Lab 2: Budget Accounting and Instrumentation

18-648: Embedded Real-Time Systems

Out: September 29, 2016, 00:00:00 AM EDT
Due: October 12, 2016, 11:59:59 PM EDT

1 Introduction

You should learn the following from this lab:

- Linux scheduler basics
- Timers
- Signals
- The `sysfs` interface between the kernel and userspace
- Android application development with graphics features

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 Resource Reservation Framework

In real-time systems threads 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 admint a thread into the system if the schedulability of the resulting set of threads is guaranteed. To perform a schedulability test, it is necessary to calculate the amount of resources each thread uses, i.e. it's utilization $U = \frac{C}{T}$.

However, our guarantee relies on the thread to never exceed its allowed compute time, C. Unfortunately, threads are not that obedient in practice and may overrun their *budget* due to interference from non-realtime best-effort tasks or an inaccurately estimated worst case execution time. To maintain the safety of the system in light of this, you will implement a *resource reservation framework* in the kernel which will keep track of and enforce the thread budgets. The accounting piece is the subject of this lab, while the enforcement piece will be the subject of the following lab.

In lecture you learned about the *resource kernel* approach for integrating real-time and non-real-time tasks. The resource kernel provides timely, guaranteed and protected access to system resources and allows

applications to specify only their resource demands, leaving the kernel to satisfy those demands using hidden management schemes. A thread can secure a shared resource by making a *reservation* defined by parameters (C, T) in units of time. A thread holding a reservation of (C, T) is allowed to compute for no more than C units of time every T units of time.

1.1.2 Thread Scheduling

For an introduction to scheduling in Linux you can read Chapter 3 of [5] (online).

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

In Lab 1 you were introduced to Linux processes and threads. The kernel time-shares the available CPU(s) among the threads in the system. The kernel must be able to suspend the instruction stream of one thread and resume the instruction stream of another previously suspended thread. This activity is referred to as a *context switch*.

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

The `__schedule()` function is central to the kernel scheduler. Its objective is to find a thread in the run-queue list and assign the CPU to it. It is invoked by several kernel routines. The function sets a local variable `next`, corresponding to the next thread to be run, so that it points to the descriptor of the thread selected to replace `current`, the thread that was running when the function was invoked. If no runnable thread has a priority greater than the priority of `current`, at the end, `next` coincides with `current` and no context switch takes place.

1.1.3 Timing

All time values you store and work with should be of type `struct timespec`.

The in-kernel timer of choice for invoking callbacks at specified points in time is the high-resolution `hrtimer`. The documentation with lots of crucial usage information is available in the kernel source tree at `Documentation/timers/hrtimers.txt` and in `include/linux/hrtimer.h`. 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 recipe for kernel panic. See the tip about pinning timers in the Multiprocessing section of Tips.

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

1.1.4 Signals

A signal is a very short message that may be sent asynchronously to a process. In your budget accounting mechanism (without enforcement), a signal can be used to notify the application when it exceeds its budget.

Signals are distinguished by a number representing a type. Examples of signals and the corresponding events that usually trigger them, include

`SIGINT`	keyboard interrupt (e.g. shells send it on `Ctrl-C`)
`SIGSTOP`	request to suspend the process (`Ctrl-Z` in shell)
`SIGCONT`	request to resume the process (`fg` command in shell)
`SIGILL`	illegal instruction
`SIGFPE`	floating point exception
`SIGKILL`	the death signal (cannot be handled by application)
`SIGSEGV`	invalid memory reference
`SIGALRM`	a timed event

The `kill` utility can be used to send a signal manually: `kill -SIGSTOP <pid>`.
Delivering a signal to a destination process occurs in two distinct steps:

- **Send operation**. The kernel sends a signal to a destination process by updating some state in the context of the destination process. The signal is delivered for one of two reasons: (1) the kernel has detected a system event such as a divide by zero error or the termination of a child process, or (2) an authorized process has invoked the `kill` system call. A process can also send a signal to itself.
- **Receive operation**. A destination process receives a signal when it is forced by the kernel to react in some way to the delivery of the signal. The process can either ignore the signal, catch the signal by executing a user-level function called a signal handler, or terminate. A process can also set signal masks to indicate the signal types that it wants to (not) receive. The `SIGKILL` signal from an authorized process cannot be caught and will terminate the receiving process.

