Assignments week 5 – TI-RTOS

So far, all work has been done using self-written code. For small systems this can be sufficient and the question remains whether pre-emptive scheduling is a necessity for these systems at all! However, for bigger systems with lots of I/O, a real-time operating system might be better suitable. In this assignment you will use TI-RTOS, a free Real-Time Operating System provided by Texas Instruments. There is a standard POSIX[1] API[2] 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 multi-threaded program. We will analyze the properties of the real-time scheduler used in TI-RTOS 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.

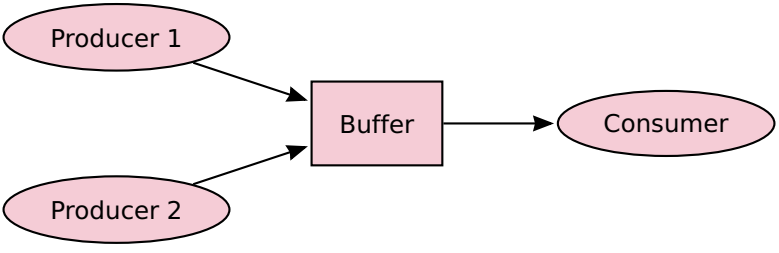

Figure 1: Two producers and one consumer communicating via a buffer.

The example program `buffer.c` consists of three threads: two producers and one consumer. Each producer produces one data element at a time and puts this into a

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

[2] https://dev.ti.com/tirex/explore/content/simplelink_cc32xx_sdk_5_20_00_06/docs/tiposix/Users_Guide.html.

shared buffer. The consumer reads one data element at a time from this buffer and consumes the data.

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( 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*[3] (represented by the letter F) and the other producer "bakes" *kroketten*[4] (represented by the letter K). The consumer "eats" the frikandellen and kroketten.

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

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

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 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 `buffer.c` on the CC3220S LaunchPad, following these steps:

- Import the example project you can find in: `C:\ti\simplelink_cc32xx_sdk_x_xx_xx_xx\examples\rtos\CC3220S_LAUNCHXL\drivers\empty\tirtos\ccs`.
- Choose Project > Properties and select (if needed) CCS General. Click on the button Manage Configurations... and activate the Debug configuration. Delete the MCU+Image configuration[5]. Finaly click on OK and Aply and Close.
- Delete `image.syscfg`, `MCU+Image`, `empty.c`, `Board.html`, `README.html`, and `README.md`
- Rename the project to `buffer_CC3220S_LAUNCHXL_tirtos_ccs`.
- Rename `main_tirtos.c` to `buffer.c`.
- Rename `empty.syscfg` to `buffer.syscfg`.
- Open the menu Project > Properties and choose Build > ARM Compiler > Advanced Options > Language Options. Now, in the dropdown box *C Dialect*, choose *Compile program in C11 mode (–c11)*, see Figure 3.
- Replace the code in `buffer.c` by the code given in `buffer.c`.

[5] This configuration is used to load the program into the Flash memory. In this case, we only want to load the into the RAM so we can debug it (using the Debug configuration).

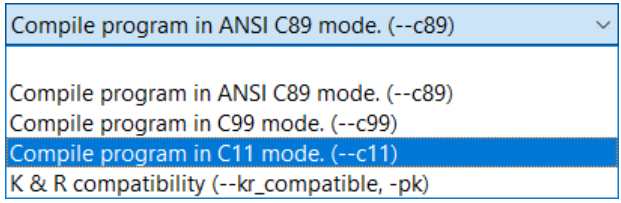

Figure 3: Choose to compile the program in C11 mode.

Compile and run this program. 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 4. 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?

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

You can use the Runtime Object View provided by Code Composer Studio to analyze the program. Set a breakpoint at line 57 `indexGet++;` and restart the program. Select the menu Tools > Runtime Object View and click on Connect, see Figure 5. Now click on Task and Semaphore and run the program with priorities 3, 2, and 1. When the breakpoint is reached the Runtime Object View displays the current state of the tasks (threads) and semaphores, see Figure 6.

You can make the following observations:

- There is a thread called `idleTask` with priority 0 which is ready to run. This task is created automatically by TI-RTOS and it runs when there is no other ready task available.
- The `main_thread` is blocked. It is waiting for the consumer thread to finish (on line 155 `pthread_join(ptc, NULL)`).

