6 - Real-time operating system

TSEA81 - Computer Engineering and Real-time Systems

This document is released - 2013-12-16 - first version (new homepage)

Author - Ola Dahl

Lecture - 6 - Real-time operating system

This lecture note treats the structure, and parts of the implementation, of a small real-time operating system, using *Simple_OS* for illustration of implementation aspects.

The notation [RT] refers to the book Realtidsprogrammering[1].

[1] https://www.studentlitteratur.se/#31445

Task switch

There may be different reasons for a task switch to occur. A classification can be done, into two categories, depending on if a task switch is initiated from a task or from an interrupt handler.

A task switch initiated from a task occurs e.g. when a task shall wait a specified time interval and performs a call to a function for this purpose (e.g. si_wait_n_ms in *Simple_OS*), or when a task performs a *Wait*-operation on a semaphore having the value zero.

A task switch initiated from an interrupt handler occurs e.g. when a clock interrupt handler finds a task which has been waiting for a specified time interval, and the time interval has expired. This task is then made ready to run, and a task switch is initiated. Another example could be an interrupt handler which communicates with an external unit, e.g. a communication port, and which initiates a task switch when data that are intended for a specific task have arrived.

A task switch is performed by saving the current values of processor registers and program counter on the stack of the running task, and then restoring the corresponding values from the stack of the task which shall resume its execution.

A task switch can be performed when the task to be resumed has been selected. This is decided using the available method for *task scheduling*, e.g. priority based scheduling. The task switch can be thought of as being initiated by calling a function, denoted *Schedule*. This function performs the actual scheduling, e.g. by selecting the task with the highest priority among the tasks which are ready for execution.

A routine which performs the actual task switch is then called. This routine saves and restores information from the stacks of the two tasks that are involved in the task switch. The routine is denoted *Context_Switch* in Chapter 10 in [RT]. The *Context_Switch* routine uses

the instructions of the processor, and is therefore implemented in assembly language, or using in-line assembly code.

A task switch initiated by a task can be described using graphical notation, as in Chapter 10 in [RT]. A task switch initiated by an interrupt handler will be described in more detail in Lecture *Embedded Systems*.

An illustration of the situation before the task switch is intiated is shown in Figure **??**, where two tasks are shown, and one of these tasks is executing.

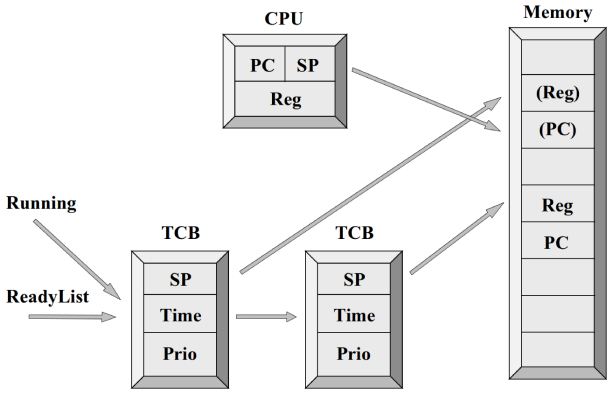

Figure 1: Two tasks, stored in the list of ready-to-run tasks. The task marked Running is executing.

Assume that the task switch is initiated by a call to a function resulting in the calling task waiting for a specified time, such as *si_wait_n_ms* in *Simple_OS*. As a first step, the TCB of the calling task, i.e. task currently running, is removed from the list of ready-to-run tasks and placed in a list of tasks waiting for time to expire. This situation is illustrated in Figure **??**, where the two lists are denoted *ReadyList* and *TimeList*, and where the runnning task is marked by the word *Running*.

As a next step, the context of the running task is saved on the stack of the running task. This means that a value of the program counter, indicating the instruction from which the task shall resume its execution later, shall be stored on the stack, and also that the current values of the processor registers shall be stored on the stack. An illustration of the situation after this has been done is shown in Figure **??**.

As can be seen in Figure **??**, the stack pointer in the CPU refers to the last stored element on the stack, while the stack pointer in the TCB refers to another location. The reason for this is that the stack pointer in the TCB has not yet been updated, and it still refers to the position in memory where the context of the task was saved, either during a previous task switch, or initially, when the task was created.