1.1.5 Android Application Development

Android applications are written in the Java programming language. The application is bundled into an archive file with an `.apk` extension. This package file can be transfered to the device and installed and/or run from the IDE. You can read the fundamental concepts related to Android applications at the official website [4].

You can develop your application in any IDE, but we recommend that you develop in Eclipse with the Android Development Tools (ADT) plugin, which is shipped in the bundle that you already installed earlier. You can run your application in the emulator or on the actual device hardware, both of which you have test driven during the development setup earlier.

Take a look at the Hello World tutorial [1], the Declaring Layout and Handling UI Events sections in [3], and the documentation about defining and using resources [2].

1.1.6 Java Native Interface

To (indirectly) call your system calls from Java application code you will need to make use of Java Native Interface (JNI). JNI is provided for writing performance critical parts of your Android application in C/C++ as native binaries. A good example is the "suggestive text" which is a part of the Android keyboard. Its API is exported via the JNI, such that your Java application is able to make calls into the natively-linked library. Android development environment allows easy development and integration of such native components into your Android application through the Native Development Kit (NDK), which you should already have from earlier.

To accomplish this task, wrap your system calls with a trivial native library developed using the NDK. Refer to the sample `hello-jni` project in `$(NDK)/samples` for a quick intro to how to put everything together. At a high-level you need to:

1. Create a `jni` directory within the main project folder to which you want to add the JNI component
2. Write your C source files in this directory
3. Add `Android.mk` file from `hello-jni` sample project (modifying it as needed). This file tells the build system how to build your source files.
4. Navigate to your projects directory in a shell and call the `ndk-build` script (available under the top-level directory of your NDK installation). This step will build your library and copy it to `libs` sub-directory inside your project directory.
5. Open you Eclipse workspace and call the library functions in your Java code.

2 Assignment

2.1 Writeup (10 points)

You must submit a report with the answers to the following questions. Please try to be brief and to the point.

1. (1 point) What is the difference between *concurrent* execution and *parallel* execution?

2. Imagine Dexter needs to instrument his kernel to collect some data points. The points are generated within the kernel at some memory address and need to be visualized in a userspace application. Suppose each point appears periodically at one memory address in the kernel, for example corresponding to a memory-mapped register of a device. If a point is not read before the next one is ready, then the point is lost.

   Suppose that on Dexter's platform

   - the computation overhead of reading the point from the initial memory address is negligible
   - it takes $100\mu s$ to complete a round-trip between user-space and kernel space
   - it takes 10ns to copy one data point value from kernel memory to user memory.

   (a) (2 points) As a trained syscall writer, Dexter creates a syscall to retrieve the one data value and call the syscall in a tight while loop from a userspace application. Suppose the data points are generated at a rate of 1,000 points per second. What fraction of points, if any, are lost due to the overhead delay?

   (b) (2 points) Dexter is not satisfied with a slow sampling rate, so from now on, suppose the data points are generated at a rate of 100,000 points per second. Assuming the same implementation approach, what fraction of points, if any, are lost due to the overhead delay under the faster sampling rate?

   (c) (2 points) Dexter abandons the naive approach and changes his implementation to amortize the delay of kernel-user crossing over many points by buffering the points in kernel memory. For a buffer size of 1000 points, what fraction of points, if any, are lost due to the overhead delay? (Assume Dexter reads the points once the buffer is full)

   (d) (3 points) Dexter is not satisfied with losing any points at all. How can he improve his implementation to achieve this?

2.2 Process Budget Accounting (40 points)

The first step toward implementing the reservation framework is accounting of computation time used by a thread. You'll do this via the following set of tasks.

2.2.1 Periodic test application (5 points)

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

Write a native userspace application that takes C, T, *cpuid* arguments on the command line and busy-loops for C μs every T μs on the CPU *cpuid*. Support C and T values up to 60,000 ms (60 secs). The app should be runnable on the stock kernel too: it does not rely on any of your modifications to the kernel.

