CS 251/EE 255: Real-Time Embedded Systems
University of California, Riverside

HW Project #2: Task Scheduling and Monitoring

Due: 2/25/2023, Saturday, 11:59 p.m. (Pacific time)

Read the entire handout carefully before starting work. Good luck and happy kernel hacking!

1. Introduction

This homework project is designed to exercise the following topics:

- Accessing task information in the kernel data structures
- Linux scheduler basics
- Timers

Before you start working:

- Read the handout carefully: background, writeup questions and the tips section aim to give you hints about design and implementation.
- Whenever the handout asks you to do something without explaining how to accomplish it, please consult references on kernel development. That's by design.
- To be awarded full credit, your kernel must ***not*** crash or freeze under any circumstances.

2. Background Information

2.1. Periodic Task Parameters

In real-time systems, tasks are conventionally modeled as a periodic sequence of jobs. Jobs are released every T units of time and each job needs to finish within a relative deadline of D units of time from its release. To guarantee schedulability, we only admit a task into the system if the schedulability of the resulting task set is guaranteed. To perform a schedulability test, it is necessary to calculate the amount of resources each task uses, e.g., task τ_i's utilization $U_i = C_i/T_i$, or the worst-case response time.

However, our guarantee relies on the task to never exceed its allowed execution time, C. Unfortunately, tasks are not that obedient in practice and may *overrun* their budget due to an inaccurately estimated execution time, timing penalties from memory access, etc. To maintain the timing safety of the system in light of this, you will implement a simple monitoring framework in the kernel which will keep track of the task budget (i.e., maximum of C computation time every T).

2.2. Task Scheduling

To keep track of the amount of computation resources a task consumes, the task will need to be followed throughout its lifetime: birth, context switch in, context switch out, and death.

In lecture, you were introduced to Linux processes and threads. The kernel shares the available CPU(s) among the tasks in the system. The kernel must be able to suspend the instruction stream of one task and resume the instruction stream of another previously-suspended task. This activity is referred to as a *context switch*.

When executing on one CPU, all tasks share the CPU registers. Hence, after suspending a task, the kernel must save the values of registers to restore them when the task is resumed. The set of data that must be loaded into the registers before the task resumes its execution on the CPU is called the *hardware context*. A context switch involves saving the task control block (TCB, `struct task_struct`) of the task being switched out and replacing it with that of the task being switched in place. The context switch in the Linux kernel is initiated from one well-defined point: `__schedule()` function in `kernel/sched/core.c`.

```
static void __sched notrace __schedule(bool preempt)
{
        struct task_struct *prev, *next;
        unsigned long *switch_count;
        unsigned long prev_state;
        struct rq_flags rf;
        struct rq *rq;
        int cpu;

        cpu = smp_processor_id(); /* get current CPU ID */
        rq = cpu_rq(cpu);
        prev = rq->curr;
        ...
```

The `__schedule()` function is central to the kernel scheduler. Its objective is to find a task in the run-queue (a list of tasks ready to run) and assign the CPU to it. The function involves several kernel routines. It sets a local variable `prev`, which points to the TCB of the currently-running task on the current CPU.

```
        ...
        next = pick_next_task(rq, prev, &rf);
        clear_tsk_need_resched(prev);
        clear_preempt_need_resched();

        if (likely(prev != next)) {
                rq->nr_switches++;
                ...
                rq = context_switch(rq, prev, next, &rf);
        } else {
                ...
        }
}
```

Next, it picks a next task to be run and sets a local variable `next` to the next task's TCB. If `next` is different from `prev`, then finally context switch happens. If no runnable task has higher priority than the current task, then `next` coincides with `prev` and no context switch takes place.

2.3. Timing

The in-kernel timer of choice for this project is the high-resolution `hrtimer`. The documentation with lots of crucial usage information is available in the kernel source tree at `Documentation/timers/hrtimers.rst` and in `include/linux/hrtimer.h`. Examples are available at https://developer.ibm.com/tutorials/l-timers-list/. Understand the various modes and choose the best ones for your purposes. Do not forget to cancel the timer after use or when it is no longer needed. Not doing so is a recipe for resource leak and kernel panic.

