CMSC 23000 **Operating Systems** **Project 4**
Winter 2006 **March 1**

RCX kernel

Due: Sunday March 12 at 10pm

1 Introduction

In this final part of the kernel project, you will add higher-level communication features, user-level timer support, and some basic I/O.

Make sure that you have committed your final version by 10pm on Sunday, March 12. Using Doxygen, generate the documentation for your code. Make sure that it includes both your group name and group member names!

2 Channels

The main addition of this project are unbuffered *channels* for inter-thread communication. Unlike mutexes and condition variables, channels may be used to communicate between threads of the same task as well as between threads of different tasks. You will have to implement the following channel API:

```
typedef struct { ... } chan_t;

void chan_init (chan_t *ch);
void chan_send (chan_t *ch, void *msg);
void *chan_recv (chan_t *ch);
```

Channel communication is unbuffered, which means that both the `chan_send` and `chan_recv` operations are blocking. Unlike the mutex and condition variable operations, however, channel operations do not block the entire task, just the calling thread. Thus, they may be used for both interthread and intertask communication.

2.1 Choice

For extra credit, you may also implement the following *choice* operation

```
void *chan_select (int *n, chan_t *ch[]);
```

which blocks the calling thread until there is a messagee available on one of an array of channels at which point it returns the message. The argument `n` is used to specify the number of channels in `ch` and will be set to the index of the channel that provided the message.

3 User-level timing

Two user-level timer services should be provided by your kernel. The first is

```
long timer_clock ();
```

which returns the time since boot-up in milliseconds. Note that this is a 32-bit value. The second is

```
void timer_sleep (int ms);
```

which causes the calling thread to sleep for the given number of milliseconds.

4 RCX Buttons

The RCX has four buttons, which are labeled *On/Off*, *Run*, *Prgm*, and *View*. Each of these buttons is connected to a bit in the RCX's input ports. In addition, the *On/Off* button can signal an `IRQ0` exception when pressed (if the interrupt is enabled) and, likewise, the `Run` button can signal an `IRQ1` when pressed.

4.1 User interface

Your kernel should provide a channel-based interface to these buttons:

```
extern chan_t EventCh;

enum {
    BUT_ON_OFF, BUT_RUN, BUT_PRGM, BUT_VIEW
};
```

where the constants `BUT_ON_OFF`, `BUT_RUN`, `BUT_PRGM` and `BUT_VIEW` are used to represent the pressing of the different buttons.

5 Ticker output

We added a simple text output device to the RCX simulator called the *ticker*. This device can display 32 ASCII characters, which are addressed from left to right (character 0 is the leftmost and character 31 is the rightmost). It is controlled by a set of four 8-bit registers.

TCR The *ticker control register* is a write-only register used to send commands to the ticker. The commands are:

- `CLEAR`, which clears the display.
- `STORE`, which causes the character in the `TDR` to be stored into the position specified by the `TAR`
- `SHIFT`, which causes the characters to shift to the left one position. The rightmost position is loaded with the character in the `TDR`.

TSR The *ticker status register* is a read-only register that signals the ticker's state. The ready bit signifies that the ticker is ready for a new command, while the busy bit signifies that it is

busy. The busy bit is set when the ticker starts processing a command and is cleared when the command is completed and the status register is read (*i.e.*, the bit is *sticky*).

TAR The *ticker address register* holds the address of the character to update in its low five bits.

TDR The *ticker data register* holds the character to be displayed on the ticker.

The protocol for controlling the ticker is:

1. Wait until the ready bit in the `TSR` is set.
2. Write the data and address values into the `TDR` and `TAR` (if necessary).
3. Write the command into the `TCR`.
4. Wait until the busy bit is signaled in the `TSR`.

5.1 User interface

Your kernel should provide the following function for writing data to the ticker:

```
void print (const char *msg);
```

Characters are added to the ticker by scrolling them in from the right at 10ms intervals. This routing should be atomic; *i.e.*, the output of multiple threads should not be commingled.

6 Kernel architecture

To implement these new features, you should first implement the channel abstraction. Then, using this abstraction, you should implement server threads (and tasks) for timers, buttons, and the ticker. You can use a request/reply protocol to implement user-level operations. For example, the user-level `timer_sleep` function might be implemented as follows:

```
extern chan_t *TimerReqCh;

typedef struct {
    int          ms;
    chan_t       * wakeup;
} timer_req_t;

void timer_sleep (int ms)
{
    chan_t       wakeup;
    timer_req_t  req;

    chan_init (&wakeup);
    req.ms = ms;
    req.wakeup = &wakeup;
    chan_send (TimerReqCh, &req);
    chan_recv (& wakeup);
}
```

In this code, the a request is sent to the timer server, which includes a unique channel for getting a wakeup message.

7 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.