Lab 4: Real-Time Operating Systems

18–349 Embedded Real-Time Systems

Part 1 (Mandatory) Due: December 10, 2011, 11:59pm EDT
Part 2 (Extra-Credit) Due: December 11, 2011, 11:59pm EDT

Contents

1 Introduction ...................................................... 2
   1.1 Overview .................................................. 2
   1.2 Tasks ...................................................... 2
   1.3 Lab Support Code ........................................... 3
   1.4 Academic Integrity .......................................... 3

2 Background ..................................................... 3

3 gravelv2 Architecture Overview ................................ 4
   3.1 Interface and Separation .................................... 4
   3.2 TCBs and Kernel Stacks ...................................... 4
   3.3 SWIs and IRQs ............................................. 5
   3.4 Context Switcher ........................................... 5
   3.5 Drivers ................................................... 5
   3.6 Scheduler ............................................... 5
   3.7 Run Queues .............................................. 5
   3.8 Execution Environment ..................................... 6

4 Code Layout .................................................... 6

5 Part 2 (Extra Credit) .............................................. 6
   5.1 Schedulability ............................................. 6
   5.2 Priority-Inheritance Protocol ............................... 7

6 Completing the Lab .............................................. 7
   6.1 What to Turn In .......................................... 7
   6.2 Where to Get Help ........................................ 8
1 Introduction

1.1 Overview

In this lab, you will get to implement the internals of a real-time operating system. Real-time operating systems have a number of subsystems, but their most important functions are task scheduling, resource allocation and memory protection (a form of fault isolation). Given the time constraints of this lab, we will not be able to fully develop all three of these aspects. Until now, resource allocation was the primary focus of your previous lab project. In this project, the focus will shift to task management. The kernel for this lab is rather uninterestingly named **Gravelv2**.

The lab is divided into two parts. Part 1 deals with basic infrastructure, context switching, task management, concurrency control through mutexes and interface layers. Part 2 (extra-credit) deals with admission control and the priority inheritance locking discipline.

Part 1 is intended to be done as mandatory group work. Part 2 is intended to be done as optional, individual work for generous extra-credit points (you can individually earn up to 30% of the entire Lab 4 grade). Any collaborative work for Part 2 will void the extra-credit points for any and all collaborators; the purpose of Part 2 is to help you earn individual extra-credit towards your individual grade.

However, Part 2 necessarily builds on your real-time kernel from Part 1, and Part 2 must be completed within a single day after Part 1 is done and submitted. Part 2 has been intentionally designed to be done within a day, and is relatively straightforward if you have followed all of the recent lectures on real-time. Also, Part 2 is designed to be fully developed and debugged using the Qemu emulation environment and the cross-compilation environment provided earlier in the course; however, before final submission, we do recommend that you run your Part 2 code on the gumstix platform that your group has, to be absolutely sure.

The primary tasks for this lab are:

- Familiarizing yourself with the real-time kernel infrastructure.
- Changing your libc and userspace to work with this new kernel.
- Implementing context switching, task management
- Implementing mutexes for concurrency control
- Implementing a predicate for the UB admission-control algorithm to verify the schedulability of a given task set.
- Implementing the priority-ceiling emulation protocol.

You are provided with basic kernel skeleton-code that you will have to extend. You will also need to port your code from previous labs (with some changes) to support interrupts and context switching. You will also be responsible for providing your own userspace libc support. Within this framework, you will implement predicates for the UB test and rate-monotonic scheduling policy routines.

Please note that this lab is due at the end of the semester, and has been intentionally restructed to help you complete it by that time. As a result, there will be NO extensions. Please start early. Go to the TAs’ office hours early and often, and make use of the course-staff mailing list. As with the previous labs, the course staff can point you in the right direction if you are experiencing problems in determining how you go about doing the lab. However, it is unlikely that the course staff would be able to point out exactly where your code is going wrong.

