Mentoring Operating System (MentOS)

Real-Time Scheduler

Enrico Fraccaroli

enrico.fraccaroli@univr.it

Table of Contents

Fields and systemcalls

Introduction

To make possible the usage of real time tasks, a new set of attributes and function has been introduced in MentOS; not only on kernel-side, but also on user-side via the **libc**, to allow to the user to create and execute its own RT programs.
On the following slides there's a presentation on every new element introduced.

Real-Time Tasks Fields

New fields were added in `struct sched_entity` to accomodate the new type of tasks:

```
struct sched_entity {
    ...

    time_t period;      // Expected period of the task
    time_t deadline;    // Absolute deadline
    time_t arrivaltime; // Absolute time of arrival of the task
    bool_t is_periodic; // Determines if it is a periodic task.

    ...
}
```

These fields refers to the basic informations we want for the tasks. N.B.: Special tasks

Real-Time Tasks Fields

These ones are instead used in the schedulability test

```
struct sched_entity {
    ...

    bool_t executed;           // Has already executed
    bool_t is_under_analysis;  // Do we need to analyze the WCET of the process?
    time_t next_period;        // Beginning of next period
    time_t worst_case_exec;    // Worst case execution time
    double utilization_factor; // Processor utilization factor

    ...
}
```

New task's fields are initialized after its creation on standard values by the `__alloc_task` function (see `<mentos>/inc/process/process.c`).

System-calls

The programmer is expected to manipulate the values for the `deadline`, `period`, `arrivaltime`, `is_periodic` and `sched_priority` fields with the given systemcalls:

```
#include <sched.h>

    //set "param" fields values to the task fields
    int res = sched_setparam(pid_t pid, const sched_param_t *param)
    //get task fields values copied in "param" fields
    int res1 = sched_getparam(pid_t pid, const sched_param_t *param)
```

Each one return an integer that could be:

- 1 if syscall had succeded.
- −1 if there was an error.

Struct sched_param_t

The `sched_param_t` structure is implemented as follows:

```c
struct sched_param_t {
    int sched_priority; // Static execution priority.
    time_t period;      // Expected period of the task
    time_t deadline;    // Absolute deadline
    time_t arrivaltime; // Absolute time of arrival of the task
    bool_t is_periodic; // Is task periodic?
}
```

Example of usage

```c
#include <sched.h>
#include <sys/unistd.h>
int main(int argc, char * argv[])
{
    pid_t cpid = getpid(); // Get process pid.
    sched_param_t param;   // Declare schduling parameter struct.

    // Get current parameters (You could print them...).
    sched_getparam(cpid, &param);

    // Change parameters.
    param.period      = 5000;
    param.deadline    = 5000;
    param.is_periodic = true;

    // Set modified parameters.
    sched_setparam(cpid, &param);

    ...
```

Waitperiod

This system call is used to interrupt the task's execution and to wait before being executed again for at least a time frame equal to its `period` field (period must be previously set with `setparam`).

```c
#include <sched.h>

// Returns 0 on normal completion, or a negative number
//  if there was an error.
int sys_waitperiod();
```

In reality `waitperiod()` does a lot more than this, but we'll see that later...

Example of usage

This is an extract of code in wich the `waitperiod()` is used.

```
...

if (waitperiod() == -1) {
    printf("[%s] Error in waitperiod: %s\n", argv[0], strerror(errno));
    break;
}

...
```

In a real period task, the core part of the code would be in a infinite loop, with `waitperiod()` being the last instruction of the cycle.

Schedulability Analysis

Introduction

A fundamental aspect when dealing with real-time tasks is the **schedulability analysis**, that is a test made before adding to the `runqueue` a new RT task.
The whole purpose of this test is to make sure that adding a new task on the running set, does not prevents the others to execute withing their own deadline.

In MentOS the schedulability analysis is implemented by executing new periodic tasks the first time using a non periodic scheduling algorithm,and then using a periodic one. Thanks to this we can have an evaluation of the task WCET(Worst Case execution time) and determine if the task is schedulable or not.

Who goes first?

Having periodic and aperiodic tasks running at the same time can be a huge problem for the system. What if an aperiodic task is being executed and this lead a periodic one to miss its deadline?

In MentOS however this is not a problem, in fact the policy of mixed task execution it's the following:

1. Execute all periodic tasks that are scheduled
2. Execute aperiodic tasks if there are no periodic left
3. Repeat

Also, it's important to specify that every time the scheduler have to choose, periodic tasks have priority over aperiodic ones.

