Jorix kernel: real-time scheduling

Joris Huizer Kwie Min Wong

May 16, 2007

1 Introduction

As a specialized part of the kernel, we implemented two real-time scheduling algorithms: `RM` (rate monotonic) and `EDF` (earliest deadline first)

A real-time scheduling algorithm is defined in terms of periods and system times. A real-time task is some program (a process or a thread) which periodically needs to do some work. An example is a task that has to respond to scanner input which comes to the system periodically, and needs to be handled. An important property in this kind of task is a loop of waiting for input and timely responding to input.

The system time is the amount of time a task spends dealing with one cycle. A period of a task has to be known to the scheduler, and it is the maximum system time the task may use.

1.1 Rate Monotonic

The rate monotonic algorithm assigns fixed priorities to tasks. It assigns higher priorities to tasks with shorter periods. When needing to choose to schedule a task, the one with the highest priority available is chosen, but of course a task that was run already in the current period cannot be chosen.

The rate monotonic algorithm forces the scheduling to be preemptive, meaning that processes are interrupted to allow other processes (with higher priorities) to run before continuing the other (with a lower priority).

RM is known to be optimal, in the sence that it is at least as good as any other real-time algorithm using fixed priorities.

The fixed priorities nature of RM means it can be implemented in hardware.

It is not trivial to see whether a given set of tasks can be scheduled by RM. A known guarentee about RM is: for n processes, the worst-case schedule bound is

$$W_n = n \times (2^{\frac{1}{n}} - 1)$$

This means, if the total sum of CPU usage of all processes exceeds W_n, there is no guarentee RM will be able to schedule the processes correctly (without missing deadlines).

This property was proven by Liu and Layland in 1973.

1.2 Earliest Deadline First

The earliest deadline first algorithm does not fix priorities; instead it looks at the deadlines of tasks; the task nearest to it's deadline gets executed.

EDF is known to be optimal, in the sense that, if a task set is schedulable, it will be schedulable by EDF.

For process i, C_i is the service time and T_i is the period of the process.

$$U = \sum_{i}^{N} \frac{C_i}{T_i} \leq 1$$

If U is less then 1 the process set is schedulable.

An advantage of EDF over other dynamic priority algorithms is, that the processor can be fully utilized, without many context switches.

2 Scheduling Environment

Real-time scheduling algorithms have the need to know when to "start" a task and when to "stop" it. They have to know when a periodic task has been finished.

In our kernel this has been implemented in the following way; The user program does some initialisation, executing some "open" system calls. Then it executes a "read" (or, a "start") system call, does it's calculations, and executes a "write" (or, a "stop") system call. After the "write" has finished, another "read" is executed (so this is some kind of loop). Finally the task exits completely (the program stops).

As you can see, it's a very simple (and some what restrictive) environment, but it suits the purpose to test the different scheduling algorithms. "Reads"

```
struct io_data
{
  int fd;
  int count;
};
```

Figure 1: File descriptor

and "writes" don't really change files, as our kernel does not have a real file system implementation.

2.1 Process Creation

During process creation (the `fork` system call), the process gets stored in the list of processes, which is used by the scheduler. This happens by calling `scheduleProcess`.

A call to `schedulePriorityChanged` tells whether rescheduling is needed; If there is a need, a special `startProcess` call will tell the scheduler to actually reschedule. This function returns a value which is to be stored in the `TSS` back link field.

2.2 File Descriptor

We have an extremely simple file descriptor, as shown in figure 1.

In Unix-like environments, a file is represented by an integer - this is the `fd` field. The only other field is the `count`, which is a number containing the number of reads can be executed on the file (the number of bytes).

The number of writes on a file is not limited so this number is irrelevant for written files.

2.3 Open System Call

Like on linux, our "open" system call requires these register settings:

- `EAX` is set to 5.
- `EBX` is set to a string, the name of the file to open.