1.2 Tasks

The end result of this lab will be a kernel that you have built, that will allow multiple tasks to run on the gumstix and also allow tasks to share data (safely) amongst each other. More specifically, here is a list of the hands-on experience that you will have by completing this lab project and the extra-credit. You will end up needing to:
1. Port your SWI handler and syscall code from the previous Gravel kernel to the Gravelv2 kernel, adding and removing syscall wrappers as needed. [Part 1]

2. Write code for additional syscalls to support task creation and task management. [Part 1]

3. Write code to perform context switching between tasks and concurrency control through mutexes. [Part 1]

4. Implement the UB admission-control test to determine if a task set is schedulable. [Part 2]

5. Write code to prevent unbounded priority inversion through highest-locker priority protocol. [Part 2]

6. Write clean, modular, maintainable, readable code. [Part 2]

1.3 Lab Support Code

Please download the support code for this lab from the course web site. The support code is available in the form of a file called lab4.tar.gz.

Throughout this lab, you will need to modify this support code in order to implement your lab solution. At the end of the lab, you must turn in these files according to the handin procedure specified in Section 6.1.

You may work and compile code using the cross-compilation environment provided earlier. Please read the cross-compilation handout for instructions on how to set up your environment. Also note that any code that you write should not be compiler- or linker-dependent. You are to strictly adhere to ATPCS standards when writing assembly code and use reasonable discretion when using GNU features in C.

1.4 Academic Integrity

The course staff will be reusing a lot of the material from this version of the course in the next one. Please do not hand other students course material from this lab as this will detract from their learning experience. The course staff may compare the code you submit with code submitted by other (current or previous) students to detect academic integrity violations.

2 Background

In this section, we introduce terminology that is common across the operating systems and real-time systems communities. This will help clarify ideas and comments in the given code, and will also make the following sections easier to follow.

Predicate A predicate is a logical statement that can either be true or false. In the context of this lab, a predicate is a function that can return either success (true) or failure (false).

Scheduler The scheduler is the part of the kernel that is in charge of deciding the order in which tasks run and the length of each run. The scheduling policy is the theoretical framework that the scheduler employs in making its allocation decisions. Note that scheduling policies often do not take into account context switching or architecture-specific implementation details. In this lab, you will be implementing portions of a kernel scheduler, and in particular, the predicates that are needed to implement the rate-monotonic scheduling (RMS) policy.

Dispatcher The dispatcher is responsible for enforcing the scheduler’s policy. The context-switch and task-switch routines fall under the purview of the dispatcher.

Run queue The run-queue or run-list is a list of tasks that are currently runnable that satisfy some criteria. On simple round-robin scheduling systems, there is a single system run-list where runnable tasks are served in a first-come first-served (FCFS) manner (hence the name run queue). Systems that have multiple task priorities do not have a universal run-queue. Gravelv2 has a “run-list” for every priority level on the system, but since there are as many priorities as tasks on Gravelv2, the scheduler enforces
that no more than one task can be in a run-queue at any time. Hence, you will not use the FCFS nature of the queue, and will work instead off a purely priority-based scheduling policy.

**Task state** During the lifetime of a task, it can be in a number of states. All tasks start out as **runnable**. The scheduler can only schedule runnable tasks to run. When a **runnable** task is scheduled to run, it is then in the **running** state. A **running** task can block on a lock or on an event; the task then moves to the **blocked** state. A task that is **blocked** cannot be scheduled to run. It can be made runnable upon the signaling of an appropriate event. In traditional operating systems, a task that is exiting will move from the **running** state to the **undead** or **zombie** state, at which point it will be reaped and have its resources returned to the system in an appropriate manner. In **Gravelv2**, no tasks exit and all of the tasks are assumed to be periodic forever (this is typical of most closed-loop real-time embedded systems that run “forever”). Please remember that the scheduler will not (and is not supposed to) schedule any task that is not runnable.

3 **Gravelv2 Architecture Overview**

In this section, we will describe the general architecture of the **Gravelv2** kernel. This is to help you get acquainted to the code base so that you can efficiently and correctly exploit all of its features.