Hrtimer works with time values of the type `ktime_t`, which is a 64-bit signed integer representing time in nanoseconds.

A good, but not the only, function for getting the current time in the kernel is `ktime_get()`.

3. Build Directory

You need to create a subfolder `proj2/` in the top-level directory of your repository. Your source files to be implemented will fall into three categories:

Category	Location in kernel repo	Build method
built-in kernel code	`proj2/kernel`	built together with the kernel image
kernel modules	`proj2/modules`	standalone using the kernel build system
user-space applications	`proj2/apps`	standalone using user-space libraries

Follow the instructions in the HW #1 handout to setup Makefile (and Kbuild for kernel source code).

4. Assignment

4.1 Periodic Real-time User-level Test Program (no points; needed for testing your kernel)

Source code location: `<your_repo>/proj2/apps/periodic/periodic.c`

This user-level application that takes **C, T, CPUID arguments** as input and busy-loops for C milliseconds every T milliseconds on the CPU CPUID. It supports C and T values up to 10,000 ms (10 secs). C and T values should be greater than 0 (C <= T). CPUID should be greater than or equal to 0 (0 means the first CPU core). The program is runnable on the stock kernel too: it does not rely on any of your modifications to the kernel. The program should continue to run until a user presses 'Ctrl-C' or terminates it by sending a KILL signal.

Download **`periodic.tgz`** from Canvas (Modules – Projects). Extract the file and build by:

```
1  $ mv periodic.tgz <your_repo>/proj2/apps/
2  $ cd <your_repo>/proj2/apps
3  $ tar xvfz periodic.tgz   # extract
4  $ cd periodic
5  $ ls
6  Makefile  periodic.c
7  $ make                    # build
```

Once compiled successfully, you can run it with some input parameters and this program will print out its PID and the given input parameters. See the below example for formatting.

Example:

```
1  #### run with C=100ms, T=300ms on CPU0
2  $ ./periodic 100 300 0
3  PID: 892, C: 100, T: 300, CPUID: 0
4  ^C
5  $
6  #### error case (C > T): C=300ms, T=100ms, CPUID=1
7  $ ./periodic 300 100 1
8  Invalid arguments: C 300, D 100, CPUID 1
9  $
10 #### run two instances in background, the one on CPU0 and the other on CPU1
11 $ ./periodic 100 500 0 &            ### '&' is to run in background
```

```
[1] 1203
PID: 1203, C: 100, T: 500, CPUID: 0
$ ./periodic 200 1000 1 &
[1] 1207
PID: 1207, C: 200, T: 1000, CPUID: 1
$
#### bringing back to foreground and terminate them
$ fg
./periodic 200 1000 1
^C
$ fg
./periodic 100 500 0
^C
$
```

Given that you are using a VM, the timing error of this program is expected to be around 20-50 ms. So use C and T params in units of 100 milliseconds (e.g., 100, 200, 300, etc.)

How to verify this app? You may want to check precisely if this app is working as expected. One way is to use `trace-cmd` (a command line tool for event tracing) and `kernelshark` (GUI front end)

Install as follows:

