5 - Real-time kernel

TSEA81 - Computer Engineering and Real-time Systems

This document is released - 2013-12-16

Author - Ola Dahl

Lecture - 5 - Real-time kernel

This lecture note starts with a description of Assignment *4 - Lift with Message Passing*

The main part of the lecture note is dedicated to describing the structure, and parts of the implementation, of a real-time kernel, using *Simple_OS* for illustration of implementation aspects.

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

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

Assignment 4 - Lift with Message Passing

The Assignment *4 - Lift with Message Passing* builds on the previous Assignment *3 - Lift with Monitors*, where a lift simulation was implemented, using *monitors*.

In Assignment *4 - Lift with Message Passing* the implementation is instead done using *message passing*.

Some comments regarding Assignment *4 - Lift with Message Passing* in comparison with Assignment *3 - Lift with Monitors*, are given as follows:

- The external requirements are kept, meaning that the lift shall move as before, passengers shall travel from one randomly selected floor to another, also randomly selected floor, and passengers shall wait for the lift, as well as wait inside the lift until they reach their destinations.
- The main contents in the data type used for the lift is kept. The main difference is that the semaphore and the condition variable are now not needed, and are therefore removed.
- The lift itself was previously defined as variable, on file scope, implemented by a global pointer referring to a struct, which constituted the lift data. Now, this is not needed. Instead, the lift can (and should!) be declared *inside* the task that uses the lift data. This idea is also used in Lecture *4 - Monitors, Message passing*, in the listing of the task *clock_task*, where it can be seen that the current time is now not a global variable, but instead declared inside *clock_task*.

Tasks

Task execution

A running task uses a *stack*. The stack is used as a temporary storage area during execution, but also as a storage area for saved information when the task is not executing. When the task executes, the stack stores information, such as information needed during function calls, e.g. local variables, parameters, parameter addresses, and return address information.

The stack is also used during a *task switch*, when a task that shall suspend its execution saves information on its stack, and where this information is re-used when the task resumes its execution again, at a later time instant.

It assumed here, and also in in [RT], that a stack grows towards *lower* memory addresses.

When a task is running, the program counter in the CPU of course changes, as execution proceeds. For the purpose of this lecture, and also with reference to Chapter 10 in [RT], a simple model of a CPU will be used. The model contains a program counter register, here denoted *PC*, a set of processor registers, here denoted *Reg*, and a stack pointer register, here denoted *SP*. A running task, using the CPU modeled in this way, and referring to its stack in memory via the stack pointer register, is shown in Figure **??**.

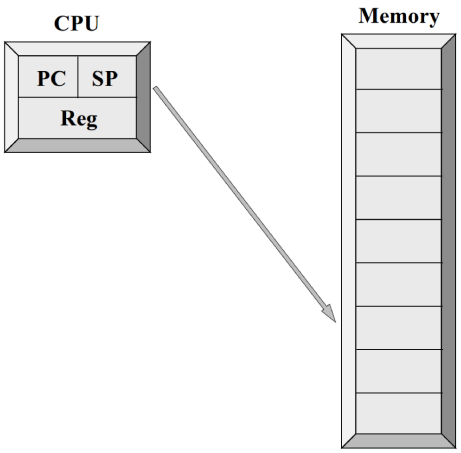

Figure 1: A running task, with the CPU stack pointer accessing memory.

Saving and restoring context

The top elements of a stack are used during a task switch. The routine which performs the task switch therefore needs knowledge about the organization of these data, i.e. how they are stored on the stack. Hence, the top elements of the stack, for a task which is not

executing, must be organized according to a given convention.

Chapter 10 i [RT] uses a convention where the top elements of the stack consist of saved register values, and a saved value of the program counter. The program counter is stored on a higher address than the register values. This means, considering a stack that grows towards lower addresses, that if *pop* and *push* instructions are used when saving data on a stack, the program counter is saved before the registers are saved. As a consequence, when restoring data, as is done when starting and restarting a task, this means that the CPU registers are restored before the program counter is restored. This is required since otherwise the task would start its execution before the registers were restored.

The stack layout used is shown in Figure **??**.

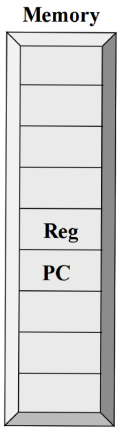

Figure 2: The layout of the stack, for a task which is not running. The stack contains saved values of the program counter, and of the processor registers.

