REAL-TIME MICRO KERNEL PROJECT

Group 50 –
Seyed Saeid Hosseini
Zeynep Temizel

Halmstad University
Computer System Engineering II
Hazem Ali
Ove Andersson
Youshra Alkabani

21.05.2021

INTRODUCTION:

In this course, we received a crude code and completed it respected to the pseudocode that given has been given to us. We use **IAR embedded workbench for arm** environment and C programming language to write our code. The aim of this course, and of course the project, was to understand how real-time operating systems are built, tested, and developed.

This project based on three categories:

1. Task Administration
2. Inter-Process Communication
3. Timing Functions

We used linked lists to create

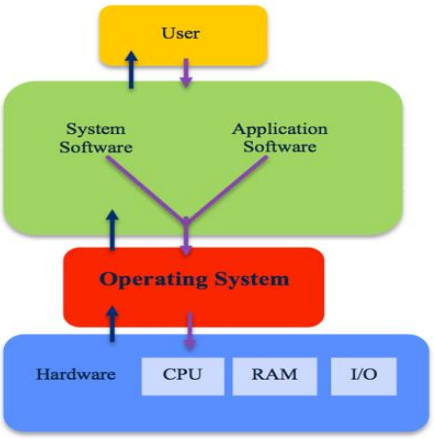
1. Ready List: carries tasks that are ready for execution
2. Waiting List: carries tasks blocked from execution and waiting for some resources
3. Timer List: carries tasks sleeping for a specific duration.

We also used linked lists for mailboxes, which is related to Inter-Process Communication. Every other thing that needed to create an RTMK given to us. Assembly codes, list structures, testing files, etc.

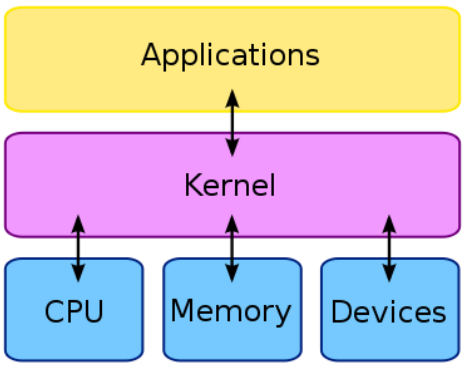
At the end of each category, we tested our code with testing files. If there is a problem, it was easy to spot thanks to that files.

OPERATION OF THE IMPLEMENTED RTMK:

The most important thing that makes an operating system necessary is to ensure that so many processes and programs are run by the processor quite efficiently at the same time, and to manage the memory and processor resource sharing between these processes.
Because the processor can process one command at a time as a principle of operation. But it's quite illogical and useless to do all this work in rows.



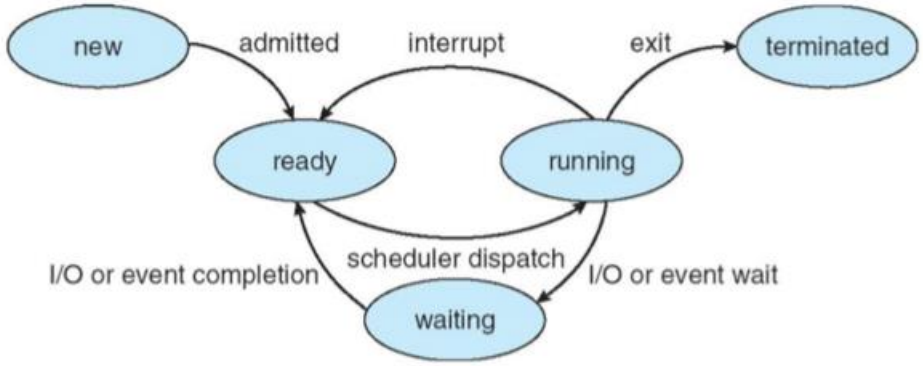
The processor needs a clock signal when processing commands. This signal is required to handle each command. Naturally, the higher the frequency of the clock, the faster you will process.
But being faster after a certain speed doesn't do much good. What will really work is to be able to run program functions or programs that perform different operations at the same time.
Therefore, the main purpose of operating systems is to divide the signal frequency of this processor for different programs and make shared use between programs. In this way, it's as if there's more than one processor, but it divides the actual processor speed into that many processors.