### 3.1 Interface and Separation

Many real-time kernels do not enforce a strict boundary between the kernel and the many tasks that run on it. This is done for practical and performance reasons. Yet other schools of thought maintain that regardless of performance, kernel design must not compromise on a clear interface layer between the kernel (sometimes called the supervisor or the monitor) and the rest of the tasks. In this lab project, we are not utilizing the ARM’s virtual-memory or memory-protection subsystems. Nevertheless, we want to encourage a modular design and adequate separation between monitor tasks and unprivileged tasks. Hence, we will still maintain two separate binaries with a well-defined ABI between them.

The **Gravelv2** kernel still follows the syscall interface from the previous labs although support for individual syscalls has changed. For example, the **exit** syscall is no longer supported because tasks are assumed to be periodic for all time. The SWI number for **exit** has been deprecated and calling **exit** will not cause an invalid syscall to happen. **Gravelv2** does not attempt to return to U-Boot under all circumstances. Even on an invalid syscall, **Gravelv2** panics and stalls, but does not return to U-boot. The kernel also ignores all arguments from U-boot. The kernel API has been updated to reflect this. A new syscall has been added: **task_create**. This syscall will be a part of this lab’s focus as it instructs the kernel to launch the given task-set. Please refer to the API spec [1] for more details.

### 3.2 TCBs and Kernel Stacks

For every task that the kernel is instructed to run, the kernel maintains an in-kernel data structure that describes the task’s current state, its general execution behavior and scheduling policy information on that task. This block of information is the called the task control block or, sometimes, the thread control block (TCB). This structure and its importance have been previously emphasized in the lectures. A part of the description of the task’s current state is the task’s kernel stack. When the task is executing a SWI, it must execute on a supervisor stack. But because we have a preemptible kernel, we cannot share the supervisor stack across multiple tasks. Thus, each task gets its own kernel stack.

**Gravelv2** can only support a fixed number (64) of tasks. Hence, all of the kernel stacks and TCBs are allocated statically during kernel initialization. Users of the kernel need not worry about their tasks’ kernel resources. They only need to ascertain that they have enough user memory to allocate the tasks’ stacks and resources.
3.3 SWIs and IRQs

In \texttt{Gravelv2}, when a `swi` instruction is executed, all of the user-context registers are saved, including the supervisor `spsr`. \texttt{Gravelv2} uses the `spsr` as a scratch register to allow the IRQ handler to work correctly. Hence, whenever interrupts are enabled, the `spsr` should be considered as a scratch register that is liable to change at any time. \texttt{Gravelv2} enables IRQs quickly after saving the user context, in the SWI, to the current task’s kernel stack. This allows \texttt{Gravelv2} to be preemptible in the kernel. IRQs do not run on a stack of their own. IRQs use a temporary buffer as scratch space to transplant themselves into the current thread’s kernel stack and continue execution there. This is done carefully to avoid overwriting any of the SWI state that is already on the stack if the IRQ happens to be handled in the middle of a SWI. Please examine the top-level IRQ-handler code that has been supplied as a part of the support code and modify your own SWI-handler routine (from previous labs) appropriately so that interrupts are enabled even when a `swi` instruction is executed and everything works properly.

3.4 Context Switcher

The context switcher in \texttt{Gravelv2} runs in supervisor/kernel mode. This can be assumed because all of the SWIs and IRQs run in supervisor mode and not in separate modes. The context switcher is parameterized to take a source and a target context region. The source region is used to store all of the callee-saved registers that need to be preserved during the switch. The same registers are reloaded from the target context region. Note that the `spsr` is not saved because it is considered to be a scratch register in this lab. This entire operation is performed with interrupts disabled. The disabling of interrupts must be done externally and is not enforced in the routine itself.

3.5 Drivers

The timer and interrupt-controller drivers follow the same specification as the drivers in Lab 3. You can use your implementation of Lab 3 for this part. If you are unsure about your own code (particularly, if you wrote “spaghetti code”) for Lab 3, please talk to the course staff.