By setting the stack pointer in the TCB of the running task to refer

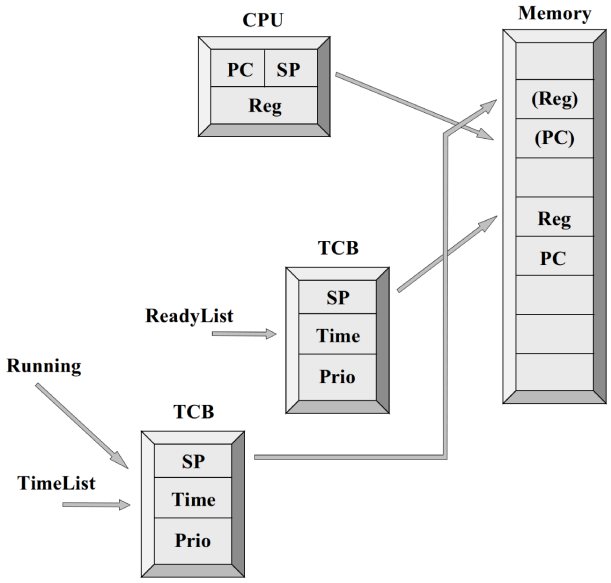

Figure 2: Two tasks, where the task marked Running is executing. The task control block of the executing task has been moved from ReadyList to TimeList, as a preparation for a task switch.

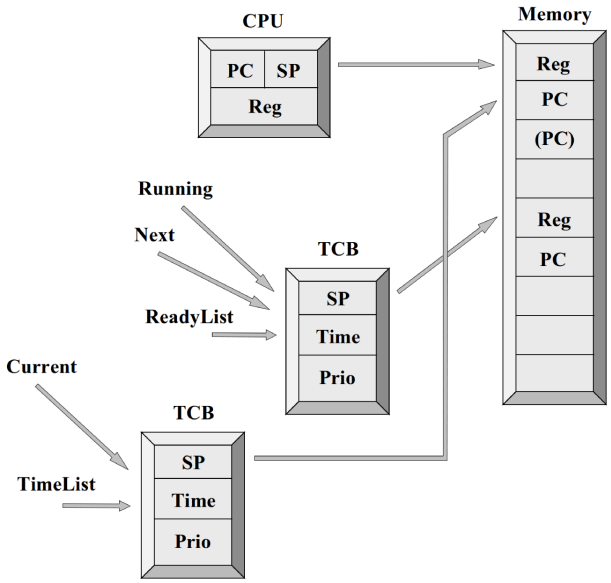

Figure 3: Two tasks, where the task marked Current is executing. Context for the running task has been saved, on the running task's stack.

to the same location as the CPU stack pointer, now pointing to the last stored element on the stack of the (still) running task, and after that, locating the TCB of the task to be resumed, the context switch can be performed. This is done by first setting the CPU stack pointer to the same value as the stack pointer in the TCB of the task to be resumed, giving a situation as shown in Figure **??**, and then restoring values from this stack. When, as a last step, the program counter is restored, the task to be resumed restarts (or starts, if it is the first time it executes) its execution, a situation which is illustrated in Figure **??**.

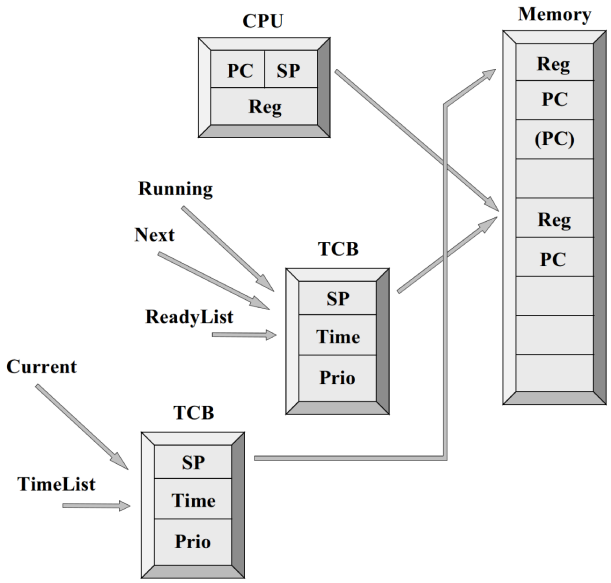

Figure 4: Two tasks, where the task marked Current is still executing. The task switch has been prepared, by setting the CPU stack pointer to refer to the saved context of the task marked Running, stored on the stack of the task marked Running.