Data structures for tasks

Task control blocks

A data-area, here denoted *TCB* (which stands for Task Control Block) is used for storage of task related information. This storage area is used as a complement to the stack. Typically, more persistent data, such as task priority, is stored in the TCB.

In Chapter 10 in [RT], a TCB with a minimal amount of information is used. The TCB contains a *reference to the stack of the task*, the *time that the task shall wait* when it is waiting a specified time interval, and the *priority* of the task.

A task control block, as used in Chapter 10 in [RT], is shown in Figure **??**.

A TCB which belongs to a certain task is connected to the task stack, via the stack reference stored in the TCB. This connection is established when a task is created. A TCB for a task, referring to

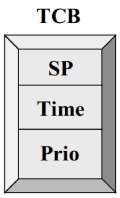

Figure 3: A task control block, with fields for stack pointer, time, and priority.

the task's stack using the stack pointer field in the TCB, is shown in Figure **??**.

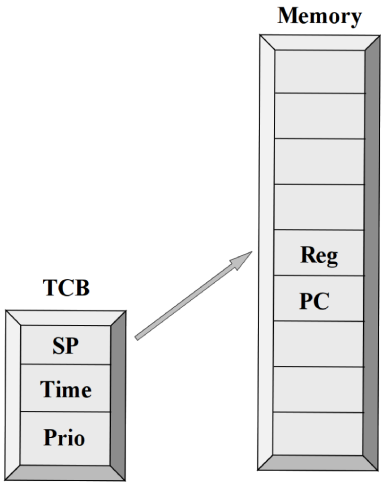

Figure 4: A task control block for a task which is not running. The stack pointer in the task control block refers to the saved context on the task's stack.

Task lists

TCBs can be stored in lists. These lists are referred to as *lists of tasks*. There can e.g. be a list of tasks which are ready to run, and a list of tasks which are waiting for specified time intervals.

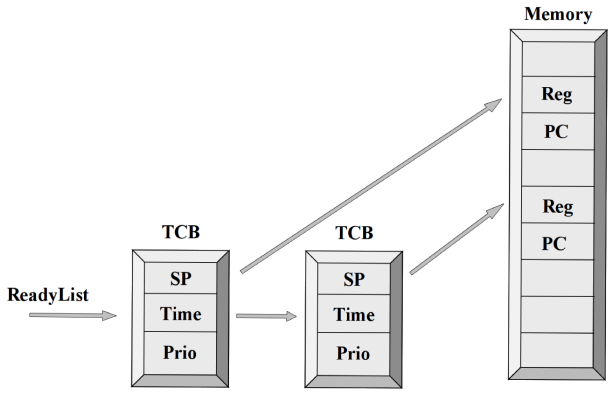

Figure 5: Task control blocks for two tasks, neither of them running, stored in a list of ready-to-run tasks.

Implementation

The task control block is implemented in *tcb.h*, as

```
/* type definition for a task control block */
typedef struct
{
    /* reference to the stack */
    mem_address stack_pointer;
    /* identity */
    int task_id;
    /* validity flag */
    int valid;
    /* waiting time in ticks */
    int wait_ticks;
    /* priority */
    int priority;
} task_control_block;
```

For comparison purposes, one can study the data structure in Linux where information associated with a task is stored. This data structure is denoted *process descriptor*. Source code defining a process descriptor in Linux can be found in the Linux source code[2].

[2] http://lxr.linux.no/linux+v3.6.6/include/linux/sched.h#L1234

Handling of TCBs in *Simple_OS*, and functionality for representing lists of tasks as lists of task identities, are described in Section 15.2.2 in [RT].

In *Simple_OS*, there is *one* list of TCBs. Each created task has its TCB in this list. The list is defined in *tcb_storage.c*, as

```
/* the list of TCBs for all created tasks */
static task_control_block TCB_List[TCB_LIST_SIZE];
```

Lists of tasks, which in the graphical layout shown e.g. in Figure **??** and in Chapter 10 in [RT] are shown as lists of TCBs, are *implemented* as *lists of task identities*. As an example, the list of ready-to-run tasks is defined in *ready_list.c*, as

```
/* size of ready-list */
#define READY_LIST_SIZE TCB_LIST_SIZE

/* the list of ready-to-run tasks */
static int Ready_List[READY_LIST_SIZE];
```

Task creation

A task can be created by first creating a TCB, or by obtaining a free TCB e.g. from a list of free TCBs. In *Simple_OS*, a task creation starts

with initialisation of a TCB, and then finding a free place in the TCB list, described in Section **??**.