Schedulability Test

Testing the program to check if it's schedulable is up to the `waitperiod()` system call, and is done differently for every scheduler algorithm implemented.
We will show the schedulability analysis code specific to an algorithm when we will present it later, with the algorithm itself.

Now a brief view on common parts of `waitperiod()`:

Waitperiod overview

```c
int sys_waitperiod() {
    // Get the current task.
    task_struct * task = runqueue.curr;

    if (task == NULL)
        return -ESRCH; // No such process.
    if (!task->se.is_periodic) {
        return -EPERM; // Operation not permitted.

    // Update the Worst Case Execution Time (WCET).
    time_t wcet = timer_get_ticks() - task->se.exec_start;
    if (task->se.worst_case_exec < wcet)
        task->se.worst_case_exec = wcet;

    // Update the utilization factor.
    task->se.utilization_factor = ((double)task->se.worst_case_exec / (double)task->se.period);

    // If the task is under analysis, we need to test if the
    // process can be placed with the other periodic tasks.
    if (task->se.is_under_analysis) {
        task->se.worst_case_exec  = task->se.sum_exec_runtime;
        bool_t is_not_schedulable = false;

        // ... here starts the algorithms specific code ...
```

Waitperiod overview

If the user process calls `waitperiod()` but `runqueue.curr`, which is the currently running process, is NULL, it returns the error `ESRCH` (`No such process`).

If the user process is not periodic but still calls `waitperiod()`, which is something that is not supposed to happen, the function returns the error `EPERM` (`Operation not permitted`).

The `is_under_analysis` field is used to determine if the process needs to go through the schedulability analysis. It cannot be changed by the user process, and it is automatically set to `true` every time you call the `sched_setparam` function. It is a way of letting the scheduler know that you changed something in the scheduling parameters of the process.

Waitperiod overview

```
#if   defined(SCHEDULER_EDF)
        ...
#elif defined(SCHEDULER_RM)
        ...
#endif
        // If it is not schedulable, we need to tell it to the process.
        if (is_not_schedulable)
            return -ENOTSCHEDULABLE;

        // Otherwise, it is schedulable.
        task->se.is_under_analysis = false;

        // The task has been executed as non-periodic process so that his
        // deadline is not been updated by the scheduling algorithm of
        // periodic tasks. We need to update it manually.
        task->se.next_period = timer_get_ticks();
        task->se.deadline = timer_get_ticks() + task->se.period;
    }
    if (timer_get_ticks() > task->se.deadline)
        pr_warning("%d > %d Missing deadline...\n", timer_get_ticks(), task->se.deadline);

    // Tell the scheduler that we have executed the periodic process.
    task->se.executed = true;
    return 0;
}
```

Waitperiod overview

After the task exit this part of code and it is deemed schedulable, it continues its execution as a proper periodic task and never does this test again.

What if during the task execution another `sched_setparam()` is called?

Scheduler Algorithms

Introduction

MentOS scheduler algorithms have been implemented based on those present in the book:

> *Buttazzo, Giorgio C. Hard real-time computing systems: predictable scheduling algorithms and applications. Vol. 24. Springer Science & Business Media, 2011.*

The algorithms implemented for periodic tasks are:

- Earliest Deadline First (EDF)
- Rate Monotonic (RM)

Also an algorithm for aperiodic tasks has been implemented:

- Aperiodic EDF (AEDF)

Earliest Deadline First

This algorithm is based on the idea of choosing for execution the task with the earliest absolute deadline.
This is an elegant and simple solution when there are tasks with **dynamics arrival-times** and **preemption is allowed**.

EDF example

Here an example of an EDF scheduling:

	J_1	J_2	J_3	J_4	J_5
a_i	0	0	2	3	6
C_i	1	2	2	2	2
d_i	2	5	4	10	9

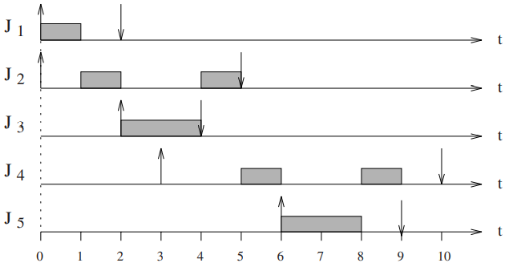

Figure: a_i = arrivaltime, C_i = computation time, d_i = deadline

EDF implementation

Necessary variables are declared and initialized at the beginning of the function.

```
static inline task_struct *scheduler_edf(runqueue_t *runqueue)
{
    // Pointer to the next task to schedule.
    task_struct *next = NULL, *entry;
    // Initialize the nearest "next deadline".
    time_t min = UINT_MAX;

    ...
```

