CS452 – FALL 2010
KERNEL 4

BILL COWAN
UNIVERSITY OF WATERLOO

A. INTRODUCTION

In the final part of your development of the kernel, you use the interrupt processing capabilities of your kernel to create tasks that provide interrupt driven input/output. User programs will then be able to call `Getc`, `Putc` and any other primitives you choose to implement to access input and output streams on RS-232 hardware.

We also expect that during this final stage of development you will polish the code of your kernel, integrating more tightly the functionality you have implemented and thereby making the code more robust.

To demonstrate your serial I/O you are to redo Assignment 0 using interrupt mediated I/O. To remind you, in Assignment 0 you used one UART to send commands to the train interface, and to receive status responses, and you used the other UART to create an interactive display on a terminal, accepting commands typed at a prompt and displaying the current state of the track. As well as testing your I/O this provides you with a valuable capability that you will certainly use when developing your train application. We explicitly expect that you will have these capabilities available for later demos.

The documentation you provide with this assignment should document your kernel completely, and will be marked with this in mind. If you have been providing good documentation as you went along most of what you need to submit has already been written.

Comment. If you are like me you have skimped on error-checking in your kernel development. This is a good time to flesh it out. In doing so divide the errors you can detect into two categories:

- errors from which it is possible to recover at run-time, and
- errors that require reprogramming.

I expect that you will find many more in the second category than in the first, because detecting an error is much easier that recovering from it. All the same internal error-detection that requires reprogramming is very useful because in most cases it provides precise diagnosis of the error.

B. DESCRIPTION

This is the part of the kernel in which you have the most freedom to make your own design decisions. Two primitives,

- `int Getc( int channel )`, and
- `int Putc( int channel, char ch )`,

are required and you will very likely wish to add others.

B.1. Kernel

At the very least, it is necessary to add one or more cases to the event handling part of your kernel. In practice, most students have a to-do list of modifications they had insufficient time to do during the earlier parts of kernel development, and this is a good opportunity to incorporate them.

B.2. Server tasks

In class I pointed out that a variety of task structures are possible. You need to create at least one server, and possibly as many as four: do what you think is best. Describe the task structure you chose, and why you chose it, in your documentation.

Regardless of task structure, you must implement `Putc` and `Getc` with the semantics given in the kernel description. They should be wrapper functions that wrap `Send`s to the serial server(s). The task ids for `Send` should be obtained by the client, using a call to the name server.

Hint. There is not a unique 'best' task structure, but some task structures are better than others. As mentioned in class, servers should almost always be `SEND_BLOCKED`, and notifiers should almost always be `EVENT_BLOCKED`. You can measure how close you come to this ideal, but it is not required to do so.

Hint. There are two serial devices (UARTs), which play very different roles in train applications. One is *much* more performance limiting than the other.

B.3. Notifier Tasks

There needs to be at least one notifier task. They may be implemented separately, or there may be a single implementation which is specialized during its initialization. The implementation should take into account the task structure of the task group that works together to provide serial I/O.

B.4. Other Tasks

Several possible auxiliary tasks – Warehouse, Secretary, etc. – are described in class. Which, if any, you implement is up to you.

B.5. Tasks to test serial I/O.

To test your serial I/O we ask you to redo assignment 0, the polling loop, this time using interrupt I/O. What collection of tasks you use is implementation-dependent, but you should describe and explain your design decisions.

B.6. Implementation comments

- Until now you have been using busy-wait I/O for debugging. It is not possible to continue doing interrupt-mediated I/O after you have

called a busy-wait primitive.* It is, however, possible to do busy-wait I/O on one UART while doing interrupt I/O on the other.

- The operation of your system depends on the priorities you choose. Starting with all tasks at the same priority and adjusting up and down is a good way of avoiding dead-ends.

C. Hand in

Hand in the following, nicely formatted and printed.

1. A description of how to operate your program, including the full pathname of your executable file which we will download for testing.
2. A description of the structure of your entire kernel.† We will judge your kernel primarily on the basis of this description. Describe which algorithms and data structures you used and why you chose them.
3. The location of all source code you created for the assignment and a set of MD5 hashes of each file. The code must remain unmodified after submission until the assignments are returned.
4. A listing of all files submitted.

* Strictly speaking, it is possible if you re-initialize the hardware, but this is not easy using normal Notifier/Server configurations. You have to do a little initialization before doing busy-wait I/O, but not much.

† If you have been accumulating your hand in documents as you have gone along, you have much of the work already done. Earl says, and his is the opinion that matters, that you documentation so far has been good.