- `ECX` is set to the corresponding flags, which tell whether to open a file for reading or writing.
- `EDX` is set to the mode (the permissions to use when creating a new file)

In our implementation, for a “read”:

- `EBX` *must* be set to “/tmp/sensor”.
- `ECX` *must* have the standard `O_RDONLY` value set (value 0).

In our implementation, for a “write”:

- `EBX` *must* be set to “/tmp/out”.
- `ECX` *must* have the standard `O_WRONLY` value set (value 1).

2.4 Read System Call

Like on linux, our “read” system call requires these register settings:

- `EAX` is set to 3.
- `EBX` is set to a file descriptor, given by the `open` system call.
- `ECX` is set to a user buffer which is to be filled.
- `EDX` is set to size of the user buffer.

As there isn’t really a file to read data from, the buffer gets filled with random data.

The purpose of executing the `read` system call in our kernel is to tell the scheduler to “start” a cycle of the task loop. After setting up the buffer, a scheduling function `restart_process` is called.

2.5 Write System Call

Like on linux, our “write” system call requires these register settings:

- `EAX` is set to 4.
- `EBX` is set to a file descriptor, given by the `open` system call.

- `ECX` is set to a user buffer which is to be written to the file.
- `EDX` is set to size of the user buffer.

As there isn't really a file to read data from, the buffer is completely ignored.

The purpose of executing the `write` system call in our kernel is to tell the scheduler to "stop" a cycle of the task loop. After setting up the buffer, a scheduling function `halt_cycle_process` is called.

3 Scheduling Interface

We wrote two real-time scheduler implementations, with a common interface. There are two kinds of functions:

- timer oriented functions
- process oriented functions

3.1 Timer Oriented Functions

The timer oriented functions give an interface to execute a function after a certain amount of "ticks" (the duration of one "tick" is dependent on initialization of the timer hardware) The following functions have been implemented:

- `addTimer`
- `deleteTimer`
- `resetTimer` (a macro function)
- `timer_handler`

The timer functions are used by both scheduling algorithms.

```
typedef void (*timerHandler)(void *ptr);
typedef struct timer
{
  unsigned long int delay;
  timerHandler handler;
  void *ptr;
  char isNew;
} timer;
```

Figure 2: Timer descriptor

3.1.1 Timer Description

The timer structure as used by timer oriented functions is defined as figure 2 shows.

delay	is the number of ticks till execution
handler	is the callback function to call when delay becomes zero
ptr	is the pointer to give as an argument to the callback function
isNew	is a marker field: it should be 0 except when the timer gets “reset”, when it becomes 1.

3.1.2 addTimer

The `addTimer` function adds the given timer to the array of timers, if there is still place left. It makes sure the array stays sorted on delay, such that the timer with the shortest delay is the first timer in the array.

3.1.3 deleteTimer

The `deleteTimer` function removes the given timer from the array of timer, if it can be found. Like the `addTimer` function it ensures the array stays sorted.

3.1.4 resetTimer

The `resetTimer` macro function gets two arguments: the timer to "reset" and the delay to use (this may be a completely different delay than the previously used value) It sets the delay, marks the timer to indicate it's a resetted timer, and adds the timer using `addTimer`.

3.1.5 timer_handler

The `timer_handler` function increments the global `timer_ticks` variable, which hold the current number amount of "ticks" between kernel boot and now.

It decrements every timer delay in the array, and calls a timer function, when it's delay reaches zero. It is careful not to decrement resetted timers, and it will mark resetted timers as normal timers for the next call to `timer_handler`.

A periodically called service is required to reinstall itself each time it's function gets called, and it can use the `resetTimer` function for this. Implemented like this, there isn't anything special about a periodically called service.

After decrementing timers, any old timer (of which the delay went to zero) is removed from the list.

3.2 Process Oriented Functions

The process oriented functions tell the scheduler to restart a process or to choose a new process to schedule.

- `scheduleProcess`
- `scheduler`
- `restart_process`
- `halt_cycle_process`
- `startProcess`
- `deleteProcess`
- `schedulePriorityChanged`

Processes can have the following states:

running	when the task is active.
died	when the task has finished completely.
waiting_io	when the task is waiting for I/O (currently not used).
waiting_sys	when the task is waiting to be scheduled for the current period.
waiting_next	when the task is waiting to be scheduled for the next period.

