

- the value of the variable c can be found in the Variables window, see Figure 4;
- the FreeRTOS Task List window displays the current state of the tasks (threads), see Figure 6;
- the FreeRTOS Semaphores window shows the semaphores, see Figure 7;
- the value of the semaphores can be found by typing their name in the Expressions window, see Figure 8.

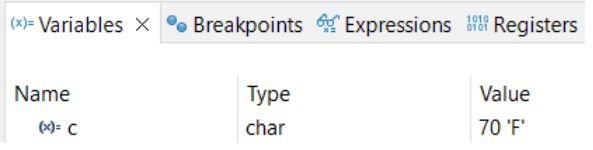

Figure 4: The value of the variable c when the breakpoint at line 58 is reached.

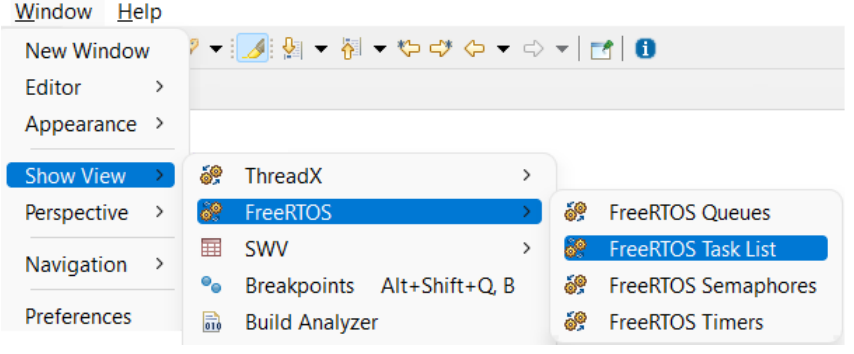

Figure 5: The available FreeRTOS-aware views.

[6] See section 6.2 of UM2609 (STM32CubeIDE user guide).

Console | Problems | Executables | Memory | FreeRTOS Task List

	Name	Priority (...	Start of Stack	Top of Stack	State
	IDLE	0/0	0x2000030c	0x200004c4 ...	READY
	pthread	15/15	0x20001110	0x2000140c ...	BLOCKED
	pthread	1/1	0x200021d8	0x2000258c ...	READY
→	pthread	3/3	0x200016a8	0x20001984 ...	RUNNING
	pthread	2/2	0x20001c40	0x20001f34 ...	READY

Figure 6: The state of the tasks (threads) when the breakpoint at line 58 is reached.

Console | Problems | Executables | Memory | FreeRTOS Task List | FreeRTOS Semaphores

Name	Address	Type	Size	Free	# Blocked tasks
semEmpty	0x200000fc	COUNTING_SEMAPHORE	32767	0	0
semFilled	0x20000150	COUNTING_SEMAPHORE	32767	0	0
semMutualExclusive	0x200000a8	COUNTING_SEMAPHORE	32767	0	0
semPrintf	0x200001a4	COUNTING_SEMAPHORE	32767	0	0

Figure 7: The semaphores when the breakpoint at line 57 is reached.

Varia... | Brea... | Expr... | Regi... | Live ... | SFRs

Expression	Type	Value
semEmpty	sem_t	{...}
xSemaphore	StaticSemaphore_t	{...}
value	int	7
semFilled	sem_t	{...}
xSemaphore	StaticSemaphore_t	{...}
value	int	0
semMutualExclusi	sem_t	{...}
xSemaphore	StaticSemaphore_t	{...}
value	int	0
semPrintf	sem_t	{...}
xSemaphore	StaticSemaphore_t	{...}
value	int	1
Add new expressior		

Figure 8: The value of the semaphores when the breakpoint at line 58 is reached.

You can make the following observations:

- There is a thread called `IDLE` with priority 0 which is ready to run. This task is created automatically by FreeRTOS, and it runs when there is no other ready task available.

