

Due Date: **Thursday, August 6, 5:00 PM**

Final grades submitted to Banner: **Friday, August 7, 5:00 PM**

Introduction

You will be integrating the device driver you wrote as part of Project #2 into your multi-threaded pre-emptive kernel from Project #3. A key detail of your implementation will be that the device drivers interact with the kernel in order to affect a thread's runnable status. More specifically, a thread must not sit in a poll-yield loop waiting for data from the device driver (or waiting for the opportunity to write data to the device driver), or constantly calling `receive()` with timeouts, etc.

Wrong Way vs. Right Way

A "naive" integration of your device driver from Project #2 with your kernel from Project #3 would not take full advantage of the infrastructure that is made available by a) an interrupt-driven device driver, and b) a pre-emptive kernel.

Consider, for example, an interrupt-driven device driver to read and write data to the Frobnitz peripheral. Calls to `frobnitz_write()` return immediately after storing data in a circular buffer, which is then emptied one character at a time by an interrupt-driven device driver. Calls to `frobnitz_read()` return immediately with any data available (stored in a circular buffer by the interrupt-driven device driver) or with no data, if none has been received.

Approach A – No Thread Infrastructure

With no threading infrastructure in place, incoming data code may look like this:

```
...
while (1) {
    if (frobnitz_read(...)) {
        // handle data
    }
}
...
```

Approach B – Coroutines

With an infrastructure based on coroutines, incoming data code may look like this:

```
...
while (1) {
    if (frobnitz_read(...)) {
        // handle incoming data
    } else yield();
}
...
```

This same code can be used with a pre-emptive multi-threaded kernel, but it is wasteful since the thread is constantly being scheduled to run, only to find that it has no data to process hence yielding to another thread. The constant context-switch overhead is a penalty we don't really have to pay.

Approach C – Pre-Emptive Kernel

We would really like the code to look like this:

```
...
while (1) {
    go_to_sleep_until_frobnitz_data_arrives();
    // handle incoming data
}
...
```

That is, the thread “puts itself to sleep” (prevents itself from being scheduled) until the Frobnitz device driver signals the kernel that data is available and the thread should be awakened. **This interaction between the device driver and the kernel is the key concept behind this project.** Make sure you understand it.

This type of sleep-until-notified mechanism (more formally known as *synchronous I/O*) is available in Unix-type operating systems as the `select()` function (type ‘`man 2 select`’) and specifically on Linux as the `poll()` function and, in the most recent 2.6 kernels, the `epoll()` function. All of these functions operate on file descriptors (opened using `open()`) which can represent special devices (e.g., `/dev/ttyUSB0`), pipes, or even regular files. Windows-type systems have the `WaitForSingleObject()` function to implement sleep-until-notified functionality, as well as many other similar functions.

In this approach, the user thread interacts with the low-level hardware driver layer through C API calls, and specifically there is a C function call for synchronous I/O, either reading or writing, to each peripheral. Another way to integrate the device driver into the kernel is to take advantage of an existing sleep-until-notified mechanism that is already in your kernel: publish/subscribe message passing. Why not simply have the device driver insert a “special” notification into the listening thread’s receive queue when data is available (perhaps on a “special” channel)? The listening thread can then both wait for inter-thread communication notifications created when other threads call `publish()`, and device driver notifications using a single `receive()` function call¹. For example:

```
...
while (1) {
    unsigned channel, value;

    if (receive(&channel, &value, 1000) == 1) {
        // Somebody sent us a message!
        if (channel == SPECIAL_KERNEL_CHANNEL) {
            // A kernel device driver sent us a message!
            // handle device driver data
        } else {
            // Another thread sent us a message
            // handle thread data
        }
    } else {
        // No messages...blink an LED or something
    }
}
...
```

So which way should you do it? It’s up to you. You must ensure, however, that you provide a mechanism that frees the developer from having to write the poll-yield loops used in the coroutine model (or the no-kernel-at-all model). That is, threads must be asleep until notified of incoming events, and should never have to periodically wake up just to check to see if events have occurred.

Project Requirements

- You are to integrate your interrupt-driven device driver from Project #2 and kernel from Project #3. All of the requirements of these projects (e.g., code resides in FLASH, threads run in user mode, etc.) still apply.

¹You may want to type ‘`man 2 recv`’ to see the inspiration for Project #2. Note that `recv()` and `select()` can interact with each other, providing yet a third way of architecting your code.

- You are to provide a mechanism for suspending thread execution until one or both device drivers indicate an event that should awaken a thread, as described above.
- You are to develop an example application that demonstrates multi-threading and device driver usage, without using any poll-yield loops or `receive()` calls that periodically must time out (timeouts must be *optional*).

Deliverables

- C/Assembly-code listing of all code that you write. Grading elements will include:
 - the ability to explain the big picture
 - clarity and helpfulness of the comments
 - proper structure and modularity
 - spelling
 - formatting and legibility
 - correctness, robustness, handling of errors and boundary cases, etc.
- A live demonstration of your working application that exercises all the functionality of your kernel and device driver. This demonstration must involve the appropriate hardware for your device driver.
- A Makefile so that your code can be compiled simply by typing 'make'.

ALL OF THE ABOVE ARE TO BE SUBMITTED ELECTRONICALLY BY E-MAIL (except for the live demonstration)

- Don't print it out.
- Don't send it to Blackboard.
- Don't be late.

Group Assignments

All students may choose to work alone or in groups of 2. Graduate students must work by themselves.

Grading

You start with 100 points. For each deliverable, your instructor will deduct points for errors, omissions, or poor implementation. A “working project” (i.e., a live working demonstration) is worth a lot.

You are allowed (and encouraged) to show your work to your instructor as you progress through it. As usual, the more you give your instructor to work with, the more they can help.

This assignment is NOT DIFFICULT unless you really manage to lead yourself astray in your thinking. Make sure you VERIFY YOUR DESIGN WITH YOUR INSTRUCTOR so that this does not happen.

Extra Credit

An extra 20 points will be awarded for a **successful** implementation that works immediately upon boot. That is, the code runs immediately from reset without requiring SAM-BA to start the program. Suggestion: have your code check for a pushbutton being held down upon boot, and if it is pressed, jump to SAM-BA else jump to your own code. This mechanism will allow you to re-flash your code using SAM-BA without having to re-install SAM-BA every time you want to reprogram your code.

Your code will have to initialize all clocks and peripherals that SAM-BA normally does. You will also have to modify the first word in FLASH to jump to your own check-the-pushbutton code instead of to SAM-BA's start address.

An extra 20 points will be awarded for a **successful** implementation that places the ARM processor in a low-power sleep mode when there are no runnable threads, and an interrupt must occur for a thread to be woken up. You will have to positively demonstrate that your processor is in a low-power sleep mode.