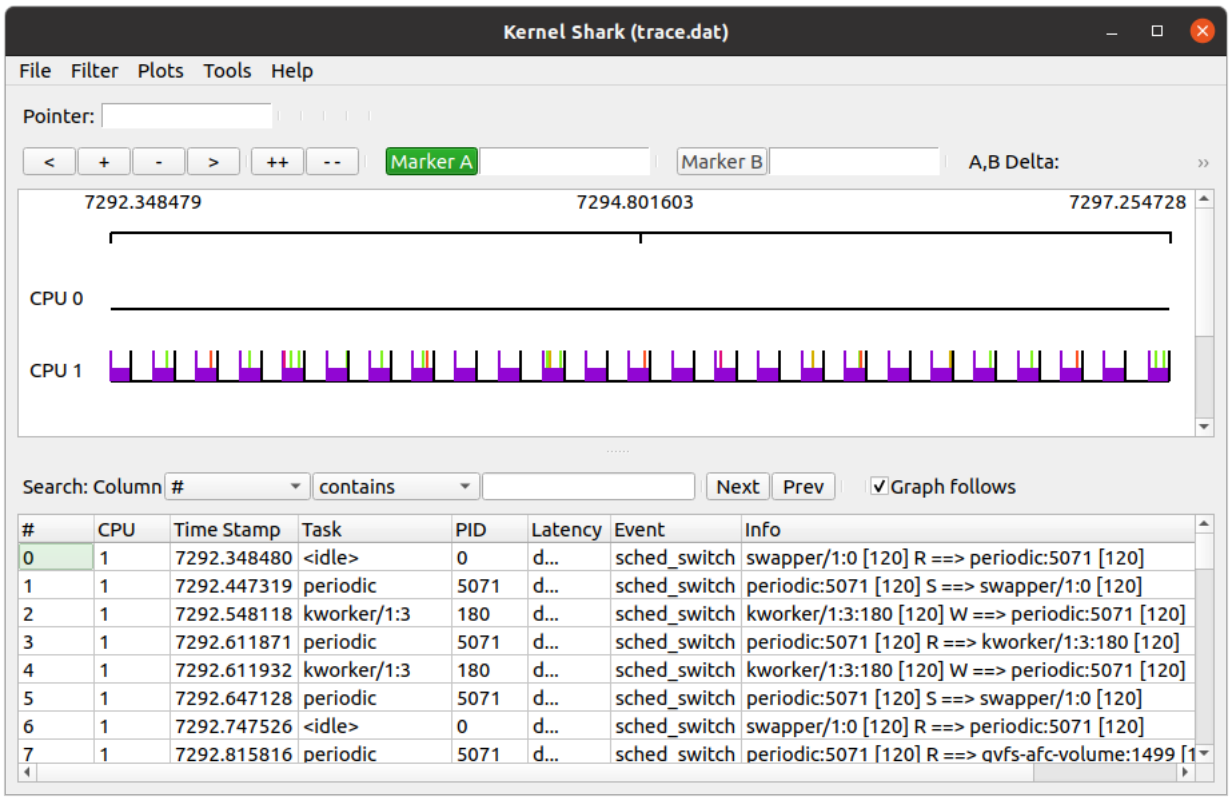
```
$ sudo apt update
$ sudo apt install trace-cmd kernelshark
```

Then, you can record scheduling events while the periodic application is running. For example:

```
$ ./periodic 100 200 1 &            ### '&' is to run in background
[1] 1207
PID: 1207, C: 100, T: 200, CPUID: 1
$
### Record scheduling events of PID 1207 for 5s
### - It takes longer than 5s to complete
### - If you want to record all tasks, try "sudo trace-cmd record -e sched sleep 5"
$ sudo trace-cmd record -e sched_switch -P 1207 sleep 5
CPU 1: 67314 events lost
CPU0 data recorded at offset=0x577000
    0 bytes in size
CPU1 data recorded at offset=0x577000
    515227648 bytes in size
...
### Terminate the periodic task after recording
$ fg
./periodic 100 200 1 &
^C
$
```

After this, you should be able to see `trace.dat` file generated in the same directory. Launch the following command to see the data.

```
$ kernelshark                ### or "kernelshark trace.dat"
```



This is the screenshot of `kernelshark`. By default, it displays all events recorded. You may want to filter out unrelated events/tasks by: clicking `Filters` > `Show tasks` or `Show events` from the menu.

If you want to trace two or more tasks, append `-P <pid>` for each of them. For example, to trace three tasks with PID 1207, 1208, and 1209 for 3 seconds: **`sudo trace-cmd record -e sched_switch -P 1207 -P 1208 -P 1209 sleep 3`**

4.2. Setting Task's Timing Parameters (20 points)

Source code location: `<your_repo>/proj2/kernel/`

(If you want to implement it as an LKM, store the source code in `<your_repo>/proj2/modules/` and indicate in the writeup that LKM needs to be loaded for testing)

As a first step to develop a task monitoring framework, your job is to add system calls that set periodic task timing parameters for a given task.