3.6 Scheduler

This section does not warrant much documentation here as you will be responsible for implementing portions of it. The scheduler uses the priority order preselected for it. This order will be determined by the code that you write. We require that you write code that implements the rate-monotonic scheduling (RMS) policy.

3.7 Run Queues

\texttt{Gravelv2} has one run-queue for every priority level on the system. Since we are implementing a rate-monotonic scheduler, there is one priority level for every task, and no more than one task is ever on the same run-queue. The task selection routines should quickly access the next highest priority task without performing a linear scan on the run-queues (as discussed during the lecture). Tasks are divided into task groups based on their priority. We have a maximum of 64 tasks in our system, requiring 6 bits to represent. We break this up into 8 groups of 8 tasks each, requiring 3 bits to represent each task-group and 3 to represent the task number within each group. We then maintain bitfields that denote whether a run queue is occupied, and whether a group is occupied.

To find the highest priority task quickly, we find the first non-empty group by using a table-based lookup. Once we have the highest group, we then look up the highest priority task in each group. Please note that priority 0 is the highest priority and 63 is the lowest. You will need to write the code for this portion in \texttt{kernel/sched/run_queue.c}. 

5
3.8 Execution Environment

The execution environment for this kernel is unchanged from Lab 3. You will follow the same procedure to load the kernel and user programs as in Lab 3. The memory layout will be unchanged. The kernel is again loaded at 0xa3000000 and the user binary is loaded at 0xa0000000.

4 Code Layout

When you download the support code, you will notice the general structure and directory organization of Gravelv2 is the same as that of Gravel. The primary differences lie in the kernel/arm and kernel/syscall directories. These directories now contain a skeleton implementation of a multi-threaded kernel. As a convention, headers that are internal to a particular kernel module are placed in the directory itself and named with a trailing _i.

kernel/arm This directory contains ARM/XScale-related hardware initialization/interaction routines. All of the drivers for Gravelv2 are contained here. The timer driver, interrupt controller driver, U-boot hi-jacker and IRQ/SWI handler all reside within this directory. Oddly enough, the SWI dispatch routine also lives in that directory.

kernel/include This directory contains all of the header files needed to successfully compile the handout. New files of particular importance are the kernel/include/sched.h and kernel/include/task.h headers. Kernel configuration parameters are contained in kernel/include/config.h.

kernel/sched This directory contains scheduler routines. You will be modifying files in here. You should read through each of the files. They contain details on how run-queues are created, how the context switcher works, along with the preconditions for invoking the context switcher, the scheduling policies and task-set schedulability predicates.

kernel/syscall This directory contains the implementation of all of the syscalls in the system. It also houses the machinery needed to verify addresses passed into the kernel. When implementing syscalls, please use these methods.

tasks/libc Please copy over your libc’s crt0 file. You will also need to add your syscall wrapper in the tasks/libc/swi subdirectory. Remove any mention of the exit syscall. Note that crt0 should now go into an infinite loop if main returns. You can feel free to add more functionality if you would like to implement test programs. You should not subtract from existing functionality.

5 Part 2 (Extra Credit)

5.1 Schedulability

Before the Gravelv2 kernel decides to schedule a task set, it attempts to ascertain the schedulability of the given task set. There are numerous techniques that one can use to verify a task-set’s schedulability. In Part 2 of the lab, we are going to use the UB admission-control test to ensure that a given task set is schedulable. The UB test has been covered in detail in the lectures. We will include the basic test here for the sake of completeness.

