| | | | |
|---|---|---|---|
|  | *Akademia Górniczo-Hutnicza w Krakowie* | | |
| Faculty: ESA | Academic Year 2019/2020 | Year of study | Fields of science: |
| Student name: | **Ioannis Manousaridis** | | |
| Course: | **Real Time Operating Systems in Control Applications** | | |
| Exercise nr: | **11P** | | |
| Exercise subject: | **Exercise RTOS -11P. FreeRTOS – synchronization** | | |
| Lecturer: dr inż. Grzegorz Wróbel | | | |
| Date: 00.00.2020 | | | Grade: |

## 1 Exercise purpose

The purpose of this exercise is to familiarize with the synchronization or protection of resources against simultaneous access of many processes which can be implemented using various tools: inter-task communication, semaphores, mutexes.

## 2 Assumptions / Theory

Having fully independent tasks is rarely possible in practice. In many cases, tasks need to be activated on a particular event, e.g., from an interrupt service routine or from another task requesting a service. In such cases, tasks often need to receive related input, i.e., parameters. Moreover, tasks often need to share hardware resources such as communication interfaces which can only be used by one task at a time, i.e. mutual exclusion, a type of synchronization. Sometimes global variables for such purposes can be used, but implementing thread-safe communication is tricky and a not a safe solution. It may fail if a task-switch strikes at a critical point.

Semaphores are meant to limit access to resources when there are many applications running and solve this problem. Some of the possible ways to do that in FreeRTOS are **the Mutex**, the **binary** semaphores, **Recursive Mutexes** and the **counting semaphores**.

Mutexes are binary semaphores that include a priority inheritance mechanism. Whereas binary semaphores are the better choice for implementing synchronisation (between tasks or between tasks and an interrupt), mutexes are the better choice for implementing simple mutual exclusion (hence ‘MUT’ual ‘EX’clusion).

When used for mutual exclusion the mutex acts like a token that is used to guard a resource. When a task wishes to access the resource it must first obtain (‘take’) the

token. When it has finished with the resource it must ‘give’ the token back – allowing other tasks the opportunity to access the same resource.

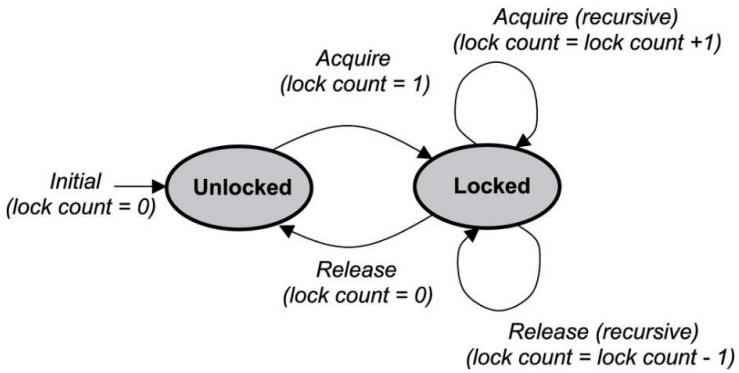


Figure 1: Illustration of the state diagram of Mutex.

Binary semaphore can also be used like a mutex, but binary semaphore doesn't provide priority inversion mechanism. Binary semaphore can be thought as a queue that can only hold one item. The queue can therefore only be empty or full (hence binary). Tasks and interrupts using the queue don’t care what the queue holds – they only want to know if the queue is empty or full. This mechanism can be exploited to synchronise (for example) a task with an interrupt.

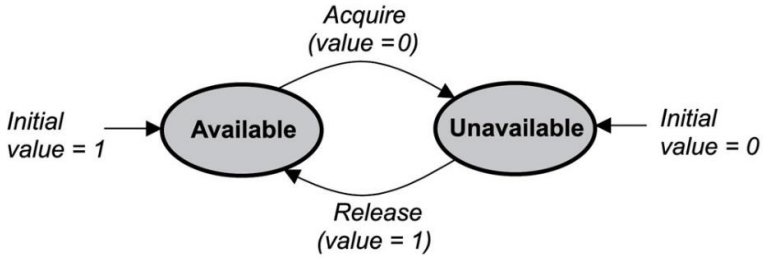


Figure 2: : Illustration of the state diagram of binary semaphores.

Recursive mutexes and counting semaphores will not be explained as they are beyond of the scope of this exercise.

## 3 Description of implementation

For the scope of this exercise the mutex method was used. There two implemantaions in the file.

In the first implementation two simple tasks are being created. Periodically they run and access the stdout resource to print on the screen. Both tasks have the same priority and they are executing concurrently.

In the second implementation again two tasks are being created but the difference is that also a gatekeeper is being created with the use of xSemaphoreCreateMutex() function. Both tasks again have the same priority. The difference lies now in the functions of the two tasks. Both tasks before accesing the stdout resource, they check the gatekeeper to see if this resource if free. If the resource is free, they change the state of the gatekeeper and the use the stdout to print on the screen. After finishing, they change the state of the gatekeeper to let know the system that the resource is free again.

In order to check the functionality of the mutex, if the resource is busy and a task ask the gatekeeper to use the source the request will be denied. In this case, a message will be printed on the screen stating which task was denied the stdout resource. It is kind of

contradicted to use the stdout resource when the access is denied but this was done to understand how the program work through to the lack of real equipment.