First, add timing parameters in a TCB:

```
ktime_t C, T; // C: execution time, T: period
```

The `C` and `T` parameters in the TCB should be in the `ktime_t` type so that they can be directly used with hrtimers. Recall that `ktime_t` represents time in nanoseconds!

Then implement the following two syscalls:

```
int set_rtmon(pid_t pid, unsigned int C_ms, unsigned int T_ms);
int cancel_rtmon(pid t pid);
```

Use the syscall number **450** for `set_rtmon`, and **451** for `cancel_rtmon`.

The `pid` argument is the target task's ID. The `C_ms` and `T_ms` arguments of `set_rtmon` are time values in milliseconds (ms). The maximum `C_ms` and `T_ms` allowed is 10000 ms (= 10 secs) and the minimum is 1 ms. The role of these two syscalls are simple; `set_rtmon` sets `C` and `T` parameters for the target task in its TCB, and `cancel_rtmon` clears these parameters of the target task (i.e., setting `C` and `T` to all zero).

Your syscalls are supposed to check if the input arguments are all valid (e.g., valid pid; C_ms and T_ms are within the maximum/minimum range; don't trust the provided user-level program!). The syscalls should return 0 if there is no error, and -1 if there was any error. If `set_rtmon` is called for a task that has non-zero `C` and `T` parameters in its TCB, it should return -1. Likewise, if `cancel_rtmon` is called for a task with zero `C` and `T` parameters, it should return -1.

The syscalls should work with processes as well as threads. When the given `pid` is of a thread, the setting should apply to that thread and not to any of its siblings, parent or children.

4.3. Printing Task's Timing Parameters (10 points)

Source code location: `<your_repo>/proj2/kernel/`

(If you want to implement it as an LKM, store the source code in `<your_repo>/proj2/modules/`)

Implement a syscall that prints out task timing parameters (`C` and `T` values).

```
int print_rtmon(pid_t pid);
```

Use the syscall number **452** for `print_rtmon`.

The `pid` argument is the target task's ID. `print_rtmon` prints out the target task's C and T parameters with its task ID using `printk`. If pid is -1, then `print_rtmon` should print the C and T values of **all** tasks whose C and T values are not zero. The syscalls should work with processes as well as threads.

`print_rtmon` should return 0 if it found a task with pid or pid is -1. Otherwise, it returns -1.

For example:

```
#### dmesg ####
# print_rtmon(1295);
[    256.201952] print_rtmon: PID 1295, C 200 ms, T 1200 ms
# print_rtmon(1302);
[    271.129505] print_rtmon: PID 1302, C 0 ms, T 0 ms
# print_rtmon(-1);
[    312.201952] print_rtmon: PID 1295, C 200 ms, T 1200 ms
[    312.202015] print_rtmon: PID 1323, C 100 ms, T 500 ms
[    312.202035] print_rtmon: PID 1324, C 300 ms, T 800 ms
```

In this example, PID 1302 prints all zero values, meaning that no value has been set for this task. There are a total of three tasks, PID 1295, 1323, 1324, whose C and T values are non-zero (set by `set_rtmon`).

4.4. Admission Control (20 points)

Add an admission control feature to your `set_rtmon()`. It should check check if tasks with active rtmons would be schedulable or not when the new task (= caller of `set_rtmon`) is admitted.

Implement the response time test for checking task schedulability. Assume Rate Monotonic as a priority assignment scheme. Also, assume that the system has only one CPU core (= uniprocessor system) regardless of actual number of CPUs your machine/VM has.

If all tasks including the new task remain schedulable, `set_rtmon` should returns 0. Otherwise, the rtmon should not be created, meaning that C and T values should not be assigned to the task, and `set_rtmon` should return -1.

Terminated tasks should ***not*** be considered by the schedulability test.

Note: For the ease of development, you may want to first develop and test the logic of the response time test in a userspace program and then port it to the kernel space.