EDF implementation

```
...

// Inter over the runqueue to find the task with the earliest absolute deadline.
list_for_each_decl(it, &runqueue->queue)
{
    // Check if we reached the head of list_head, and skip it.
    if (it == &runqueue->queue)
        continue;

    // Get the current entry.
    entry = list_entry(it, task_struct, run_list);

    // If the entry is not a periodic task, or is a periodic task but under analysis, skip it.
    if (!entry->se.is_periodic || entry->se.is_under_analysis)
        continue;

    ...
```

EDF implementation

```
        ...
        if (entry->se.executed) {
            // If the period for the entry is starting again and it has
            //  already executed, set it as 'executable again'.
            // Deadline and next_period are propagated.
            if (entry->se.next_period <= timer_get_ticks()) {
                entry->se.executed     = false;
                entry->se.deadline    += entry->se.period;
                entry->se.next_period += entry->se.period;
            }
        } else {
            // If the deadline of the process is less than the minimum value,
            //pick it.
            if ((entry->se.deadline < min) && (!entry->se.executed)) {
                next = entry;
                min  = next->se.deadline;
            }
        }
    } // list_for_each_decl(it, &runqueue->queue)
    ...
```

EDF implementation

Here is clearly visible the solution that allow to aperiodic tasks to be executed together with periodic ones.

```
    ...

    // If there are no periodic task to execute, just pick an aperiodic task by using CFS.
    if (next == NULL)
        next = scheduler_cfs(runqueue, true);

    return next;
}
```

EDF schedulability analysis

In EDF a task is deemed schedulable if:

1. given that previous set of running tasks was schedulable;
2. adding the new task still allow $U \leq 1$.

The utilization factor U is computed for a given set of n tasks as follows:

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

where C_i is the *Computation time*, and T_i is the *period*.

EDF schedulability analysis code

As previously shown this is part of `waitperiod()` system call, and is specific to EDF algorithm.

```
#if defined(SCHEDULER_EDF)
    double U = 0;
    // Iterate over the runnqueue.
    list_for_each_decl(it, &runqueue.queue)
    {
        task_struct *entry = list_entry(it, task_struct, run_list);

        // Sum the utilization factor of all periodic tasks.
        if (entry->se.is_periodic)
            U += entry->se.utilization_factor;
    }
    // If U is above 1, then adding the process would make the set of tasks
    // non schedulable.
    if (U > 1)
        is_not_schedulable = true;
```

Rate Monotonic

Rate Monotonic is a fixed priority algorithm. The priority is assigned to every task during its schedulability analysis phase in a way that: **higher the request rate**(i.e.: shorter the period), **higher the priority** for a given task.
Moreover Rate Monotonic is **intrinsically preemptive**: an executing task is preempted by a new arriving one with shorter period.

Rate Monotonic Implementation

Necessary variables are declared and initialized at the beginning of the function.

```
static inline task_struct *scheduler_rm(runqueue_t *runqueue)
{
    // Pointer to the next task to schedule.
    task_struct *next = NULL, *entry;
    // Initialize the nearest "next period".
    time_t min = UINT_MAX;

    ...
```

Rate Monotonic Implementation

```c
...

// Iter over the runqueue to find the task with the shortest period.
list_for_each_decl(it, &runqueue->queue)
{
    // Check if we reached the head of list_head, and skip it.
    if (it == &runqueue->queue)
        continue;

    // Get the current entry.
    entry = list_entry(it, task_struct, run_list);

    // If the entry is not a periodic task, or is a periodic task but under analysis, skip it.
    if (!entry->se.is_periodic || entry->se.is_under_analysis)
        continue;

    ...
```

Rate Monotonic Implementation

```
    ...

    if (entry->se.executed) {
        // If the period for the entry is starting again and it has
        //  already executed, set it as 'executable again'.
        // Deadline and next_period are propagated.
        if (entry->se.next_period <= timer_get_ticks()) {
            entry->se.executed     = false;
            entry->se.deadline    += entry->se.period;
            entry->se.next_period += entry->se.period;
        }
    } else {
        // If the next period of the process is less than the minimum
        // value, pick it.
        if (entry->se.next_period < min) {
            next = entry;
            min  = next->se.next_period;
        }
    }
} // list_for_each_decl(it, &runqueue->queue)

...
```

Rate Monotonic Implementation

```
    ...

    // If there are no periodic task to execute, just pick an aperiodic task by using CFS.
    if (next == NULL)
        next = scheduler_cfs(runqueue, true);

    return next;
}
```

As in EDF, here is clearly visible the solution that allow to aperiodic tasks to be executed together with periodic ones.