A scheme of the program can be seen below which explans very simple how the original code works. The scheme is written in pseudocode.

```
// Scheme of the program

int main(){
Input = get_user_input();

if(input == 1){
        Create_task1_simple()
        Create_task2_simple()}
else{
        gatekeeper = create_mutex();
        Create_task1()
        Create_task2()}

vTaskStartScheduler();

print("The scheduler failed. Insufficient FreeRTOS heap memory");
Return 0;}

Task_1_simple(){
        while (1){
        printf("....");
        vTaskDelay(400);
        }
}

Task_2_simple(){
        while (1){
        printf("....");
        vTaskDelay(500);
        }
}

Task1() {
        while (1) {
                if (if_resource_is_free_use_it(gatekeeper) {
                        printf("……");
                        vTaskDelay(250);
                        free_resource(gatekeeper);
                }
        else {printf("I was denied to use the resource");}}
Task2() {
        while (1) {
                if (if_resource_is_free_use_it(gatekeeper) {
                        printf("……");
                        vTaskDelay(400);
                        free_resource(gatekeeper);
```

```
}else { printf("I was denied to use the resource");}}
```

In the following pictures some timeshots are presented from the execution of the program.

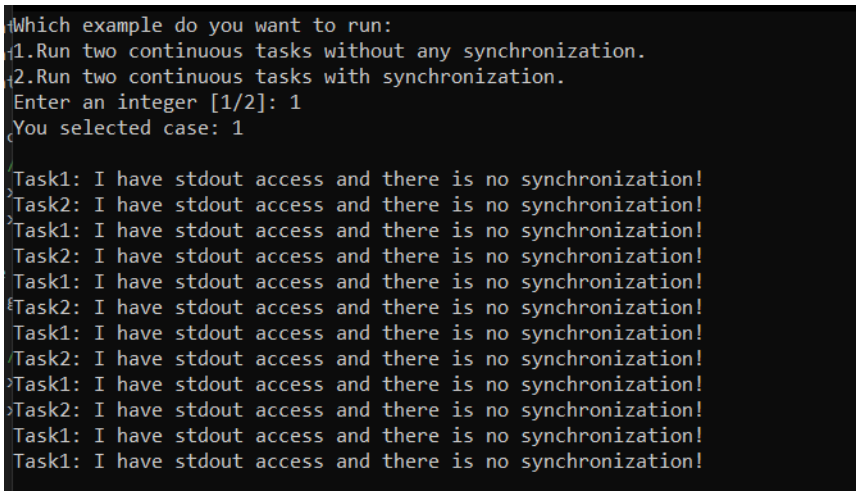


Figure 3:Implementation in which there is no synchronization.

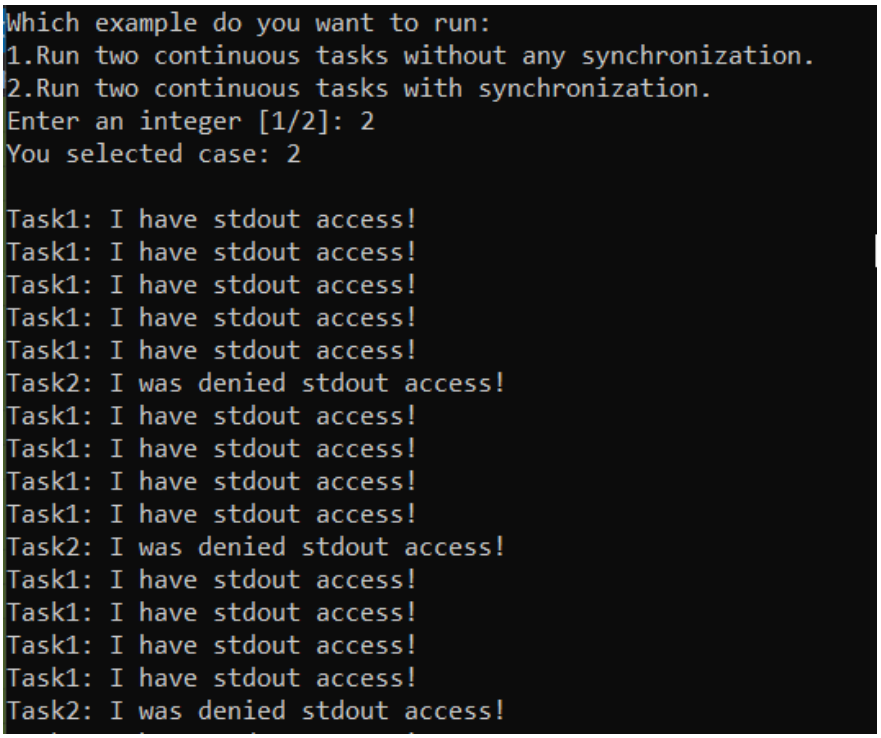


Figure 4:Implementation in which synchronization is done with the Mutex method.

# 4 Conclusions

In general with this lab is clear how important that the multiple concurrent threads within an application must be able to synchronize their execution and coordinate mutually exclusive access to shared resources. The methods of mutex and binary semaphores are easy to understand and implement. Although, they are also tricks. In the second example with the mutual exclusion, they were only two tasks which were requesting access to the screen (stdout). From the figure 4 it can be seen that the task 1 because is faster is executing all the time and the task 2 is always denied the access to the screen. This is an example that the semaphores should be treated carefully otherwise the result will not be the expected one.

Of course, the exchange of the access to one resource can be achieved by other methods as well such as queues, global variables and priorities. However, as it was already mention these techniques just make the program more complicated and they are not the proper way to secure the access to the system resources.