Use this app to test your budget accounting and task monitor. For example, to create a periodic task that performs 250 ms of computation every 500 ms on CPU 0:

```
1   $ ./periodic 250 500 0
```

2.2.2 Reserve management syscalls (10 points)

Implement the following two syscalls:

```
int set_reserve(pid_t tid, struct timespec *C, struct timespec *T, int cpuid);
int cancel_reserve(pid_t tid);
```

The `tid` argument is the thread ID for the syscall. If the `tid` argument is 0, the call should apply to the calling thread. The C and T values should be `struct timespec`s like all time values you work with. Your kernel should support reserve parameters on the order of hundreds of microseconds. The `cpuid` specifies the CPU ID to which the thread should be pinned. Since your target platform has four CPU cores, the range of `cpuid` is 0 to 3. Your syscalls are supposed to check if the input arguments are are correct. The syscalls should return 0 if there is no error, and a negative errno error code if there was any error.

Setting a reservation should associate the C and the T value with the thread, pin the thread to the given core, and initialize the timers you need. Cancelling a reservation should remove the thread from the watchful eyes of your reservation framework and perform any cleanup that is needed. When the thread exits, any reservation-related cleanup must be taken care of. Your kernel should support changing the reservation parameters by calling `set_reserve` on a thread with an existing reservation. The reservation should apply to the thread proper and not to any of its siblings or children.

2.2.3 Computation time tracking (15 points)

For each thread with an active reservation, in the thread object, keep track of computation time used by the task since the reservation was set and across context switches. Reset this time accumulator each period. Implement your own accumulator, **do NOT use** existing accumulators that may be in the task struct.

2.2.4 Budget overrun notification (5 points)

When the kernel finds that a thread has exceeded its budget, the kernel should send a custom signal `SIGEXCESS` to the process to which the offending thread belongs. The userspace application should be able to register a handler using standard mechanisms and handle this signal however it wants.

It is sufficient to send the signal at the end of the period. However, if you like to jump-start on the following lab, feel free to make the notification happen immediately at the instant when the budget is exhausted.

2.2.5 Reservation control application (5 points)

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

Write a native userspace application that takes a string command, a thread ID, and command specific arguments and carries out the operation by executing the respective syscall:

command	command-specific arguments
`set`	budget C in ms, period T in ms, a CPU core ID
`cancel`	*none*

For example, to set and then to cancel a reserve with a budget of 250 ms and a period of 500 ms on thread with ID 101:

```
1  $ ./reserve set 101 250 500 0
2  $ ./reserve cancel 101
```

2.3 Kernel Instrumentation for Task Utilization Measurement (25 points)

Add instrumentation functionality to the kernel for estimating computation demands of threads. The kernel should support collecting utilization values for threads with active reservations during a user-initiated *monitoring session*. A utilization value for a given period is the computation time used by the thread during

that period. The instrumentation framework should provide an interface for userspace via the *sysfs* virtual filesystem.

To start and end a monitoring session, the user must write the ASCII character '1' or '0', respectively, to `/sys/rtes/taskmon/enabled` (a vitual file which is to be created by your kernel code). If the virtual file is read, the data returned in the buffer should be '1' if monitoring session is active and '0' otherwise. For example, to collect data for 10 seconds:

```
$ echo 1 > /sys/rtes/taskmon/enabled
$ sleep 10
$ echo 0 > /sys/rtes/taskmon/enabled
```

During a monitoring session, data from all threads with active reserves should be collected. If a thread terminates or has its reserve cancelled during the session, then any data collected for that thread *may* be discarded. For threads for which a reserve is set during the session data should start to be collected as soon as the reserve is set.

The data points pertaining to a given thread should be made accessible through a virtual file at path `/sys/rtes/taskmon/util/`*tid*, where *tid* is the ID of the thread. The virtual file should be created for each thread when an reserve is set on the thread and removed when the reserve is canceled or the thread terminates. Each line of the virtual file should contain a data point, that is a timestamp in ms and a utilization fraction separated by a space.