Task switch implementation

In *Simple_OS*, a task switch initiated by a task starts with the task calling the function *schedule*. This function is implemented in *schedule.c*, as

```
/* schedule: perform priority based scheduling */
void schedule(void)
{
    /* task id for the running task */
    int task_id_running;

    /* task id for the task in ready list with
       highest priority */
    int task_id_highest_prio;
```

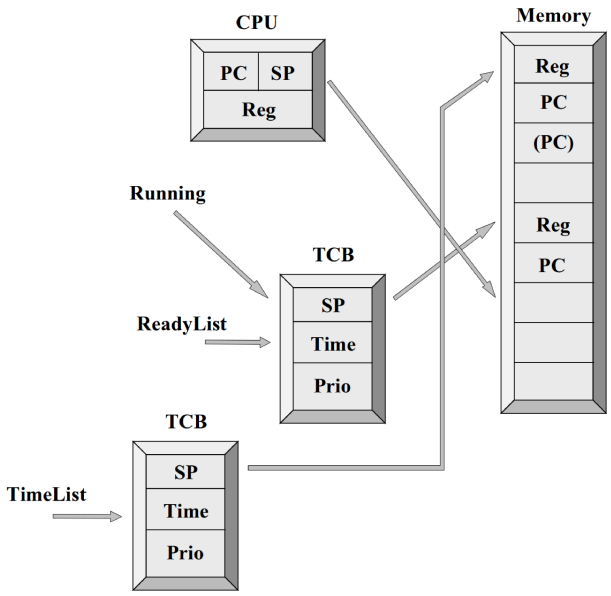

Figure 5: The task marked Running is now executing. The execution was started after the context of the task was restored, by restoring also the saved value of the program counter.

```
    /* get task id for the running task */
    task_id_running = task_get_task_id_running();

    /* get task id for task in ready list with
       highest priority */
    task_id_highest_prio =
        ready_list_get_task_id_highest_prio();

    /* check if a task switch shall be performed */
    if (task_id_highest_prio != task_id_running)
    {
        /* perform task switch */
        task_switch(task_id_running, task_id_highest_prio);
    }
    else
    {
        return; /* no task switch */
    }
}
```

The function *schedule* performs priority based scheduling, and is described in Section 15.2.3 in [RT].

The task switch, when running *Simple_OS* inside Linux or Windows, as is done in the Assignments in the course, is implemented as

```
void task_switch(int task_id_old, int task_id_new)
{
```

```
/* a pointer to the old stack pointer */
mem_address *old_stack_pointer;
/* the new stack pointer */
mem_address new_stack_pointer;

/* pointers to TCB for the two tasks */
task_control_block *old_tcb_ref;
task_control_block *new_tcb_ref;

/* get references to the TCBs */
old_tcb_ref = tcb_storage_get_tcb_ref(task_id_old);
new_tcb_ref = tcb_storage_get_tcb_ref(task_id_new);

/* set pointer to old stack pointer */
old_stack_pointer = &old_tcb_ref->stack_pointer;
/* set new stack pointer */
new_stack_pointer = new_tcb_ref->stack_pointer;

/* set Task_Id_Running to task id of new task */
Task_Id_Running = task_id_new;

/* do the task switch on the host */
context_switch(old_stack_pointer, new_stack_pointer);
```

Here we note that *context_switch* is implemented in assembly.

A task switch on a target, where *Simple_OS* runs as the operating system, as will be done in *Lab 1 - Lift in an Embedded System*, involves additional mechanisms, such as the use of *software interrupt* instead of a function call for starting a task switch initiated by a task, and also using architecture dependent mechanisms for handling context switches initiated by an interrupt handler. These mechanisms are described in more detail in Lecture *Embedded Systems*.

Clock interrupts

A clock interrupt is a periodic interrupt, which is used in a real-time kernel for the purpose of time handling. A typical task for the interrupt handler which is activated when a clock interrupt occurs, is to traverse the list of tasks waiting a specified time interval, decrement the TCB field which indicates the remaining waiting time, while at the same time checking which of these fields that are decremented to the value zero.

