

## 2 - Shared Resources

TSEA81 - Computer Engineering and Real-time Systems

*This document is released - 2014-11-07 - first version*

*Author - Ola Dahl, Andreas Ehliar*

### Lecture - 2 - Shared Resources

This lecture note treats shared resources and mutual exclusion. The lecture note also describes semaphores, and how semaphores can be used to obtain mutual exclusion. The lecture note also gives information related to real-time operating systems, such as usage, in Section ??, and hardware adaptations, in Section ??. The notation [RT] refers to the book *Realtidsprogrammering*<sup>1</sup>.

<sup>1</sup> <https://www.studentlitteratur.se/#31445>

### *Shared resources*

A *shared resource* is a resource which is accessed by one or more tasks. A shared resource can be data, e.g. a common buffer between two communicating tasks. A shared resource can also be hardware, e.g. a display unit, a timer, or an A/D converter.

Section 4.1 in [RT] describes shared resources, using a program example, which is based on the program in the file *odd\_even.c*, used in Assignment 1 - *Introduction, Shared Resources*.

### *Critical Regions*

Section 4.1 in [RT] also introduces the concepts

- *critical region*, which is a sequence in the program code of a task where a shared resource is used,
- *relatively indivisible critical region* which is a critical region where the task is allowed to be interrupted but no other task is allowed to use the shared resource, and
- *absolutely indivisible critical region* which is a critical region where the task is not allowed to be interrupted.

### *Mutual Exclusion*

A common requirement on a real-time program with shared resources is that only one task at a time is allowed access to a shared resource. This can be formulated as a wish to obtain *mutual exclusion* between the tasks that need access to a shared resource.

There are situations where it is not necessary to require that a critical region is indivisible. One such situation occurs when there is a shared resource which can be simultaneously read by several tasks, but only one task at a time is allowed to write to the shared resource. This problem formulation is referred to as *readers-writers*, and is described briefly in Section 4.3 in [RT].

## Mutexes

Semaphores can be used to implement mutual exclusion. Section 5.1.1 in [RT] treats semaphores, and shows how semaphores can be used to ensure mutual exclusion.

A semaphore is presented in Section 5.1.1 as a data type with three operations: one operation for initialisation, and two operations denoted *Wait* and *Signal*. Mutual exclusion using a semaphore is obtained by enclosing every critical region with calls to *Wait* and *Signal*, and by initialising the semaphore to the value one.

Semaphores are available in real-time operating systems, but also in ordinary operating systems like Windows and Linux. The use of semaphores in *Simple\_OS* for the purpose of implementing mutual exclusion is described in Section 5.1.1 in [RT], which also illustrates the use of semaphores by extending the example from Section 4.1.

An example of using semaphores to protect an array can be seen in the following listing:

```
int array[10];

si_semaphore Mutex;
void shift_task(void)
{
    int i, tmp;
    for(i=0; i < 10; i++){
        array[i] = i;
    }
    while(1){
        si_sem_wait(&Mutex);
        tmp=array[0];
        for(i=0; i <= 8 ; i++){
            array[i]=array[i+1];
        }
        array[9] = tmp;
        si_sem_signal(&Mutex);
    }
}
```

```

/* print_task: print task */
void print_task(void)
{
    int sum,i = 0;
    while (1) {
        sum=0;
        si_sem_wait(&Mutex);
        for(i=0; i < 10; i++){
            sum += array[i];
        }
        si_sem_signal(&Mutex);
        console_put_string("Sum: ");
        console_put_hex(sum);
        console_put_string("\n");
        si_wait_n_ms(1000);
    }
}

```

When this example is run under simple os, it will print the same sum at all times. However, if the calls to `si_sem_wait()` and `si_sem_signal()` are removed, it will almost always print different values.

It is possible to implement mutual exclusion without using semaphores. One method, which can be used when there is only one processor, is to disable interrupts during execution in a critical region. This leads to an absolutely indivisible critical region. There are also other methods, which are based on algorithms for implementing mutual exclusion without using semaphores, and which can be used in multiprocessor systems. Mutual exclusion without semaphores is treated in Section 5.1.2 in [RT].