In a set of \( n \) periodic tasks scheduled with rate-monotonic policy, where task \( \tau_k \) has periodicity \( T_k \) and worst case execution time of \( C_k \), \( \tau_k \) will always meet its deadlines, for all task phasings, if

\[
\sum_{i=1}^{k-1} \frac{C_i}{T_i} + \frac{C_k + B_k}{T_k} \leq U(k) = k(\sqrt{2} - 1)
\]

where \( B_k \) represents the worst-case blocking the task experiences from lower priority tasks and we are assuming that tasks have been sorted in ascending order of time periods, i.e., \( T_i < T_{i+1} \).
The task scheduler calls `assign_schedule` to create a schedule for the given task set. You will implement this function. It is located in `kernel/ub_test.c` and takes an array of TCB pointers and the number of task control block pointers in the array. It is then supposed to reorder the list and return 1 if the task set is schedulable and return a 0 if it isn’t. First, sort the input list so that it satisfies rate-monotonicity. Then, run the UB test on it to see if the given task set is schedulable.

For this part of the lab, you will modify the `task_t` structure to include a blocking term, `B`, which refers to the amount of blocking time that a task experiences from all of the lower-priority tasks. You can assume that the user application will specify a correct value of this term. The new structure will be:

```
typedef struct task
{
    void (*lambda)(void*);
    void *data;
    void *stack_pos;
    unsigned long C;
    unsigned long T;
    unsigned long B;
} task_t;
```

### 5.2 Priority-Inheritance Protocol

In this part of the lab, you will also modify the `mutex_create`, `mutex_lock` and `mutex_unlock` syscalls to implement the Highest-Locker Priority (HLP) protocol that has been covered in the lectures. You can assume that all of the mutexes follow HLP for this part of the lab (your kernel implementation can use a `#define` in `lock.h` to use HLP for this part of the lab). You should modify the `dev_wait` for this part to return a `EHOLDSLOCK` error if a task calls `dev_wait` while holding a lock.

### 6 Completing the Lab

#### 6.1 What to Turn In

**Part 1:** When finished with part1, please submit the following source code and project files in an archive `lab4-part1-group-XX.tar.gz` to your group’s assigned AFS space, where XX is your lab group number (maintain the directory paths in the archive if you do not want to lose points). Do not turn in unnecessary object dumps, preprocessed sources, compiler generated assembly or object files or files that are not relevant to the lab.

**Part 2:** Because this part is optional, it is for each individual in the class to choose to undertake for individual extra credit. If you pursue this portion of the lab, here are instructions that apply to you (not your group). Please bundle all of your source code and project files for Part 2 into an archive `lab4-part2-AndrewID.tar.gz` for submission through a separate mechanism (please watch the course mailing list for announcements on how to submit the extra-credit portion) where AndrewID is your individual andrew ID (maintain the directory paths in the archive if you do not want to lose points). Do not turn in unnecessary object dumps, preprocessed sources, compiler generated assembly or object files or files that are not relevant to the lab. This portion of the lab is not intended to be pursued collaboratively, and group members must work on it separately and submit it individually. The extra-credit does not apply to the entire group, but only to individuals who attempt the extra-credit. The extra-credit is worth an additional 30% of the grade of the entire lab. The extra-credit portion is not difficult in itself, but does require that you understand the principles of real-time scheduling, admission-control tests, priority-inversion, etc., that we have covered in the lectures. Note that there is sufficient time to finish up the extra-credit portion, beyond the submission of Part 1 of the lab. For Part 2, any collaborative work and/or copying of someone else’s code (with or without variable names being modified) will be penalized, and will result in both parties (the copying person(s) and copied-from person(s)) losing the extra-credit points.

---

1. Update `kernel/include/bits/errno.h` and `tasks/libc/include/bits/errno.h` to define `EHOLDSLOCK` to be 60.
6.2 Where to Get Help

The class lecture notes are full of information and detail on how to implement your lab. At this point, all of
the lecture material necessary to complete both Part 1 and Part 2 has been intentionally covered in depth in
the lectures. Furthermore, previous lab handouts contain numerous references and useful links. Please use
these resources if you run into difficulties.

Go to the office hours of the course staff. Feel free to ask the TAs any conceptual questions and questions
about specifications. Post messages early and often to the course-staff mailing list; this is the fastest method
of getting answers to your questions.

References

projects.html](http://www.ece.cmu.edu/~ee349/projects.html)