A kernel is a name given to the interface that provides the connection between hardware and software on the computer. It establishes communication between basic processes in the system while the operating system is running, and is involved in executing and sharing many other processes, such as processor management, memory management, and I/O (entry-output) operations.

It can be said that kernel has complete control over the system. To be able to connect the software to the computer's hardware, we must use the kernel.

TASK STATES, SCHEDULING AND CONTEXT SWITCH



T

There are 3 states of the task: Ready, running, and waiting/blocked.

When a task is created, it moves to the ready state. It waiting to be assigned to a processor by the OS.

Then it moves to the running state. It runs until:

· It comes to an end and is terminated.
· I/O or event interrupt – it moves to a waiting state.
· The deadline is reached.

As we can see in the diagram, the task should come to the ready state before it goes running no matter what. If the task is interrupted, it goes to the ready state, THEN the running state.

When it comes to scheduling, we must know about deadlines. Deadline is the last time the task should be completed and the task with the lowest deadline has the highest priority. When a task is running, and another one comes with a lower deadline, the running task will be sent to the waiting/blocking state and it waits to be executed again. At the same time, the lower deadline task moves to the running state from the ready state. Like this, the lowest deadline is always executed first.

As the CPU passes to another process, the system must record the state of the old process and then load the saved state of the new process after this registration. The environment of a process is kept in Process Control Block (PCB), and time is spent on the change of order. And that's overhead. Because during the exchange, the processor does not do a job that directly works for the user. The time spent may vary depending on hardware support. The processor can deal with a single job instantly (CPU), according to the operating system property; it is permissible to feel as if more than one job is being run at the same time. The structure that instantly runs a single process and makes it look like it is doing more than one job; can be called multitasking.

DETAILED LOOK ON THE CODE

In the introduction, we mentioned about three categories. In this section we will be processing them. Once we finished writing our code respect to the pseudocode, we tested it with the test file that was provided to us.

1. TASK ADMINISTRATION

In this section there are 4 functions:

- init_kernel()
- create_task()
- run()
- terminate()

init_kernel() is used for initialization of the kernel and creating ready, waiting, and timer lists. It also creates the “idle task” .

```
void idle_task() {     /** DONE **/
        while (1);
}
```

As you can see it is a while loop. We only call this function when there is no task to execute for keeping the program running.

create_task() is taking deadline and the task body as parameters. Data structures are created here and if the kernel is already running, rescheduling and context switching are on the table.

run() simply runs the program waiting in the ready list (as we mentioned, with the lowest deadline) and updates the list.

terminate() function called when the running task should be terminated, this function calls and we delete the task from the list and set the next task.

This section was about managing the task between states and creating some needed lists.

2. INTER-PROCESS COMMUNICATION

We were really challenged at this part. Our helping functions started not to work, and we must rewrite some of them, etc.

Inter-process communication (IPC) is a set of programming interfaces for programs that allow programmers to coordinate different processes so that they can work simultaneously on an operating system and exchange information with each other. This allows a program to meet the needs of many users at the same time.