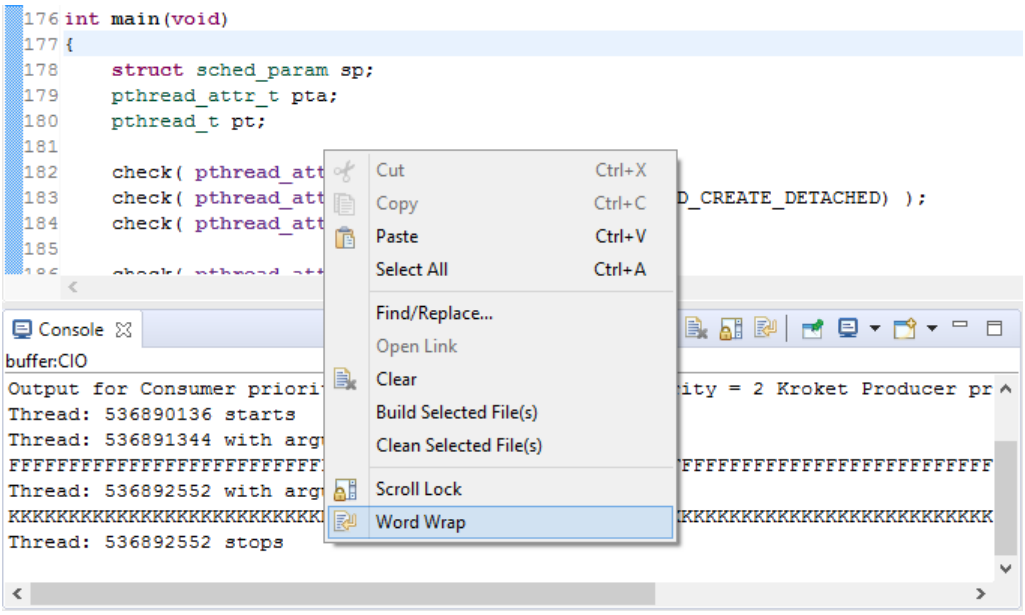

Figure 4: Enable the Word Wrap option in the Console window.

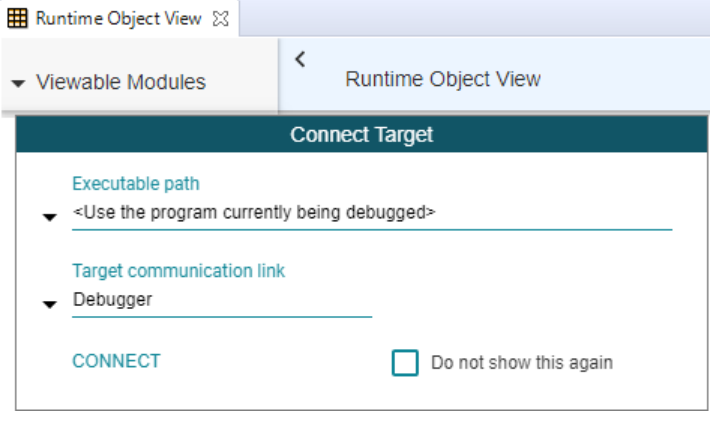

Figure 5: Click on Connect in the Runtime Object View.

- The consumer thread is currently running. This thread has called the function `get` in which the breakpoint was hit. The call stacks of the

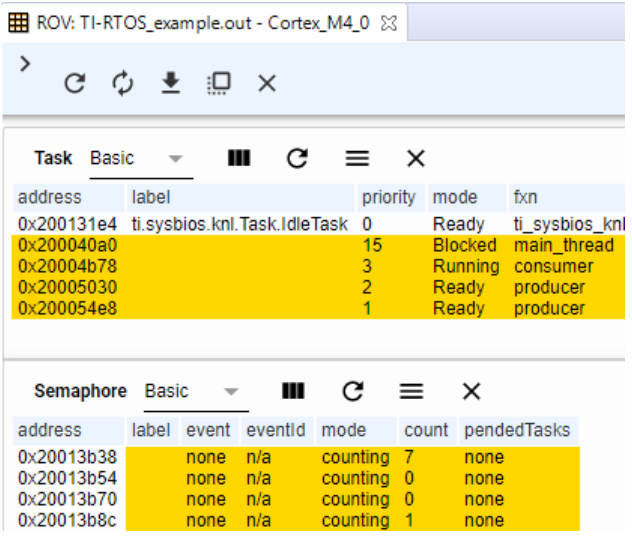

Figure 6: Runtime Object View when the breakpoint at line 57 is reached.

threads can be inspected by using the dropdown box in the Task window, see Figure 7.

- Both producer threads are ready to run.
- The semaphores are apparently shown in the following order: `semEmpty`, `semFilled`, `semMutualExclusive`, and `semPrintf`.
- There are seven empty places in the buffer. `sem_post(&semEmpty)` has not been executed yet.
- There are zero filled places in the buffer. `sem_wait(&semFilled)` has already executed.
- `semMutualExclusive` is zero, so the buffer is not available for another thread.
- `semPrintf` is one, so the `printf` function is available to be used by one thread.