4.5. Computation Time Tracking (20 points)

For each task with non-zero C and T values in the task's TCB (= rtmon tasks), keep track of the cumulative CPU time used by the task within the current period (i.e., from the beginning of a period to the end of that period). We want very precise time tracking. So you must check the time from when the task is scheduled to run and to when it is scheduled out, and accumulate this time until the end of the period.

At the end of the current period (= start of the new period), reset the task's time accumulator so that it can check CPU time again for the new period. To do this, you need to know task periodicity – start a periodic hrtimer for a task when an rtmon is created for that task. Of course, you will need to cancel the task's hrtimer when the corresponding rtmon is canceled or that task is terminated.

Extend `print_rtmon()` to print out the cumulative CPU time of the target task (usage), along with other parameters already implemented. For example:

```
#### dmesg ####
# print_rtmon(1295);
[    305.129521] print_rtmon: PID 1295, C 200 ms, T 1200 ms, usage 87 ms
[    305.402832] print_rtmon: PID 1295, C 200 ms, T 1200 ms, usage 122 ms
[    306.284120] print_rtmon: PID 1295, C 200 ms, T 1200 ms, usage 48 ms
```

IMPORTANT: Clean up any timer used upon task termination and rtmon cancellation. Remember that task may be terminated at any time by 'Ctrl-C', so `cancel_rtmon` may not be explicitly called by the task. Your kernel should take care of such cases and it should never crash.

Test your kernel with periods of tens to hundreds of milliseconds. Kernelshark will be useful to verify correctness. Small timing errors (< 10 ms in most cases and intermittent spikes) are acceptable in a VM environment.

4.6. Periodic Task Support (20 points)

Source code location: `<your_repo>/proj2/kernel/`

(If you want to implement it as an LKM, store the source code in `<your_repo>/proj2/modules/`)

Add an additional syscall to facilitate the implementation of periodic tasks under your monitoring framework:

```
int wait_until_next_period(void);
```

Use the syscall number of **453**. When a task with non-zero C and T values makes this call, the kernel suspends the task until the beginning of the next period of this task. In order to implement this feature, you would need to start a periodic hrtimer (if you already created a periodic hrtimer for Section 4.5, you can use that one).

The syscall should return 0 when the calling task has non-zero C and T values, and -1 otherwise.

An example of using this syscall to implement a periodic task would be:

```
set_rtmon(pid, C_ms, T_ms);
while (1) {
    // do computation
    wait_until_next_period();
}
```

4.7. Enforced Budget (Bonus: 10 points)

Ensure that a task with an active rtmon (non-zero C and T values) executes for up to C time units in each period, and it is suspended **immediately** when the budget is exhausted (so the task never runs more then C per T). The task will wake up at the start of the next period as the budget will be replenished. See the Tips section for suspending and resuming tasks and hrtimers.

If you implemented this bonus part, please indicate that in the writeup.

Hint: The requirement is to suspend the task **as soon as** its cumulative execution time has reached C. You will need another hrtimer for each task to enforce its remaining CPU time whenever that task is scheduled.

4.6. Writeup (10 points)

Writeup location: `<your_repo>/proj2/proj2_team00.pdf` or `proj2_team00.txt`

Submit a **plain text** or **PDF** document with brief answers to the following questions:

1. Run **three** instances of the periodic program (periodic.c) and capture their execution traces for 3 seconds using `trace-cmd`. Use the following task parameters:
   - Task 1: C = 40 ms, T = 250 ms, CPUID = 1
   - Task 2: C = 60 ms, T = 500 ms, CPUID = 1
   - Task 3: C = 200 ms, T = 1000 ms, CPUID = 1

   (If your system has only one CPU, use CPUID = 0)

   To launch the tasks at the same time:

   ```
   $ ./periodic 40 250 1 & ./periodic 60 500 1 & ./periodic 200 1000 1 &
   ```

   Attach a screenshot of `kernelshark`.
2. Run the same taskset, but after launching them, assign **real-time priorities** to each task by using `chrt` on the command line (e.g., `$ sudo chrt -f -p 80 <pid>`; see Lecture 7 slides). Use `SCHED_FIFO`.
   - Task 1: C = 40 ms, T = 250 ms, CPUID = 1 -- real-time priority: 80
   - Task 2: C = 60 ms, T = 500 ms, CPUID = 1 -- real-time priority: 79
   - Task 3: C = 200 ms, T = 1000 ms, CPUID = 1 -- real-time priority: 78

   Attach a screenshot of `kernelshark`.
3. Compare the two results and give an explanation of task behavior.
4. Give a brief summary of the contributions of each member.