If the file is read *after* the monitoring session is stopped, then one read of the file up to end-of-file will return the data and subsequent reads will return nothing. If the file is read *during* the monitoring session, then consecutive reads up to end-of-file may all return data (if any points were generated in the time between the reads). For example, to get data points for a thread with ID 101 that had a reserve set with period 4 ms, the user might run `cat` twice during the monitoring session to get:

```
$ cat /sys/rtes/taskmon/util/101
10 0.5
14 0.25
18 0.25

$ cat /sys/rtes/taskmon/util/101
22 0.25
26 0.25
30 0.75
34 0.75
```

2.4 Task Monitor Android Application (15 points + 10 bonus)

Source code location: `taskmon/TaskMon/` (`build.xml` should be in this directory)

Unleash the power of a GUI in an Android application for interfacing into your shiny new reservation and instrumentation framework. This application will serve you well for estimating the resource demands of tasks!

2.4.1 Reservation management UI (5 points)

Create a UI layout which supports setting and canceling reserves by specifying the target thread ID, a budget C in ms, a period T in ms, and a CPU core ID. Add a selection list with a list of real-time threads running on the system as an alternative way of specifying the target thread.

Use a Java Native Interface library (Section 1.1.6) to call you thread-information and reservation-management syscalls from your Java application code.

2.4.2 Average utilization display (10 points)

To visualize the computation demands of tasks over time, add a UI layout for displaying the average utilization of threads with active reserves. Create a button (or buttons) for starting and stopping a monitoring session (see Section 2.3). After monitoring session is stopped, the UI should compute and display the average utilization of threads with active reserves collected during the session. Threads that have reserves set during the monitoring session should be included.

Retrieve the utilization data points from the kernel by reading the `sysfs` virtual files that you implemented in Section 2.3. List the `/sys/rtes/taskmon/util` directory to get the list of threads that might have data. You may choose to retrieve the data either periodically during the monitoring session or immediately after the session is stopped.

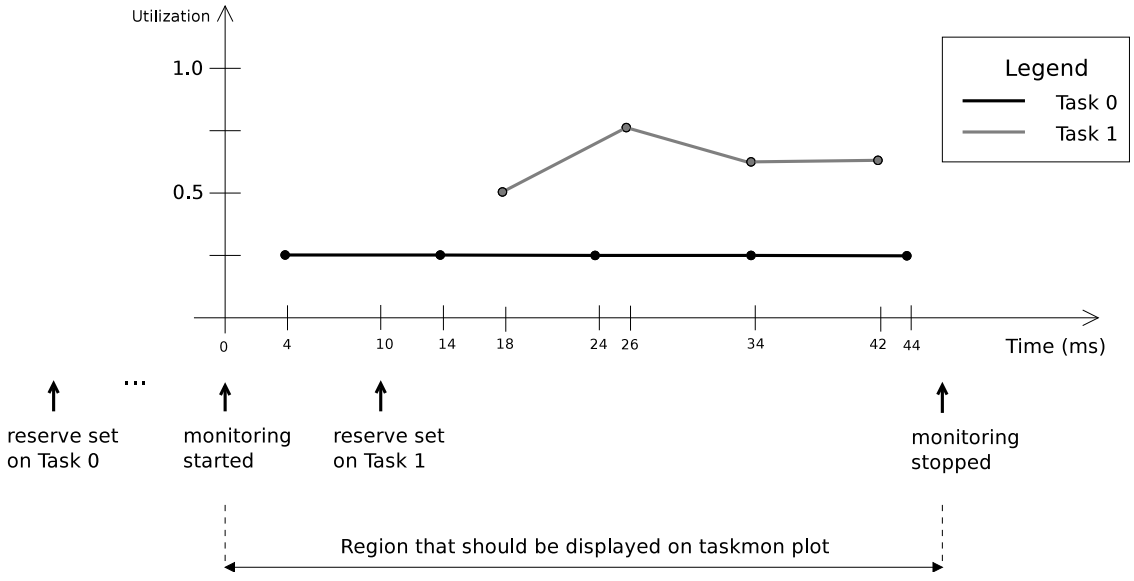
Your app should continue to record data if it is sent to background while the monitoring is on. This allows measuring the utilization of foreground applications (e.g. YouTube decoding thread).