3.2.1 scheduleProcess

The `scheduleProcess` function is called to correctly add a given process to the array of processes. Details are dependant on the scheduling implementation.

3.2.2 scheduler

The `scheduler` function is called to choose a new process to be scheduled. It chooses the new process, which has to have a state `running` (avoiding choosing the idle task if possible). Then it updates the `TSS` descriptor to hold the process address. Finaly it returns the `TSS` selector to the caller.

3.2.3 restart_process

The `restart_process` function is called to tell the scheduler the next iteration of the user task loop starts. The scheduler does bookkeeping dependent on the implementation.

3.2.4 halt_cycle_process

The `halt_cycle_process` function is called to tell the scheduler the current user task loop ends. The scheduler will select the next task to run, depending on the implementation.

3.2.5 startProcess

The `startProcess` function is to be called directly after a `fork` created a new process, if `schedulePriorityChanged` returned true, or in case only the idle task is running.

3.2.6 deleteProcess

The `deleteProcess` function gets called when the `exit` system call gets executed. It does whatever is necessary to schedule another process and removes the given process from the process array.

If the last process has executed an `exit`, the number of deadline misses is printed per process.

3.2.7 schedulePriorityChanged

The function `schedulePriorityChanged` indicates after process creation whether rescheduling is required.

4 Rate monotonic implementation

The rate monotonic implementation works as follows:

- when a task cycle is started, the `startTick` field of the process is updated to the current `timer_ticks` value.
- when a task cycle is stopped, the `state` field is set to `waiting_next`. The next process gets selected and scheduled.
- when the period of the process, has elapsed, the timer callback, `rescheduleHandler`, gets called. It performs the following actions:
    1. it checks whether the process is in the `waiting_next` state. It will set the `state` field to `waiting_sys` if this is the case. If it is not, it means the process has not done all its work yet and it missed its deadline. A miss will be remembered.
    2. If the process is the first process, it gets restarted immediately (meaning it stops the previously active process).
- the timer for the next period is setup.

When a process gets selected, a check is done whether it missed its deadline.

4.1 scheduleProcess

The function `scheduleProcess` takes a new process which has to be taken into account by the scheduling algorithm.

It will first cleanup the process array, such that any `NULL` entry is removed, and then adds the given process to the array.

It indicates that a priority changed if the process is to be scheduled for execution immediately. This is true if it currently has the highest priority.

The function ensures the process array stays sorted, as the rest of the algorithm assumes it is sorted.

4.2 restart_process

The function `restart_process` just updates the `startTick` field of the process structure to the current `timer_ticks` value.

4.3 selectNextProcess

The function `selectNextProcess` loops over the available processes. If it finds a process with the `waiting_sys` state, it returns that process. Otherwise it will return a `NULL` pointer. The next `scheduler` call will in this case select the idle task.

4.4 request_schedule

The function `request_schedule` calls `selectNextProcess` to select the next process with the highest priority, which has not run in the current period.

Then it calls the `scheduler()` function and stores the returned TSS descriptor in the `int_tss` back link.

4.5 halt_cycle_process

The function `halt_cycle_process` takes a process pointer as an argument. It sets the `state` field to `waiting_next`, which indicates that the process won't run untill it's new period begins. Then the function `request_schedule` is called to schedule another process for execution.

5 Earliest Deadline First Implementation

The earliest deadline first implementation works as follows.

- when a task cycle is started, a number of things happen.

  1. The process struct has a special field `periodCount` which indicates in which period number the process started. When the value is incorrect, a miss is remembered.
  2. The `startTick` field of the process is updated to the current `timer_ticks` value.
  3. The `periodCount` is updated.

- when a task cycle is stopped, the `state` field is set to `waiting_next`. The next process gets selected and scheduled. The `periodCount` is checked, and misses are remembered. The `periodCount` is updated if necessary.