### *Peterson's algorithm*

Section 5.1.2 in [RT] describes how an algorithm called Peterson's algorithm<sup>2</sup> can be used to achieve mutual exclusion. The example uses two tasks, here shown in the following listing:

<sup>2</sup> [http://en.wikipedia.org/wiki/Peterson%27s\\_algorithm](http://en.wikipedia.org/wiki/Peterson%27s_algorithm)

```

/* global variables */
int flag[2];
int turn;

/* task P0 */
void P0(void)
{
    while (1)

```

```

{
    flag[0] = 1;
    turn = 1;
    while (flag[1] && turn == 1)
    {
        /* do nothing */
    }

    CRITICAL REGION

    flag[0] = 0;
}

/* task P1 */
void P1(void)
{
    while (1)
    {
        flag[1] = 1;
        turn = 0;
        while (flag[0] && turn == 0)
        {
            /* do nothing */
        }

        CRITICAL REGION

        flag[1] = 0;
    }
}

```

The following reasoning can be used to show that Peterson's algorithm, as listed above, gives mutual exclusion.

Mutual exclusion is violated, either if both tasks  $P_0$  and  $P_1$  in the above listing enter their critical regions simultaneously, or if one of the tasks enter its critical region while the other task is executing inside its critical region.

Both tasks can enter their critical regions simultaneously, if the conditions in the *while*-statements in the two tasks are false at the same time. If both tasks are executing their *while*-statements, we see, from the listing above, that the variables  $flag[0]$  and  $flag[1]$  both have the value one. And, since the variable  $turn$  always have one of the values one or zero, the conditions  $turn == 1$  and  $turn == 0$  cannot both be false at the same time.

For the case of one task entering its critical region while the other task is executing inside its critical region, suppose that  $P_0$  is executing inside its critical region. Then,  $P_1$  can enter its critical region if the condition in the *while*-statement in  $P_1$  becomes false. This can happen if  $flag[0]$  is zero, or if  $turn$  is one. Since  $P_0$  is executing inside its critical region, we see, from the listing of  $P_0$  above, that  $flag[0]$  is one. It remains to check the value of  $turn$ . From the listing above, we see that  $turn$  is set to one by  $P_0$  and set to zero by  $P_1$ . So if  $turn$  shall have the value one, when  $P_0$  is executing inside its critical region and  $P_1$  is executing its *while*-statement to enter its critical region, the assignment of  $turn$  by  $P_0$  must have happened *after* the assignment of  $turn$  by  $P_1$ . But this could not have happened, since then it would have been impossible for  $P_0$  to enter its critical section, which contradicts our assumption of  $P_0$  executing inside its critical region. Hence, mutual exclusion is obtained.

As described in Section 5.1.2 in [RT], it is also possible to determine that both tasks  $P_0$  and  $P_1$  cannot stay in their *while*-statements for an indefinite time, since the conditions in the *while*-statements cannot be true at the same time.

It is also shown, in Section 5.1.2 in [RT], that if the variable  $turn$  is removed, mutual exclusion is indeed obtained, but it is possible to come to a situation where both tasks execute in their *while*-statements for an indefinite amount of time.

Another modification of Peterson's algorithm is to instead remove the variables  $flag[0]$  and  $flag[1]$ . The resulting algorithm becomes

```
/* global variables */
int turn;

/* task P0 */
void P0(void)
{
    while (1)
    {
        turn = 1;
        while (turn == 1)
        {
            /* do nothing */
        }

        CRITICAL REGION
    }
}

/* task P1 */
```

```

void P1(void)
{
    while (1)
    {
        turn = 0;
        while (turn == 0)
        {
            /* do nothing */
        }

        CRITICAL REGION
    }
}

```

As can be seen in this listing, both tasks cannot enter their critical regions simultaneously. It is also not possible for  $P_1$  to enter its critical region when  $P_0$  executes inside its critical region, since when this is the case, the variable *turn* must have the value zero, otherwise it would not have been possible for  $P_0$  to enter its critical region, giving a contradiction as in the reasoning above for Peterson's algorithm. However, the construction in this algorithm requires that the two tasks enter their critical regions in sequence, and further if one of the tasks spend time in executing code outside its critical region (after the critical region but before the assignment of *turn*), this will prevent the other task from entering its critical region.

Note however that the code above should be seen as pseudo code as it will probably not work correctly out-of-the box in a real system. One problem is that a compiler has a large degree of freedom to reorganize the code during the optimization phase. Special measures has to be taken to ensure that this doesn't happen. In C this can be done by using the *volatile* keyword.

Another issue with the code above is that in a multiprocessor/multicore system with processors that has *out of order execution*, one task may not see the memory accesses in the same order as the other task. This can be solved by using so called memory barriers.

