Lab 3: Reservation Enforcement and Guarantee

18-648: Embedded Real-Time Systems

Out: October 18, 2016, 12:00:00 AM Eastern Time
Due: November 7, 2016, 11:59:59 PM Eastern Time

1 Introduction

You will explore the following topics in this lab:

- Reservation enforcement
- Admission control

Before you start hacking:

- Read the entire lab handout before starting work: writeup questions and the tips section aim to give you hints about design and implementation.
- If you complete any work separately, make sure to explain to each other how things work in detail, incl. the writeup questions. During demo we may ask any teammate about any part of the lab.
- Whenever the lab 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. You are an RTOS vendor!

1.1 Background Information

1.1.1 Reservation Enforcement

As part of Lab 2 you associated a reservation with a thread and accounted the computation time it consumed. However, when a thread exceeded its budget the most your kernel did was to notify the offender with a harmless signal. Your tasks could care about budget as much as the government does. Anarchy no more! Time to endow your reservation framework with the ultimate power of suspending and resuming threads so that a thread can only run while its budget lasts.

An enforced resource reservation framework can be used to effectively transform any thread into a periodic task. You will also support an API for creating tasks that are periodic by design, by allowing a thread to voluntarily end each of its *jobs*. Recall that in our terminology a *task* is a sequence of *jobs* each released at a period boundary.

In order to keep the implementation manageable, we will not support consuming the budget by multiple instruction flows in parallel. In other words, reservations must be associated with and apply to threads and not processes.

1.1.2 Process Control

A process (or thread, which is almost identical in this respect) can be in one the following states:

`TASK_RUNNING`	Process is running. The run-queue lists all processes in this state.
`TASK_STOPPED`†	Process execution has been paused (e.g. by `SIGSTOP`).
`TASK_INTERRUPTIBLE`‡	Process is suspended (sleeping) waiting on a condition to become true. The process may be interrupted by signals.
`TASK_UNINTERRUPTIBLE`‡	Same as `TASK_INTERRUPTIBLE`, except can't be interrupted by signals.
`EXIT_ZOMBIE`†	Process has terminated, but the parent hasn't noticed yet.
`EXIT_DEAD`†	Process has terminated, and the parent is cleaning it up.

† Not linked in specific lists. Access by pid or by linked lists of the child processes for a particular parent.
‡ Subdivided into many classes, each of which corresponds to a specific event*. These are in wait queues.
* Such as a hardware interrupt, releasing a system resource, or delivering a signal.

In order to suspend a task you will need to manipulate its state and interact with the scheduler. It is important to not violate any invariants assumed by the scheduler. The most evident restrictions come from working from an interrupt context (ISR), in which hrtimer handlers execute. There is no thread that backs a software interrupt, so the handler cannot block or wait in any way since it is not an entity that can be scheduled. Also, when an ISR might want to suspend a task, that task might be the current task which makes it unsafe to call various scheduling functions.

To safely suspend a task from inside an ISR you might need to design a message-passing-like mechanism and leverage the `set_tsk_need_resched` for triggering a context switch upon return from the interrupt handler. During the context switch it becomes possible to manipulate the task by setting its state appropriately and removing it from the run-queue by using `deactivate_task`. To bring the task back to life `wake_up_process` can be used, however it is up to you to determine which context it is safe to call it from.

1.1.3 Multiprocessor Scheduling

On a multiprocessor system, tasks (and the associated reserves) can be scheduled using either the *global scheduling* paradigm or *partitioned scheduling* paradigm. This lab focuses on the problem of partitioning tasks among processors so that schedulability of the taskset on each processor is preserved. The problem is equivalent to the NP-hard bin-packing problem and is usually solved heuristically in practice.

1.1.4 Lunar Lander Application

Download the source of the Lunar Lander game using the Android SDK: `android`. Once the SDK manager opens, open the `Android x.x (API x)` node, select Samples for SDK and Install. This will download the Lunar Lander source code to the following directory: `android-sdk-linux/samples/android-1x/LunarLander`. Commit the source into the `lunarlander` repo.

To import the source code into Eclipse, select `File->New->Other` and Android Project From Existing Code under the Android folder. Hit next and point the Root Directory to the Lunar Lander directory. Select the `com.example.android.lunarlander.LunarLander` project and Finish.

Lunar Lander comes with a bug that causes it to crash upon transitioning between foreground and background mode. Please use the following link to fix this bug: `http://andgamesdevblog.blogspot.com/2011/07/where-did-i-start-and-how-to-fix-lunar.html`

2 Assignment

2.1 Writeup (10 points)

Please submit a written report with the answers to the following questions. Please be brief and to the point.