2.4.3 Utilization data display (10 bonus points)

To visualize the computation demands of tasks over time, add a UI layout for displaying a **plot** of utilization over time for threads with active reserves. Create a button for starting and stopping a monitoring session (see Section 2.3). After monitoring session is stopped, the plot should display the data points collected during the session. All threads with active reserves should be overlayed on one plot. Threads that have reserves set during the monitoring session should be included. Provide an ability to zoom into the plot in the time dimension (i.e. change the range of x-axis). Use any visualization component you like to implement the plot UI, e.g., AndroidPlot (androidplot.com) and GraphView (www.android-graphview.org).

A schematic representation of the plot for an example scenario is shown below. The X axis is the time (in ms) and begins at the instant the user started monitoring. The Y axis is the utilization value per period as a fraction. Data points would only exist at period boundaries. The x-value of each data point should be based on a real timestamp taken at time the utilization value was generated.

The illustrated example consists of two tasks onto which reserves with periods $T_0 = 10$ ms and $T_1 = 8$ ms respectively were set. Reserve on Task 0 was set before monitoring began. Task 0 does the same amount of computation (2.5 ms) each period: $U_0 = 2.5/10 = 0.25$. Reserve on Task 1 was set while the monitoring was in progress. The utilization of Task 1 varied from 0.5 in its first recorded period to 0.75 in its second period and to 0.625 in its remaining two periods.



2.5 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 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	`18648/team`*number*`/kernel`
Android task monitor application	`18648/team`*number*`/taskmon`
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.

4 Tips

4.1 Multi-processing

- By default, on a multi-processor or multi-core platform, tasks and high resolution timers may migrate freely among cores. This complicates the synchronization issues that need to be handled in the implementation.
- To prevent migration between cores, you can pin a thread to a core using `sched_setaffinity`.
- To force the handlers of high-resolution timers to execute on the core on which the timer was started, start the timer in with the `HRTIMER_MODE_PINNED` bit set in the mode bitmask (see `hrtimer.h`).
- Doing both of the above will allow you to reduce the number of possible concurrency scenarios that could happen.
- The device has an AutoHotplug module that turns CPUs on and off automatically for power saving. In addition, the device has CPUFreq that dynamically changes the clock frequency of the CPU to increase battery lifetime. You may need to disable those features for your tests:

```
# Turn off
$ echo 0 > /sys/module/cpu_tegra3/parameters/auto_hotplug
# Disable freq scaling for all cpus (recommended to do it in these two steps in this order):
$ CPU_PATH=/sys/devices/system/cpu
$ for cpu in 0 1 2 3; do echo 1 > $CPU_PATH/cpu$cpu/online; sleep 1; done
$ for cpu in 0 1 2 3; do echo performance > $CPU_PATH/cpu$cpu/cpufreq/scaling_governor; done
```

- To temporarily convert your device into a single-processor system to help determine what's a concurrency bug and what's not, you can disable CPUs 1-3, but before that you need to disable AutoHotplug module:

```
$ echo 0 > /sys/module/cpu_tegra3/parameters/auto_hotplug
$ for i in 1 2 3; do echo 0 > /sys/devices/system/cpu/cpu$i/online; done
```

4.2 Time

- Start your computation time tracking implementation by first tracking the total thread computation time, ignoring the period. Once that is working, add the periodic functionality.
- For programming periodic work, look into `setitimer` and signals.
- Test your code with reservation periods on the orders of tens and hundreds of milliseconds.

4.3 Instrumentation for Task Utilization

- This can be implemented using a per-thread data buffer or one data buffer shared across all threads. Both approaches would work nicely, but each has its own advantages and disadvantages.

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/tutorials/hello-world.html`

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

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

[4] `http://developer.android.com/guide/topics/fundamentals.html`

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

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

[7] Development Environment Setup Guide on Piazza (`devenv.pdf`).