In this section there are 7 functions:

- create_mailbox()
- remove_mailbox()
- send_wait()
- receive_wait()
- send_no_wait()
- receive_no_wait()
- wait()

create_mailbox() takes 2 integers as parameters. One is for setting the max number of messages for the mailbox and the other one is the size of the message. We allocate the memory for the mailbox and initialize the mailbox structure and return it to the mailbox.

remove_mailbox() checks if the mailbox is empty and if it is empty clears its data.

send_wait() sends a message to the specified mailbox in the parameter. If there is a receiving task waiting for a Message on the specified Mailbox, and it will deliver it and the receiving task will be moved to the ready list. Otherwise, if there is not a receiving task waiting for a Message on the specified Mailbox, the sending task will be blocked. In both cases, a new task schedule is done and possibly a context switch. **receive_wait()** receives those messages and applies the same operation.

send_no_wait() will send a message to the specified Mailbox. The sending task will continue execution after the call. When the Mailbox is full, the oldest message will be overwritten. The send_no_wait call will imply new scheduling and possibly a context switch. Note: send_wait and send_no_wait Messages shall not be mixed in the same Mailbox. And again **receive_no_wait()** will attempt to receive a message from the specified mailbox.

wait() function will send the running task that calls it to sleep.

3. TIMER FUNCTIONS

This was the easiest part for us!

In this section there are 5 functions:

- set_ticks()
- ticks()
- deadline()
- set_deadline()
- timerInt()

Ticks were given to us as a global value as a tick counter.

set_ticks() taking uint as a parameter and set the ticks to that number.

ticks() returns to tick counter.

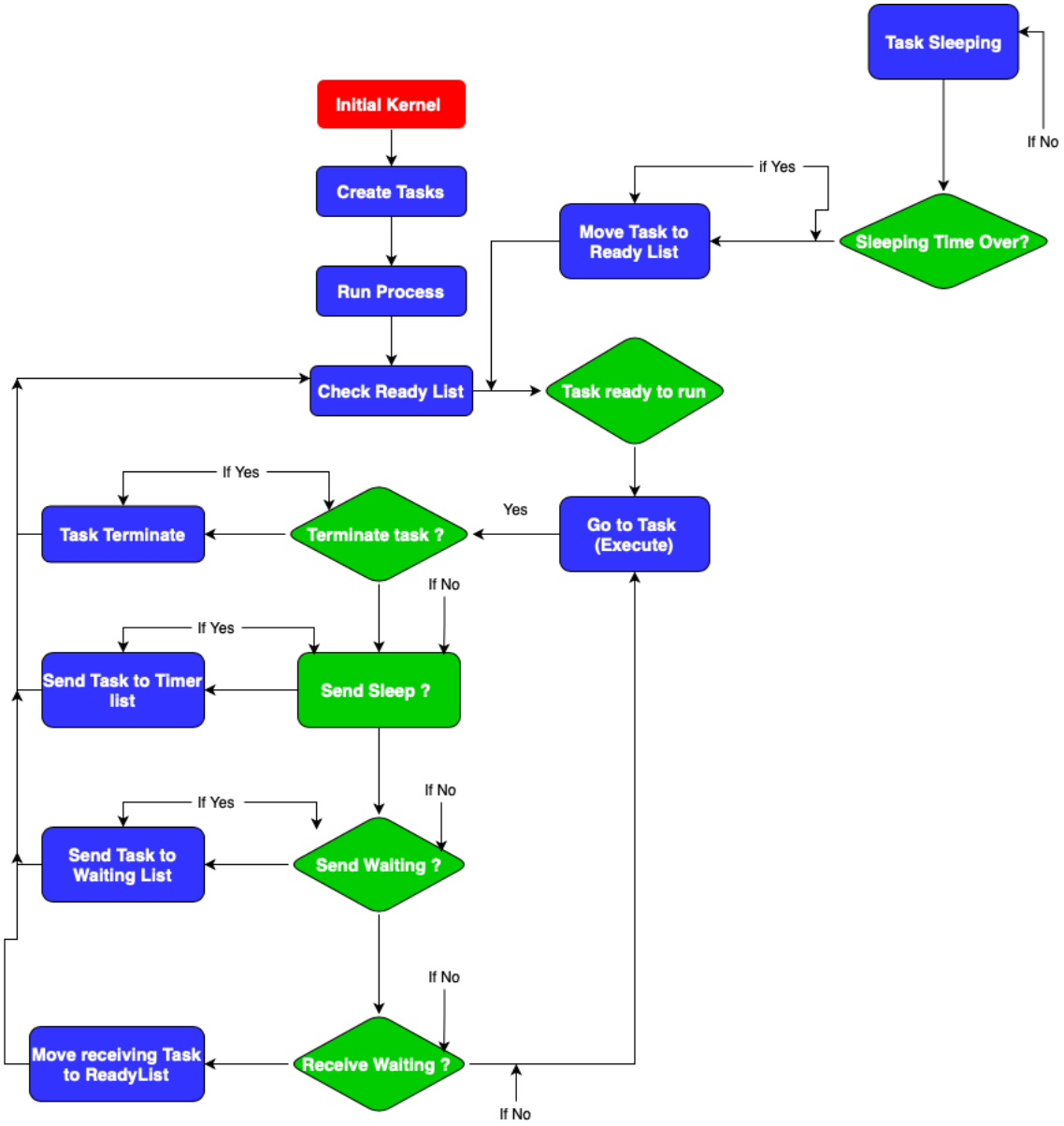
deadline() returns the deadline of the current task.