### *Hardware support for Mutual Exclusion*

The method of disabling interrupts for the purpose of achieving mutual exclusion uses the built-in property of the hardware (the processor) to disable and enable interrupts.

As described in Section 5.1.2 in [RT], mutual exclusion can be implemented without using semaphores, and instead using specific algorithms for mutual exclusion, e.g. Peterson's algorithm<sup>3</sup>.

<sup>3</sup> [http://en.wikipedia.org/wiki/Peterson%27s\\_algorithm](http://en.wikipedia.org/wiki/Peterson%27s_algorithm)

Peterson's algorithm, and other algorithms of this kind, assume that a load-operation can be done atomically, i.e. as one instruction, and also that a store-operation can be done atomically. An alternative is to use processor instructions where a load *and* a store in sequence can be done atomically. Using this type of instruction, the actual algorithm for mutual exclusion can be simplified, compared to e.g. Peterson's algorithm. One instruction of this type is called *test-and-set*.

A test-and-set instruction can be used for reading a variable, e.g. a boolean variable indicating if a resource is free or not. The test-and-set instruction enables reading of the variable, then setting the variable to true, and as a result, returning the value read (i.e. the value the variable had before it was set), all as an atomic operation.

Mutual exclusion can be implemented by using a test-and-set instruction, and a boolean variable representing if a shared resource is available or reserved.

A task starting its execution in a critical region executes a test-and-set-instruction. If the value read from the boolean variable indicates that the resource is free, then the task can consider the resource reserved (since the variable was set to true and read without any interrupts). The task can then continue its execution in the critical region.

If the value read instead indicates that the resource is reserved, the task continues, repeatedly, to execute the *test-and-set*-instruction.

A task finishing its execution in a critical region resets the variable, i.e. assigns the value false to the variable, which then marks the resource as free. It is assumed that this assignment is done atomically.

An example of what this looks like in MC68000 assembler can be found in the listing below:

```
; MC68000 example of Test-and-set instruction
loop:
    TAS (a0)    ; Set Z flag according to contents of memory position pointed to
                ; by a0
    BNE loop

    CRITICAL REGION HERE

    CLR.B (a0)  ; Release lock
```

### *More about RTOS*

When using a real-time operating system, there are certain restrictions on how an application program is constructed. For example, the following aspects need to be considered:

- Access to the RTOS API, reflected in the use of include directives and compiler flags. When using *Simple\_OS* as in this course, these aspects are seen in the Makefiles used for compilation and linking, in the course assignments and labs.
- Access to the RTOS library, reflected in the use of directory structures and linker flags. These aspects are also seen in the Makefiles used.
- Structure of main function, e.g. with respect to initialisation of the RTOS, creation of tasks, and starting of the RTOS.
- Structure of tasks, reflected in requirements on C-functions that shall be used as tasks. In *Simple\_OS*, it is required that a task is written as a C-function with no parameters, and with no return value. It is also required that the function contains an infinite loop, i.e. it is not allowed for the function to return.
- Task priorities. In *Simple\_OS* these are set during task creation.
- Usage of the RTOS, reflected in requirements on how to call functions in the RTOS API, and on requirements on how to write interrupt handlers. The *Simple\_OS* API is defined by the file *simple\_os.h* (and the files included from this file). The file *simple\_os.h* is included by all *Simple\_OS* applications.
- Usage of the RTOS for a specific target, e.g. an ARM board. Here, one may find requirements on e.g. compiler vendor, methods for download, and usage and implementation of device drivers.

### *RTOS for different processor architectures*

When using an RTOS, certain parts of the RTOS software need adaptation to the specific hardware architecture used. There are adaptations that must be done for the specific *processor architecture* used, e.g. adaptations for an ARM<sup>4</sup> architecture.

There are also adaptations that must be done for a specific *hardware system*. The hardware system can e.g. be a development board, like the Beagleboard<sup>5</sup> used in the course, or it could be an ASIC System-on-chip, like the ST-Ericsson Thor M7400 Mobile platform<sup>6</sup>.

The adaptations required typically involves aspects like setup and initialisation of interrupts, interrupt handling and implementation of interrupt service routines, task stack creation, task switch, and external device handling, e.g. when implementing drivers for timers, and peripheral units.

<sup>4</sup> <http://arm.com/>

<sup>5</sup> <http://beagleboard.org/>

<sup>6</sup> <http://stericsson.com/products/m7400-thor.jsp>