Then, the stack is prepared, resulting in a stack layout as in Figure **??**, and the stack pointer in the TCB is set to refer to the top of the prepared stack, which establishes the connection between the TCB and the task stack.

A task in *Simple_OS* is created by calling the function *si_task_create*. The address of the function which shall be made into a task is given as argument to *si_task_create*. The address of the first element of the stack, and the task priority, are also given as arguments to *si_task_create*. In this way, the *si_task_create* function achieves information which is necessary for the task creation.

A graphical illustration of a newly created, but not yet started, task, is shown in Figure **??**.

Task start

Before a task starts its execution, it is placed in the list of ready-to-run tasks. The situation is now as in Figure **??**.

The task is started by restoring values from its stack. This means that values are copied, from the task stack to registers in the CPU. For this to be feasible, the task stack must be located. This can be done using the stack reference field in the TCB. The value stored in this field is then copied to the stack pointer of the CPU, which makes the CPU stack pointer refer to the task stack.

The available processor instructions for working with the stack are then used, to perform the actual copying of register values and a program counter value, from the stack to the corresponding registers in the CPU.

The task starts its execution when the program counter is copied from the task stack to the CPU. An illustration of this moment is shown in Figure **??**, where it can be seen that the stack pointer of the CPU now refers to the memory location after the stored value of the program counter. Assuming that this stored value, during preparation of the stack contents, was selected as the start address of the task, it can be seen that Figure **??** illustrates a situation where the task is executing its very first instruction. Note that Figure **??** could also illustrate a situation later on, where the task executes any of its instructions, and for the moment is not using its stack for temporary storage, e.g. of local variables in functions.

In *Simple_OS*, task start is implemented by a function *task_start*, in *task.c*. This function is implemented as

```
void task_start(int task_id)
```

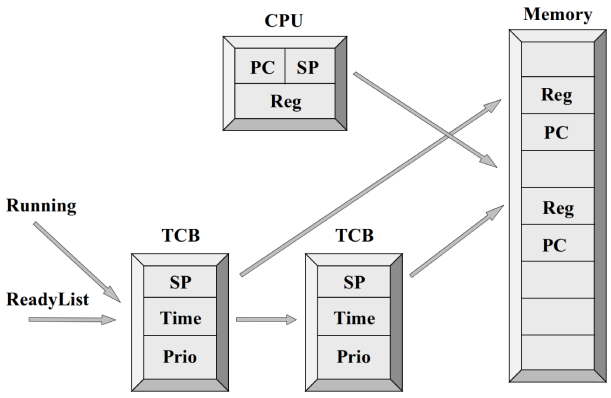

Figure 6: Two tasks, both ready to run. Context for the the task marked Running has been restored, and the task has started its execution.

```
{
    /* a pointer to a TCB */
    task_control_block *tcb_ref;

    /* get a pointer to the TCB associated with
       task identity task_id */
    tcb_ref = tcb_storage_get_tcb_ref(task_id);

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

    /* restore context for the task with this
       TCB */
    context_restore(tcb_ref->stack_pointer);
}
```

The function *task_start* starts a task, given a task identity, by copying register values and a program counter value from the stack of the task, to the corresponding registers in the CPU.

The function *task_start* is called by the function *si_kernel_start*, which is implemented in the file *si_kernel.c* as

```
/* si_kernel_start: start real-time kernel */
void si_kernel_start(void)
{
    /* task_id for task with highest priority */
    int task_id_highest;

    /* the kernel is running */
    Kernel_Running = 1;

    /* print version information */
    console_put_string(SIMPLE_OS_VERSION_STRING);
```

```
    /* get task_id for task with highest priority */
    task_id_highest = ready_list_get_task_id_highest_prio();

    /* start the task with highest priority */
    task_start(task_id_highest);
}
```

and which is called in a real-time program during start-up, when *Simple_OS* shall be started.

The function *si_kernel_start* is described in Section 15.5 in [RT].

Here, it can be noted that *context_restore*, which is called by *task_start* as shown above, is written in assembly.