CMSC 23000 **Operating Systems** **Project 2**
Autumn 2006 **October 18**

RCX kernel

Due: Monday, November 6 at 10pm

1 Introduction

This project builds on the previous project by adding support for multiple threads of control and preemptive scheduling. As part of this project, you will need to implement an interrupt handler for timer interrupts, low-level synchronization primitives, support for cooperatively scheduled threads, and support for preemptively-scheduled tasks.

Make sure that you have committed your final version by 10pm on Monday, November 6. Using Doxygen, generate the documentation for your code. Make sure that it includes both your group name and group member names! The project documentation is due in class on Tuesday, November 7.

2 Threads

The first part of Project 2 is to implement cooperative scheduled threads. The API for this mechanism is defined in `thread.h`. Each thread has its own copy of the machine registers (`r0`-`r7`, `ccr`, and `pc`) and stack. Since memory on the RCX is limited, thread stacks should be 256 bytes. You will need to design a *thread control block* (TCB) data structure for holding the state of suspended threads.

Threads are created using the function `thread_spawn`, which takes a function pointer and a data pointer as arguments. The new thread will evaluate the function applied to the data. It terminates when the function returns or when `thread_exit` is called (or when the host task terminates).

Threads are cooperatively scheduled in a round-robin order. The functions `thread_spawn`, `thread_exit`, and `thread_yield` all result in a thread context switches. In addition, the function `task_join` blocks the calling thread, which also results in a context switch.

3 Tasks

Your RCX kernel will also support preemptive scheduling of *tasks*. A task consists of one or more threads. At any time, a task has a current thread. If the thread does a blocking operation (*i.e.*, waits on a lock), then the whole task blocks. If all the threads in a task terminate, then the task terminates. The API for this mechanism is defined in `task.h`.

Table 1: 16-bit timer registers

Address	Name	Size	Description
0xff90	TIER	8	Timer interrupt enable register. This register is used to control which timer interrupts are enabled.
0xff91	TCSR	8	Timer control/status register.
0xff92	FCR	16	Free-running counter register. This register holds the current timer value.
0xff94	OCRA/B	16	Output compare register A/B. These two registers are mapped to the same address (access is controlled by the OCRS bit in the TOCR). Their values are tested against the FCR register at every timer tick; when they match an interrupt will be signaled (if it is enabled).
0xff96	TCR	8	Timer control register.
0xff97	TOCR	8	Timer output-compare control register.
0xff98	ICRA	16	Input capture register A (unused)
0xff9a	ICRB	16	Input capture register B (unused)
0xff9c	ICRC	16	Input capture register C (unused)
0xff9e	ICRD	16	Input capture register D (unused)

Tasks are created using the `task_create`, which takes a function pointer and a data pointer as arguments and creates a new task with an initial thread that evaluates the function applied to the data. A task terminates when either all of its threads terminate or one of its threads calls `task_exit`.

3.1 The 16-bit timer

You will need to use timer interrupts to drive preemptive scheduling. The RCX includes a builtin 16-bit timer, which you can use for this purpose. The workings of the timer are controlled by a collection of device registers that are mapped to memory addresses `0xff90`–`0xff9f`. Table **??** gives a complete list of these device registers and their addresses. This timer has a 16-bit counter register (`FRC`) and two 16-bit match registers (`OCRA` and `OCRB`). It can generate three different interrupts:

1. An `OCIA` interrupt is generated when the `FCR` register matches the `OCRA`.
2. An `OCIB` interrupt is generated when the `FCR` register matches the `OCRB`.
3. An `FOVI` interrupt when the `FCR` register overflows (*i.e.*, its value changes from `0xffff` to `0x0000`).

These interrupts can be individually enabled and disabled by setting the appropriate bits in the `TIER` device register. An interface to the timer device is given in `kernel/include/rcx.h`.

In addition to triggering interrupts, the above conditions also cause bits in the `TCSR` register to be set. These bits cannot be set by software, but they can be cleared by first reading them and then setting them to 0. The `TCSR` also has a control bit (called `CCLRA`) that, when set, causes the `FCR`

to be cleared upon a match with the OCRA. The rcx.h header file defines symbolic constants for testing these bits.

The FCR register is incremented at various different frequencies depending on the values of the bits 0 and 1 of the TCR register as follows:

Bit 1	Bit 0	Frequency
0	0	$\frac{1}{2}$ internal clock (8MHz)
0	1	$\frac{1}{8}$ internal clock (2MHz)
1	0	$\frac{1}{32}$ internal clock (500KHz)
1	1	Timer is disabled

3.2 Interrupt handlers

As part of initialization, your kernel will need to initialize the RCX's RAM interrupt vector. For most interrupts, you should use the default handler at ROM address 0x046a, but for the timer you will need to install your own handler(s). The rcx.h header provides definitions for accessing the vector.

4 Synchronization

With the introduction of preemptive scheduling, you must implement synchronization primitives. At the lowest level, you can use the interrupt mask bit of the CCR to disable interrupts (and thus preemptive context switches) for short periods of time. On top of this mechansism, you should build mutex locks and condition variables. The API for this mechanism is defined in mutex.h. You will need to modify the mutex.h file to include your representations of locks and condition variables. When a task terminates, any locks that it holds should be released.

Note that the memory management library you wrote for Project 1 is now a shared resource. Thus, the heap data structures must be protected from simultaneous access.

5 Inline assembly code

Implementing certain features of this project will require a small amount of assembly code (*e.g.*, saving and restoring machine registers). You may either write this code as a separate assembly-code file (use the .s suffix) or you may use inline assembly directives. For example, the interrupt-mask bit of the CCR can be set using the following statement:[1]

```
__asm__ ("orc #0x80:8,ccr");
```

More information about inline assembly can be found at http://gcc.gnu.org/onlinedocs/gcc-3.4.5/gcc/C-Extensions.html. Information about the H8/300 instruction set is available from the course web page (and in Handout 3).

[1] Note: we provide this mechanism in the rcx.h header file.

6 Grading

Your project will be graded on both correctness (70%) and programming style (30%). The documentation is evaluated as part of the style component of your grade. Failure to document your code will result in no credit for the style portion of your grade.