The tasks with TCBs where the time field is decremented to zero shall finish their waiting, and are therefore made ready for execution.

A task switch is then initiated. When priority based scheduling is used, this means that if any of the tasks which were made ready for execution have a higher priority than the currently running task, i.e. the task which was interrupted by the clock interrupt, a task switch actually occurs.

When using *Simple_OS* together with Linux or Windows, which is the way *Simple_OS* is used in the assignments in this course, a clock interrupt is simulated using mechanisms available in Linux or Windows. When using Linux, a mechanism referred to as *signals* is used, and when using Windows, a mechanism referred to as multimedia interrupts is used.

When using *Simple_OS* as an independent real-time operating system, in an embedded system or on a PC, a real, periodic, interrupt is used. The frequency of the interrupt is set during start-up.

In *Simple_OS*, a clock interrupt leads to a call of the function *tick_handler_function* in the file *tick_handler.c*. This function traverses the list of tasks waiting for specified time intervals and decrements, in each TCB, the field where the remaining waiting time is stored. The tasks where this field is decremented to zero are then made ready for execution, by placing the tasks in the list of ready-to-run tasks. A task switch is then intiated, by a call to the *schedule* function.

Clock interrupts, and related aspects of time handling, e.g. functions for waiting a specified time interval or until a specified time instant, are described in Sections 15.3.1 and 15.4 in [RT].

Semaphores

A general implementation of semaphores can be done by first defining a data structure representing a semaphore. A data structure of this kind is referred to as a Semaphore Control Block, which is abbreviated SCB. The data structure contains two fields. One of the fields represents the value of the semaphore, and the other field is a list of TCBs for the tasks waiting on the semahore.

Three semaphore operations are needed: one operation for initialisation, a *Wait*-operation and a *Signal*-operation.

Section 10.6 in [RT] describes a general semaphore implementation. The semaphore data structure is described, and general descriptions of the operations are given.

There are situations where the actual semaphore implementation affects the execution of a real-time program. A situation of this kind occurs when a task with high priority repeatedly tries to reserve a shared resource which is protected by a semaphore, and where a task with low priority becomes waiting for the semaphore during a time interval when the task with high priority has reservered the resource.

Section 10.6.1 in [RT] describes how the execution in this situation is affected by the actual semaphore implementation.

One should note that the implementations described in Section 10.6.1 affect a specific situation, and that they do not affect other properties of the semaphores, e.g. that they can be used to ensure mutual exclusion.

An example of literature where different types of semaphores are discussed is the article The Well-Tempered Semaphore: Theme With Variations[2], by Kenneth A. Reek.

[2] http://www.cs.rit.edu/~kar/papers/wts/paper.pdf

In *Simple_OS*, semaphores are implemented using an SCB defined in *si_semaphore.h*, as

```
typedef struct
{
    /* the list of waiting processes */
    int wait_list[WAIT_LIST_SIZE];
    /* semaphore value */
    int counter;
} si_semaphore;
```

An operation for initialisation is defined by the function *si_sem_init*, implemented as

```
/* si_sem_init: intialisation of semaphore sem */
void si_sem_init(si_semaphore *sem, int init_val)
{
    wait_list_reset(sem->wait_list, WAIT_LIST_SIZE);
    sem->counter = init_val;
}
```

The function *si_sem_wait* implements a *Wait*-operation, as

```
/* si_sem_wait: wait operation on semaphore sem */
void si_sem_wait(si_semaphore *sem)
{
    /* task id */
    int task_id;
    /* disable interrupts */
    DISABLE_INTERRUPTS;

    /* check counter */
    if (sem->counter > 0)
    {
        /* decrement */
        sem->counter--;
    }
    else
```

```
    {
        /* get task_id of running task */
        task_id = task_get_task_id_running();
        /* remove it from ready list */
        ready_list_remove(task_id);
        /* insert it into the semaphore waiting list */
        wait_list_insert(
            sem->wait_list, WAIT_LIST_SIZE, task_id);
        /* call schedule */
        schedule();
    }
    /* enable interrupts */
    ENABLE_INTERRUPTS;
}
```