- when the period of the process, has elapsed, the timer callback, `rescheduleHandler`, gets called. It will schedule a process if currently the idle task is active; otherwise, no scheduling will happen, meaning our EDF implementation is not preemptive.

  The timer will always get re-installed.

5.1 getProcessDeadline

The function `getProcessDeadline` takes a process pointer and returns its current deadline.

The deadline of $process_i$ is calculated by:

$$deadline_i = period_i - \tau \ \mathbf{mod}\ period_i$$

Here τ is the current time, the current amount of ticks.

5.2 selectProcessByEDF

The function `selectProcessByEDF` finds the process that has to be scheduled, according to the EDF algorithm.

It will loop over the array of processes, and for every process that can be selected, any process in the `waiting_sys` or the `running` state, it calculates the priority, using `getProcessDeadline`. Finally it returns the process that has the earliest deadline.

5.3 scheduleProcess

The function `scheduleProcess` takes a new process which has to be taken into account by the scheduling algorithm.

The scheduling fields of process structure are initialised. The state is set to `waiting_sys` and a timer gets installed, such that at the end of the period, the process may be restarted.

A check is performed by calling `selectPocessByEDF()` and the boolean value `priorityChanged` is updated. If the selected process is not the current one, a priority change occurred.

5.4 restart_process

The function `restart_process` does a number of things:

1. It checks whether the process missed a deadline, and updates the miss counter if this is the case
2. It updates the `startTick` field of the process structure to the current `timer_ticks` value.
3. It updates the `periodCount` field to indicate a new period has started.

5.5 request_schedule

In the silent implementation, this just calls `selectProcessByEDF`. In the verbose version it will print output if no new process was selected, as none of the periods could be restarted.

This is the case if all the processes have done the work for the current period, and no new period has started.

5.6 halt_cycle_process

The function `halt_cycle_process` takes a process pointer as an argument. It does a number of things:

1. It checks whether the process missed a deadline, and updates the miss counter if this is the case
2. It updates the state to `waiting_next`.
3. It calls `request_schedule` and then calls `scheduler` to schedule a new process.
4. It updates the `periodCount` field to the current value, using the following formula:
$$periodCount_i = \frac{\tau}{period_i}$$
Here τ is the current time, the current amount of ticks.

6 Test Results

We will describe some test results, a process is described as T1 : (1,4) where T1 is just a name, the 1 means it's (maximum) amount of time the process is using CPU, and 4 is it's period; Both numbers are expressed in 'ticks' (which are fired by the hardware timer).

Because of a low CPU speed in the emulated environment of Bochs, all periods and system times are around hundred ticks, or above this number.

We found a lot of printing output causes delays - even such delays that timer problems occur - the CPU getting a new timer interrupt while the previous timer interrupt is still finishing; This results in a GPF (general protection fault, interrupt 13). Therefor we removed all output prints for the actual tests.

Dead line misses are reported in this way:

$$x/y/z$$

x is the number of misses for the first process, y the number for the second process, and z is the number for the third process.

6.1 RM Test Results

Some tests results of RM with three processes are shown in table 1.