RM schedulability analysis

As previusly shown this is part of `waitperiod()` system call, and is specific to RM algorithm.

```
#elif defined(SCHEDULER_RM)
    // Calculating least upper bound of utilization factor.
    // For large amount of processes ULUB asymptotically should
    // reach ln(2).

    double ULUB = (runqueue.num_periodic * (pow(2, (1.0 / runqueue.num_periodic)) - 1));
    double U    = 0;

    list_for_each_decl(it, &runqueue.queue)
    {
        task_struct *entry = list_entry(it, task_struct, run_list);

        // Sum the utilization factor of all periodic tasks.
        if (entry->se.is_periodic)
            U += entry->se.utilization_factor;
    }

    ...
```

RM schedulability analysis

```
...

// If the sum of utilization factor is bounded between ULUB and 1
// we need to calculate the response time analysis for each process.
if (U > 1)
    is_not_schedulable = true;
else if (U <= ULUB)
    is_not_schedulable = false;
else
    is_not_schedulable = __response_time_analysis();
```

If the utilization factor calculated is set between ULUB (least upper bound for U) and 1 (if > 1 means that CPU is over 100% of work load), the response time analysis must be calculated to check if the introduction of the new task is feasable for the schedule of the entire set of tasks.

RM schedulability analysis

Here is shown the function that calculate the response time analysis for a given task:

```
static int __response_time_analysis()
{
    task_struct *previous;
    time_t r, previous_r = 0;

    list_for_each_decl(it, &runqueue.queue) {
        task_struct *entry = list_entry(it, task_struct, run_list);
        // Skip non-periodic processes.
        if (!entry->se.is_periodic) continue;

        // Set r equal to worst case exec because is the first point in
        // time that the task could possibly complete.
        r = entry->se.worst_case_exec;

        // Reset the previous r.
        previous_r = 0;

        ...
```

RM schedulability analysis

```
        ...

        // The analysis can be completed either missing the deadline or
        // reaching a fixed point
        while (r < entry->se.deadline && r != previous_r) {
            previous_r = r;
            r          = entry->se.worst_case_exec;
            list_for_each_decl(it2, &runqueue.queue) {
                previous = list_entry(it2, task_struct, run_list);
                // Skip non-periodic tasks.
                if (!previous->se.is_periodic)
                    continue;
                // Check the interferences of higher priority processes.
                if (previous->se.period < entry->se.period)
                    r += (int)ceil((double)previous_r / (double)previous->se.period) *
                        previous->se.worst_case_exec;
            }
        }
        // Feasibility of scheduler is guaranteed if and only if the
        // response time analysis is lower than deadline.
        if (r > entry->se.deadline)
            return 1;
    }
    return 0;
}
```

Aperiodic EDF

The EDF scheduler for aperiodic tasks is implemented almost like the EDF for periodic tasks, with the difference that this one does not need a schedulability test. Doing so however may bring some task to miss their deadline.

AEDF Implementation

```
static inline task_struct *scheduler_aedf(runqueue_t *runqueue)
{
    // Pointer to the next task to schedule.
    task_struct *next = NULL;
    // Initialize the nearest "next deadline".
    time_t min = UINT_MAX;

    // Iter over the runqueue to find the task with the earliest absolute deadline.
    list_for_each_decl(it, &runqueue->queue) {
        task_struct *entry = list_entry(it, task_struct, run_list);

        //Check if entry is a "special" task i.e. tasks with deadline = 0
        if(entry->se.deadline != 0) {
            //Select the task with minimum absolute deadline
            if ((entry->se.deadline <= min)) {
                next = entry;
                min  = next->se.deadline;
            }
        }
    }

    ...
```

AEDF Implementation

If there are no "real-time" tasks to be executed, pick a normal one:

```
    ...

    // If there are no tasks with deadline != 0 to execute, just pick another task by using CFS.
    if (next == NULL)
        next = scheduler_cfs(runqueue, true);

    return next;
}
```