and the function *si_sem_signal* implements a *Signal*-operation, as

```
/* si_sem_signal: signal operation on semaphore sem */
void si_sem_signal(si_semaphore *sem)
{
    /* task id */
    int task_id;

    /* disable interrupts */
    DISABLE_INTERRUPTS;

    /* check if tasks are waiting */
    if (!wait_list_is_empty(
            sem->wait_list, WAIT_LIST_SIZE))
    {
        /* get task_id with highest priority */
        task_id = wait_list_remove_highest_prio(
            sem->wait_list, WAIT_LIST_SIZE);
        /* make this task ready to run */
        ready_list_insert(task_id);
        /* call schedule */
        schedule();
    }
    else
    {
        /* increment counter */
        sem->counter++;
    }
    /* enable interrupts */
    ENABLE_INTERRUPTS;
```

}

Semaphores with time-out

Sometimes it is desirable to limit the maximal waiting time when a task waits on a semaphore. One reason could be that the task which performs the *Wait*-operation is not allowed to wait more than a specified amount of time, e.g. due to safety requirements. A limitation of the waiting time can be achieved, using semaphores with *time-out*, where the maximum waiting time can be given as an extra parameter in the function implementing the *Wait*-operation. The parameter denotes the maximum waiting time expressed e.g. in milliseconds.

A general implementation of semaphores with time-out is described in Section 10.6.2 in [RT].

Condition variables

A general implementation of condition variables can be performed, starting with a data structure for representation of a condition variable. The data structure, which is referred to as Event Control Block (abbreviated ECB), contains two fields.

One field is a reference to the semaphore which is associated with the condition variable, and the other field is a list of TCBs for tasks waiting on the condition variable.

Three operations are needed for a condition variable: one operation for initialisation, an *Await*-operation and a *Cause*-operation.

Section 10.7 in [RT] describes a general implementation of condition variables. The data structure for condition variables is described, and general descriptions for the operations are given.

Condition variables are implemented in *Simple_OS*, in the files *si_condvar.h* and *si_condvar.c*.

Message passing

Functionality for message passing is implemented in *Simple_OS*. There are functions for sending and receiving messages. A message is sent by calling the function *si_message_send*. The task identity of the receiving task is given as a parameter to *si_message_send*.

A message is sent as an array of bytes, which is copied from the sending task to the receiving task. The message is stored in a *message buffer*, which is associated with the receiving task. When a receiving task calls *si_message_receive*, it copies a message from its message buffer to an array which is given as a parameter to *si_message_receive*.

If the message buffer for a task is empty when the task calls *si_message_receive*, the task blocks.

If the message buffer for a receiving task is full when a sending task calls *si_message_receive* with the task id of the receiving task as parameter, the sending task blocks.

The functions *si_message_send* and *si_message_receive* are implemented in the source file *si_message.c*. The message buffer functionality is implemented in *tcb_message.c*. The data structure for a message buffer can be seen in the file *tcb_message.h*.

Assignment 5 - Modification of a Real-time Kernel

In Assignment 5 (2013 version), the source code of a real-time kernel is modified. The modification is done by adding functionality for fair scheduling to *Simple_OS*.

Case study: Three different semaphore examples

```
Example 1:

P1 (highest priority)     P2
wait 10ms                 si_sem_wait(&M1);
si_sem_wait(&M1);         do_work();    // Takes about 50 ms to run
do_work();                si_sem_signal(&M1);
si_sem_signal(&M1);       wait 1000 ms


Example 2:

P1 (highest priority)     P2
si_sem_wait(&M1);         si_sem_wait(&M1);
wait 10ms                 do_work();
do_work();                si_sem_signal(&M1);
si_sem_signal(&M1);       wait 1000 ms
do_work();
si_sem_wait(&M1);
do_work();
si_sem_signal(&M1);
wait 1000 ms


Example 3:

P1 (highest priority):    P2:
```

```
si_sem_wait(&M1)          si_sem_wait(&M2);
wait 10 ms;               wait 10 ms;
si_sem_wait(&M2)          si_sem_wait(&M1);
do_work()                 do_work();
si_sem_signal(&M2);       si_sem_signal(&M1);
si_sem_signal(&M1);       si_sem_signal(&M2);
```