1. (2 points) Your hardware device, like any other, has a very limited number of hardware timers. Does that limit extend to the number of hrtimers that can be created? Why or why not?

2. (2 points) Assume a thread makes a blocking file read I/O call and the OS does not have the data handy in memory. The OS blocks and deschedules the thread until the data arrives from disk. When the data does arrive, how does the OS know which thread to wake up? Which kernel mechanism is used?

3. (2 points) Periodic work in the kernel can be performed by adding it to a *work queue* or to an hrtimer callback. What is a *work queue* handler allowed to do that an hrtimer handler is not?

4. (2 points) What is the difference between regular signals and real-time signals?

5. (2 points) How do Android applications work without a main function?

2.2 Reservation Enforcement (40 points)

To complete the reservation framework, the budget accounting and monitoring needs to be augmented with enforcement functionality.

2.2.1 Enforced budget (20 points)

Ensure that a thread with an active reserve runs only while its budget lasts and is suspended immediately when the budget is exhausted. While the budget is exhausted, even signals should not be able to wake up the thread. See Section 1.1.2 for details on suspending and resuming tasks.

2.2.2 An abstraction for periodic tasks (10 points)

Source code location: `kernel/rtes/apps/easyperiodic/easyperiodic.c`

Provide a system call `end_job()`[1], which suspends the calling thread until the beginning of its next period, at which point it should be scheduled normally according to its reservation. Any unused budget does *not* get carried over into the next period. While the thread is suspended, signals should not wake it up.

Create a new native application `easyperiodic` with the same semantics as the `periodic` application from Lab 2 but take pleasure in implementing it using your powerful `end_job` abstraction. Ensure that the utilization collection, export via `sysfs`, and its display in TaskMon that you implemented in Lab 2 still works for testing purposes.

2.2.3 Enforcement in action (10 points)

Source code location: `lunarlander/LunarLander/` (`build.xml` should be in this directory) and `kernel/rtes/apps/almostperiodic/almostperiodic.c`

To create a soft-real-time test subject for your reservation enforcement, hack on the popular **Lunar Lander** Android interactive video game (see Section 1.1.4) to extend it with touch-screen input. Add buttons for Start, Pause/Resume, Left Turn, Right Turn, and Fire Engine actions.

Create a new native application `almostperiodic` that is the same as the `easyperiodic` app in Section 2.2.2, except for an artificially introduced bug that at random times roughly 10-20 sec apart causes it to stop calling `end_job` for a random number of seconds roughly within 5-10s range. At all other times, the application should behave as a normal periodic app.

Disable all but one CPU to simulate a single-processor platform. Have your partner launch `almostperiodic` while you relax with a game of space travel in Lunar Lander. Set both processes to run at the same real-time priority within the `SCHED_FIFO` scheduling policy. Once you can no longer tolerate the non-real-time performance of your favorite interactive space game, use the reservation framework you have built to handle

[1] Recall that a *task* is a sequence of *jobs* each of which is released at a period boundary

both applications without jeopardizing the real-time requirement. Make use of your reservation management controls in `TaskMon` to set reserves on both applications. Observe the utilization trace of both applications before and after setting the reserve. Safe space travels!

2.3 Reservation Guarantees (40 points + 5 bonus)

Your reservation framework now ensures that no task can use more resources than were reserved for it. Next, it will also *guarantee* reserved resources, i.e. ensure that a task can use at least the reserved amount.

2.3.1 Reservation status (5 points)

Create a virtual `sysfs` file at `/sys/rtes/reserves` whose dynamic contents is a list of threads with active reserves. For each thread display its thread ID, the process ID, real-time priority, command name, and the CPU ID to which the thread is pinned in the following format:

```
TID    PID    PRIO    CPU    NAME
101    101      99      2     adb
```

2.3.2 Admission test on one processor (15 points)

When the `cpuid` argument to `set_reverse` syscall is a valid non-nonnegative value, accept a new or revised reservation only if the tasks with active reservations, including the new task, on the specified processor would remain schedulable. Check schedulability by the utilization bound test when it is sufficient or by response-time test otherwise (using Rate Monotonic as a priority assignment scheme). If a reservation cannot be guaranteed, then fail the `set_reserve` request with `EBUSY`.

2.3.3 Admission test on many processors (20 points)

When the `cpuid` argument to `set_reserve` syscall is a sentinel negative value (`-1`) that will signify *any processor*, accept a new or revised reservation only if, after partitioning the tasks among processors according to a specified heuristic, all tasks would be assigned a processor and the task set on each processor would remain schedulable. Check schedulability by the utilization bound when it is sufficient or the response-time test otherwise.