- The `main_thread` with priority 15 is blocked. It is waiting for the consumer thread to finish (on line 165 `pthread_join(ptc, NULL)`).
- The consumer thread, with priority 3, is currently running. This thread has called the function `get` in which the breakpoint was hit. The call stacks of this thread can be inspected in the Debug window, see Figure 9.
- Both producer threads are ready to run.
- The value of semaphore `SemEmpty` equals 7 indicating that there are seven empty places in the buffer. The character `'F'` is already retrieved from the buffer, but this is not recorded in the state of the semaphore yet. If you step through the function until line 65 is reached and the `sem_post(&semEmpty)` has been executed, the value of `SemEmpty` is 8. The buffer is completely empty.
- The value of semaphore `SemFilled` equals 0, when you hit the breakpoint at line 58 indicating that there are zero filled places in the buffer. This is the case because the `sem_wait(&semFilled)` on line 55 has already executed when the breakpoint is hit.
- The value of semaphore `semMutualExclusive` equals zero, so the buffer is not available for another thread.
- The value of semaphore `semPrintf` equals one, so the `printf` function is available to be used by one thread.

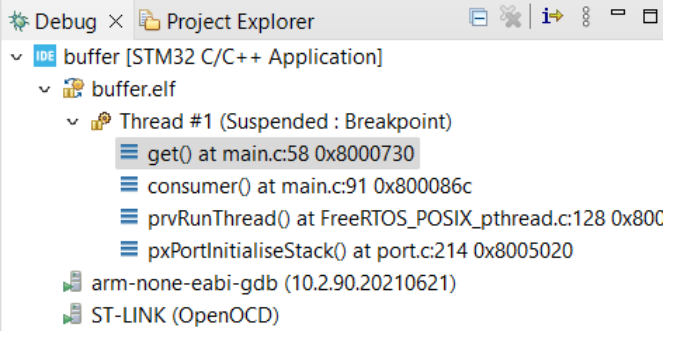

Figure 9: The call stack of the running thread can be viewed in the Debug window.

You can use these tools to help you understand what is going on. Remove the breakpoint when you are finished analyzing the program.

C The code of the consumer seems inefficient:

```
char c = get();
check_errno( sem_wait(&semPrintf) );
check_errno( printf("%c", c) );
check_errno( fflush(stdout) );
check_errno( sem_post(&semPrintf) );
```

The use of the local variable c seems unnecessary.

The following code is more compact:

```
check_errno( sem_wait(&semPrintf) );
check_errno( printf("%c", get()) );
check_errno( fflush(stdout) );
check_errno( sem_post(&semPrintf) );
```

Adjust the consumer code as discussed above. Build the program and debug it with priorities 3, 2, and 1. Explain why the program stalls. Use the FreeRTOS aware Views to help you understand what is going on.

Make notes, so the teacher can easily give you feedback on this assignment.

D Compile the (original) program and run it with the priorities: Consumer = 1, Frikandel Producer = 2 and Kroket producer = 3. Explain the output. Be precise in your explanation. Explain why the kroketten are consumed first. Explain why there are 9 kroketten still consumed after the kroketten producer has stopped. Also explain why there are nine frikandellen still consumed after the frikandel producer has stopped. Only eight of them fit in the buffer!

Is the behavior of the semaphore correct in a real-time environment? Why (not)?

E Try to predict (without debugging the program) what the flow of execution will be when the priorities are changed to: Consumer = 1, Frikandel Producer = 3 and Kroket producer = 2. Test your prediction by running the program.

F Try to predict (without debugging the program) what the flow of execution will be when the priorities are changed to: Consumer = 2, Frikandel Producer = 1 and Kroket producer = 1. Test your prediction by running the program.

G Now finally, try to predict (without debugging the program) what the flow of execution will be when the priorities are changed to: Consumer = 1, Frikandel Producer = 2 and Kroket producer = 2. Think carefully! Test your prediction by running the program.