set_deadline() function sets the deadline then makes some rescheduling and updating in the ready list.

timerInt() this function is called by an Interrupt Service Routine invoked every tick. It checks the timer list for the tasks that ready for execution. It also checks the waiting list for tasks that have expired their deadlines, moves these to the ready list.

This section was about managing time.

FLOWCHART



We can see the whole process in the flowchart above. First, we initial the kernel then we create the task and start the process. While running, we always check the ready list if there is any task ready to run with a lower deadline. While the task is executed it can be terminated or go to the waiting list. At the same time the tasks in the timer list are sleeping we constantly check if they are ready to move to the ready list.

ASSISTING LIBRARIES IMPLEMENTED

void addTask(list **name, listobj *task)
When we want to add a new task to a list, we scan the list and find the right position for it. For example, we check the deadline of the objects in the list. If the inserted task's deadline is bigger than the tail's deadline we use this function **void addTaskAfter(list **name, listobj *task1, listobj *task2)** to add it after.

void taskdelete(list **name, listobj *obj)
taskdelete function takes listobj that we want to remove and the name of the list that we want it to be removed from. We specifically check if the list is empty, has only one obj, the object is the head or tail, or just a normal obj and treats it according to.

bool memberCheck(list ** name, listobj *obj)
memberCheck function scans the list and search for a specific listobj. If it founds, function returns true.

void listSorting(list **name)
It sorts the list by scanning it.

void mailboxMsgAdd(mailbox *mBox, msg* msg)
We add a msg to mailbox. We check if it is empty.
void mailboxMsgDelete(mailbox* mBox, msg *msg)
We delete a msg from mailbox. We check if it is empty or the msg is in the head/tail. So we can update the mailbox correctly.

msg* mailboxOldDelete(mailbox *mBox)
It creates a temp msg and then deletes it from the mailbox. Then returns to it. We used it to delete the oldest msg in the structure if the mailbox is full.

TESTING

The Task administration testing file was a success. We didn't spend much time on the first section. Then, we come to the second part. We had to rewrite some of our helping codes and %90 of our time spent on this section. We could not pass our testing file right away. There was always a problem with receive_no_wait() as dummy handler said to us, we had problems creating msg delete and add functions, etc. At some point, we just gave up and started to write our code from the scratch and tried to make up our helping codes to section two. As we tried to follow a different path for our helping codes in this re-exam, it took more time to understand and implement everything all over again. But when we finished and finally saw "Alles gut" the happiness was worth it. Section three was rather easier than sections two and one. We had no problem with it.

We also tested our helping function. We changed them when we started over and had to make sure that all functions are compatible with all sections.