P1	P2	P3	load	missed
T1(100,200)	T2(100,600)	T3(100,1000)	0.767	0/0/0
T1(100,300)	T2(100,400)	T3(100,600)	0.750	0/0/0
T1(100,300)	T2(100,400)	T3(500,3000)	0.750	0/0/0
T1(100,300)	T2(100,400)	T3(500,1700)	0.877	0/0/0
T1(100,400)	T2(200,600)	T3(500,1700)	0.877	0/0/0
T1(100,400)	T2(200,600)	T3(500,1800)	0.861	0/0/0
T1(100,400)	T2(200,600)	T3(500,1900)	0.846	0/0/0
T1(100,400)	T2(200,600)	T3(500,2000)	0.833	0/0/0
T1(100,400)	T2(170,400)	T3(100,800)	0.800	0/0/0
T1(190,400)	T2(100,400)	T3(100,800)	0.850	0/0/0
T1(100,300)	T2(100,400)	T3(300,800)	0.958	0/0/12
T1(100,300)	T2(100,400)	T3(300,800)	0.958	0/0/12
T1(200,600)	T2(300,600)	T3(100,700)	0.976	0/0/43
T1(100,600)	T2(200,600)	T3(300,600)	1.000	0/0/50
T1(100,500)	T2(200,600)	T3(300,600)	1.033	0/8/42
T1(100,300)	T2(100,300)	T3(100,400)	0.917	0/0/25
T1(100,300)	T2(100,300)	T3(100,300)	1.000	0/0/50
T1(100,200)	T2(100,300)	T3(100,700)	0.976	0/33/0
T1(100,200)	T2(100,300)	T3(100,600)	1.000	0/33/0
T1(100,1000)	T2(300,1000)	T3(500,1000)	0.900	0/0/0
T1(190,1000)	T2(300,1000)	T3(500,1000)	0.990	0/0/1
T1(200,400)	T2(190,1000)	T3(300,1000)	0.990	0/0/1
T1(100,200)	T2(190,1000)	T3(300,1000)	0.990	0/0/0

Table 1: RM test results

The 0/0/1 entries must be caused by starting delays, as a `fork` and an `execve` call take some time.

In general, it is hard to see whether a set of processes will be correctly scheduled. In many cases the theoretical W_n is exceeded but no misses occur.

6.2 EDF test results

Some tests results of EDF with three processes are shown in table 2.

P1	P2	P3	load	missed
T1(100,200)	T2(100,600)	T3(100,1000)	0.767	0/0/0
T1(100,300)	T2(100,400)	T3(100,600)	0.750	0/0/0
T1(100,300)	T2(100,400)	T3(500,3000)	0.750	0/0/0
T1(100,300)	T2(100,400)	T3(500,1700)	0.877	0/0/0
T1(100,400)	T2(200,600)	T3(500,1700)	0.877	0/0/0
T1(100,400)	T2(200,600)	T3(500,1800)	0.861	0/0/0
T1(100,400)	T2(200,600)	T3(500,1900)	0.846	0/0/0
T1(100,400)	T2(200,600)	T3(500,2000)	0.833	0/0/0
T1(170,400)	T2(100,400)	T3(100,800)	0.800	0/0/0
T1(190,400)	T2(100,400)	T3(100,800)	0.850	0/0/0
T1(100,300)	T2(100,400)	T3(300,800)	0.958	0/0/0
T1(100,300)	T2(100,400)	T3(300,800)	0.958	0/0/0
T1(200,600)	T2(300,600)	T3(100,700)	0.976	0/0/0
T1(100,600)	T2(200,600)	T3(300,600)	1.000	0/0/0
T1(100,500)	T2(200,600)	T3(300,600)	1.033	9/0/3
T1(100,300)	T2(100,300)	T3(100,400)	0.917	0/0/0
T1(100,300)	T2(100,300)	T3(100,300)	1.000	0/0/0
T1(100,200)	T2(100,300)	T3(100,700)	0.976	0/0/0
T1(100,200)	T2(100,300)	T3(100,600)	1.000	0/0/0
T1(100,1000)	T2(300,1000)	T3(500,1000)	0.900	0/0/0
T1(190,1000)	T2(300,1000)	T3(500,1000)	0.990	0/0/0
T1(200,400)	T2(190,1000)	T3(300,1000)	0.990	0/0/0
T1(100,200)	T2(190,1000)	T3(300,1000)	0.990	0/0/0

Table 2: EDF test results

The 0/0/1 entry must be caused by starting delays.

Our EDF implementation has preemption. Preemption is required to solve the following problem: One process has a very long service time (and a very long period), and whenever it gets scheduled, other processes would start missing deadlines, if they wouldn't get rescheduled.

7 Reference

http://www.embedded.com/story/OEG20020221S0089
http://hartik.sssup.it/ lipari/rtos/lucidi/edf.pdf