Semaphores use shared memory to synchronize. However, it is also possible to synchronize using message passing. The IEEE Std 1003.1 defines so-called message queues for this purpose. The relevant functions are:

- `mq_open`;
- `mq_close`;
- `mq_unlink`;
- `mq_send`;
- `mq_receive`.

A very simple example using a POSIX message queue which can be run on a STM32F411E-DISCO development board is given in `mqueue.zip`.

5.2 In this assignment you are going to replace the buffer with a message queue.

A Copy the project `buffer` to `buffer_mqueue` in the Project Explorer of STM32-CubeIDE. Remove the `Debug` folder from the `buffer_mqueue` project, rename `buffer.cfg` to `buffer_mqueue.cfg`, and rename `buffer.launch` to `buffer_mqueue.launch`. Open `buffer_mqueue.launch` and replace all occurrences of `buffer` with `buffer_mqueue`. Build and debug the project `buffer_mqueueue` to verify that it is a correct copy of project `buffer`.

B Replace the global variable `buffer` and the semaphores `semMutualExclusive`, `semEmpty` and `semFilled` with a message queue.

C What priority should you give to the messages in order to implement real-time behavior? You can not verify your answer to this last question by using FreeRTOS because it's implementation of POSIX's `mqueue` does *not* support prioritized messages.[7] The fourth argument given to `mq_send` the `msg_prio` is simply ignored in the implementation.[8]

[7] See the note at: https://www.freertos.org/Documentation/api-ref/POSIX/mqueue_8h.html#a753177f77f6eec2a80b57e8a68b36bed.

[8] You can find the implementation of `mq_send` in the file `FreeRTOS_POSIX_mqueue.c` which is located in the project folder `buffer ▸ Middlewares ▸ Third_Party ▸ Lab-Project-FreeRTOS-POSIX-master ▸ FreeRTOS-Plus-POSIX ▸ source ▸`. As you can see on line 626, `mq_send` just calls `mq_timedsend` that simply ignores the `msg_prio` parameter on line 736.

The semaphore `semPrintf` is used to assure that only one thread at a time is using the function `printf`. This prevents the output from alternating due to parallel calls to `printf`. Mutual exclusivity can also be achieved by using a so-called `mutex`. The IEEE Std 1003.1 defines mutexes for this purpose. The relevant functions are:

- `pthread_mutex_init`;
- `pthread_mutex_destroy`;
- `pthread_mutex_lock`;
- `pthread_mutex_unlock`.

A very simple example using a POSIX mutex which can be run on the STM32F411E-DISCO development board is given in `mutex.zip`.

5.3 In this assignment you are going to replace the semaphore with a mutex.

A Copy the project `buffer_mqueue` to `buffer_mutex` in the same way as described in assignment 5.2 part A.

B Replace the semaphore `semPrintf` with a mutex.

Report Week 3 – 6

To conclude the second part of the course a report will have to be written.

This report should include two parts:

- the relevant source codes of the weekly assignments for week 3 to 5 and a short explanation per assignment, this explanation should also include difficulties and decisions made to finish the assignment and should provide ample evidence of the code's authenticity;
- the elaboration of the calculation task (Dutch: rekenopdracht) you will receive in week 6.

Document

The document can be written in English or in Dutch. The relevant and modified source code should be attached using color coding. Hence, the report should contain the following:

- title page with the student names, student numbers, and name of this course;

- information about each weekly assignment with source codes;
- the answers to the questions posed in the assignments;
- a brief explanation about the choices that you have made;
- a short discussion about the advantages and disadvantages of the choices you have made;
- a description of your test method and test results;
- the calculations and results of the calculating assignment of week 6, these may be handwritten and scanned in.

Delivery

The report shall be delivered as a single PDF document. The file name has the following convention: `studentnumber_surname_studentnumber_surname.pdf`. For example: `1012345_Dijkstra_1054321_Hoare.pdf`

This file must be uploaded into the the assignment in the Brightspace couse. Only one of the two students working together has to upload the file.