5. How to submit?

Follow the instructions in the HW #1 handout.

6. Tips

6.1. General

- Look into `include/linux/sched.h` for TCB definition.
- Try to add all necessary variables in the TCB at once to avoid repeated long compilations. `sched.h` is used by many kernel source files. So whenever you modify it, kernel compile takes long as a large portion of the kernel needs to be rebuilt.
- You may need to install VirtualBox Guest Additions again when the TCB data structure is modified.
- If you don't see any "[M]" symbol when recompiling the kernel, that means no kernel module has been recompiled and you can skip "sudo make modules_install" to save time. But if you are unsure, follow all the steps: make, sudo make modules_install, and then sudo make install (and of course, reboot with the new kernel).

6.2. Task management

- You can pin a task to a core using `sched_setaffinity()`.
- `grep` inside the core kernel folder for the macros `for_each_process` and `for_each_process_thread`.
- Remember to protect accesses to shared data structures. Some of the ones you will need may be protected by semaphores, mutexes, spinlocks or RCUs. When in doubt, look at how the needed data structure is accessed elsewhere in kernel code.
- How to check each task's CPU time usage? A task consumes CPU time from when it is context-switched in to when it is switched out. So you can measure and compare timestamps within the schedule function (e.g., right before the context switch function).
- How to suspend a task? Basically, each task maintains its ***state*** in its TCB (e.g., `TASK_RUNNING` for running or ready tasks; `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE` for non-ready, blocked tasks). If the currently-running task changes its state from `TASK_RUNNING` to `TASK_INTERRUPTIBLE` and `schedule()` is *invoked*, the scheduler will find another ready task and switch to it.
- There are two ways for the scheduler invocation: direct and indirect. If the kernel is in task context (e.g., running a system call function), the scheduler can be invoked *directly* by calling `schedule()`. If the kernel is in interrupt context, the scheduler should be invoked *indirectly* by using `set_tsk_need_resched()`. These will invoke the scheduler on the current CPU before returning to user-space.
- `wake_up_process()` can be used to wake up a suspended task.

6.3. hrtimer

- Hrtimer handlers can be executed in interrupt context. This means the hrtimer handlers cannot suspend and should not use blocking functions like `msleep()` and semaphore/mutex locks (`*_trylock` functions are usable as they do not block).

- By default, on a multi-processor or multi-core platform, the handlers of hrtimrs may migrate freely among cores. This may complicate the synchronization and scheduler-invocation issues that need to be handled in the implementation.
- If you define hrtimer objects in a TCB, the `container_of()` macro can be used in an hrtimer callback function to retrieve the corresponding TCB.
- Your kernel must cancel hrtimer when a task finishes. Recall any terminating task issues an exit() syscall (even those terminated by Ctrl+C).
- To force the handlers of hrtimers to execute on the core on which the timer was started, start the hrtimer with the `HRTIMER_MODE_PINNED` bit set in the mode bitmask (e.g., for absolute time, use "`HRTIMER_MODE_ABS | HRTIMER_MODE_PINNED`" or "`HRTIMER_MODE_ABS_PINNED`" for the mode parameter; see `hrtimer.h`).

References

1. Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, Linux Device Drivers, Third Edition, O'Reilly, 2005. http://lwn.net/Kernel/LDD3/
2. Daniel P. Bovet and Marco Cesati, "Understanding the Linux Kernel," O'Reilly Media, ISBN: 978-0596005658, 2005 (3rd Edition).
3. Robert Love, "Linux Kernel Development," Addison-Wesley Professional, ISBN: 978-0672329463, 2010 (3rd Edition).