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

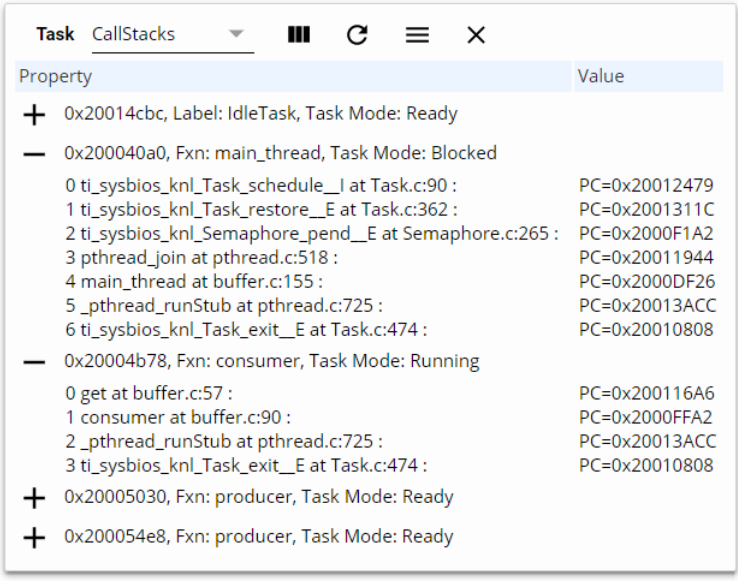

Figure 7: The call stacks of the threads can be viewed in the Runtime Object View.

5.2 The code of the consumer seems inefficient:

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

Adjust the consumer code as discussed above. Compile the program and run it with priorities 3, 2, and 1. Explain why the program stalls. Use the Runtime Object View to help you understand what is going on.

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

5.3 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 first *nine* kroketten are consumed. Explain why one frikandel is consumed followed by a kroket etc. Explain why there are nine kroketten still consumed after the kroket producer has stopped. Only eight of them fit in the buffer!

Is the behavior of the semaphore correct in a real-time environment?

Open the file `release.cfg` in the project `tirtos_builds_CC3220S_LAUNCHXL_release_ccs` and change the line:

```
Semaphore.supportsPriority = false;
```

into:

```
Semaphore.supportsPriority = true;
```

Recompile the program and run it again with the same priorities: Consumer = 1, Frikandel Producer = 2 and Kroket producer = 3. Discuss the output, which is surprisingly unchanged. Is the behavior of the semaphore correct in a real-time environment? You may want to peek at Assignment 5.4.

5.4 As you have seen in Assignment 5.3 the behavior of the semaphore is *not* correct in a real-time environment. The implementation of the POSIX-function `sem_init` provided by TI is *not* correct[6]. Your teacher has written a correct implementation which is called `sem_priority_init`. Download the files `sem_priority_init.c` and `sem_priority_init.h` and add them to the project's directory in your workspace. Include the file `sem_priority_init.h` in the file `buffer.c`. Replace the calls to `sem_init` with calls to `sem_priority_init`.

[6] TI is aware of this problem but has not done anything about it. See: https://e2e.ti.com/support/wireless-connectivity/wifi/f/968/t/761146?RTOS-CC3220-How-to-create-a-priority-aware-semaphore-using-the-POSIX-API-of-TI-RTOS.

Recompile the program and run it again with the same priorities: Consumer = 1, Frikandel Producer = 2 and Kroket producer = 3. Explain the output. Be precise in your explanation. Is the behavior of the semaphore correct in a real-time environment?

5.5 Now try to predict (without running the program) what the flow of execution will be when the priorities are changed to: Consumer = 1, Frikandel Producer = 3 and Kroket producer = 2. Think carefully! Test your prediction by running the program.

Sempahores 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 CC3220S LaunchPad is given in `mqueue.c`.

5.6 Copy the project you created in Assignment 5.1 to `Assignment5.6`. Replace the global variable `buffer` and the semaphores `semMutualExclusive`, `semEmpty` and `semFilled` with a message queue. 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 TI-RTOS because TI's implementation of POSIX's `mqueue` does *not* support prioritized messages.[7]

[7] See: https://e2e.ti.com/support/wireless-connectivity/sub-1-ghz/f/156/t/817039.

The fourth argument given to `mq_send` the `msg_prio` is simply ignored in the implementation.[8]

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 a CC3220S LaunchPad is given in `mutex.c`.

5.7 Copy the project you created in Assignment 5.6 to `Assignment5.7`. Replace the semaphore `semPrintf` with a mutex.

Report Week 1–5

To conclude the first part of the course a report will have to be written. The contents of the report should include the relevant source codes of the weekly assignments 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 codes authenticity.

Document

The document can be written in English or in Dutch and should be about 4-8 page sides long excluding code but including the title page. The relevant and

[8] See: `..\ti\simplelink_cc32xx_sdk_x_xx_xx_xx\source\ti\posix\tirtos\mqueue.c`.

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.
- Briefly explain which algorithm was chosen and how it operates. What advantages and disadvantages does it have?
- Expand on how you implemented this algorithm.
- Describe how you tested the implementation and present the test results.

Delivery

The report shall be delivered as a single PDF document. The file name has the following convention:
`ROS01_studentnumber_surname_studentnumber_surname.pdf`
For example:
`ROS01_0912345_Dijkstra_0954321_Hoare.pdf`

This file must be uploaded into the MS Teams Assignment in the ROS01 team. Only one of the two students working together has to upload the file.