When the `cpuid` argument to the `set_reserve` system call is a special negative value, `-1`, that designates *any processor*, accept a new or revised reservation only if the current task partitioning heuristic is able to successfuly assign the new thread to a processor. An assignment is successful if only if the augmented task set is guaranteed to be schedulabile by the utilization bound test if the test is sufficient or the response-time test otherwise.

Implement all of the following bin-packing heuristics for assigning tasks to processors:

First-Fit	FF
Next-Fit	NF
Best-Fit	BF
Worst-Fit	WF

The "bins" are the physical processors that exist on the platform. The bins are ordered by ascending CPU ID. In accordance with the definition of these heuristics, a new bin can be "opened" only if an object does not fit into any of the already open bins. A bin should be "closed" when the last reservation assigned to the corresponding processor is canceled or the thread holding that reservation terminates.

If the new reservation is admitted, then the thread should be pinned to the processor to which it was assigned by the task partitioning heuristic. Existing threads with active reservations should not be affected.

Provide a virtual `sysfs` file at `/sys/rtes/partition_policy` for specifying and querying the current partitioning heuristic. Accept and return the name abbreviation in upper-case. Let `FF` be the initial default upon kernel boot or module load. Changes to the partitioning policy should be allowed only while there are

no active reservations in the system, otherwise the write should fail with `EBUSY`. The new policy selection should be in effect when the next reservation is set.

2.3.4 Period-Aware partitioning heuristic (5 bonus points)

Be creative and develop a new partitioning heuristic, with abbreviation PA, which could do better than any of the classical heuristics in terms of the number of processors required for *some* tasksets.

2.4 Demo (10 points)

Each group will also have to *demonstrate* their lab work in action to the TAs. The date and schedule of the demos will be announced via Piazza.

3 Tips

3.1 Reservations

- Please do not involve existing budgeting mechanisms, like cgroups, into your reservation framework.
- Use `struct timespec` for all quantities that represent time.
- Use hrtimers to implement the budget and period timers.
- Some reservation-related state may need to be initialized: do so at the *one* point in kernel where threads are born.
- Remember to cleanup any reservation-related state upon thread termination.

3.2 Task partitioning

- To turn the automatic Tegra-specific agent that turns cores on/off automatically, to disable frequency scaling, and to enable all cores:

```
1  $ echo 0 > /sys/module/cpu_tegra3/parameters/auto_hotplug
2  $ CPU_PATH=/sys/devices/system/cpu
3  $ for cpu in 0 1 2 3; do echo 1 > $CPU_PATH/cpu$cpu/online; sleep 1; done
4  $ for cpu in 0 1 2 3; do echo performance > $CPU_PATH/cpu$cpu/cpufreq/scaling_governor; done
```

- Feel free to do intermediate tests with only one processor enabled, but your submission must work with all cores on.

3.3 Native applications

- Use `clock_gettime` for all your timing-telling purposes in userspace.
- The `setitimer` may also be useful.

3.4 Android Programming

- Understand how resources are defined and used [1]
- Read the sections titled "Declaring Layout" and "Handling UI Events" [2]

4 How to Submit?

You must use `git` to submit all your work for this lab, including the written report. Please make sure all your commits to source trees are pushed to the corresponding repos:

kernel code and native apps	`18648/team`*number*`/kernel`
Android Lunar Lander application	`18648/team`*number*`/lunarlander`
writeup file (incl. a compiled PDF if using markdown or LaTeX)	`18648/team`*number*`/writeups`

For the user-space Android application Java code, do not commit the `.CLASS`, any other auto-generated files, or binaries. Clean the projects before committing. Check the directory tree: `build.xml` should be in `taskmon/TaskMon`.

Make sure you don't miss any files (incl. makefiles) by checking `git status` before and after your commits. Finally, make sure your commits are pushed into the master branch of the remote repo by checking `git log origin/master`. The code that will be graded and demoed has to be in the remote repo master by the deadline. This means: push frequently as you complete each part, but always make sure that what is in the remote master builds successfully. If you want to share unfinished code, push into another branch of your making.

See Section 4.4 in Development Environment Setup handout for an example work flow.

5 CIT Plagiarism Policy

Please read and understand the CMU-CIT Plagiarism Policy. All work submitted for grading are subject to the policy, which will be strictly enforced.

References

[1] `http://developer.android.com/guide/topics/resources/index.html`

[2] `http://developer.android.com/guide/topics/ui/index.html`

[3] Advanced Linux Programming by CodeSourcery LLC, New Riders Publishing. `https://archive.org/download/ost-computer-science-advanced-linux-programming/Advanced%20Linux%20Programming.pdf`

[4] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, *Linux Device Drivers*, Third Edition, O'Reilly, 2005 http://lwn.net/Kernel/LDD3/

[5] Development Environment Setup Guide on Piazza (`lab0.pdf`).