Programming Project 8:
The Serial I/O Device Driver

Due Date: _____
Project Duration: One week

Overview and Goal

In this project, you will implement a device driver and will modify the syscall interface to allow application programs to access this device, which is the serial terminal interface. The goals include learning how the kernel makes the connection between syscalls and device drivers and gaining additional experience in concurrent programming in the context of an OS kernel.

With the addition of serial I/O to the kernel, your growing OS will now be able to run a "shell" program. This will give you the ability to interact with the OS in the same way Unix users interact with a Unix shell.

Download New Files

The files for this project are available in:

```
http://www.cs.pdx.edu/~harry/Blitz/OSProject/p8/
```

The following files are new to this project:

```
TestProgram5.h
TestProgram5.c
sh.h
sh.c
cat.h
cat.c
hello.h
hello.c
fileA
fileB
fileC
fileD
script
```

```
help
```

The following files have been modified from the last project:

```
makefile
DISK
```

The **makefile** has been modified to compile the new programs. The **DISK** file has been enlarged, since the previous version was too small to accommodate all these new files.

All remaining files are unchanged from the last project.

Changes to the Syscall Interface

In this project, you will alter a couple of the syscalls to allow the user program to access the serial device. The serial device is an ASCII "dumb" terminal; individual characters can be sent and received asynchronously, one-by-one. Characters sent as output to the BLITZ serial device will appear directly on the screen (in the window where the emulator is being run) and characters typed at the keyboard will appear as input to the BLITZ serial device.

Unix divides all I/O into 2 class called "character" and "block." In Unix, user programs can operate character-oriented devices (like keyboards, dumb terminals, tapes, etc.) using the same syscalls as for block devices (like the disk). Your kernel will also use the same syscalls, so in this project you will not add any new syscalls.

To send or receive characters to/from the serial terminal device, the user program will first invoke the **Open** syscall to get a file descriptor. Then the user program will invoke **Read** to get several characters or **Write** to put several characters to the terminal.

In the last project, the **Open** syscall was passed a filename. In this project, the behavior of **Open** will be modified slightly: if the filename argument happens to be the special string "**terminal**", your kernel will not search the disk for a file with that name; instead your kernel will return a file descriptor that refers to the serial terminal device. Sometimes we call this the "terminal file," but it is not really a file at all.

The **Close** syscall is passed a file descriptor. When **Close** is passed a file descriptor referring to the terminal "file," it will work pretty much the same (from the user-level view) as with a disk file. The file descriptor will be marked as unused and free and any further attempt to read or write with that file descriptor will cause errors.

It is an error to use the **Seek** syscall on the terminal file. If passed a file descriptor referring to the terminal file, **Seek** should return –1.

When the **Read** syscall is applied to the terminal file, it will return characters up to either the **sizeInBytes** of the buffer or to the next newline character (**\n**), whichever occurs first. **Read** will return

the number of characters gotten, including the newline character. **Read** will wait for characters to be typed, if necessary.

When the **Write** syscall is applied to the terminal file, it will send the characters in the buffer to the serial terminal device, so they will appear on the screen.

The **Create** syscall will not be implemented this term. (It would need to disallow the creation of a file called "terminal".)

Resources

The following sections from the document titled "The BLITZ Emulator" are relevant:

Emulating the BLITZ Input/Output Devices (near page 25)
Memory-Mapped I/O (near page 25)
The Serial I/O Device (near page 27)
Echoing and Buffering of Raw and Cooked Serial Input (near page 28)

You might want to stop and read this material before continuing with this document.

Implementation Hints

In this section, we will make some suggestions about how you might implement the required functionality. You are free to follow our design but you might want to stop here and think about how you might design it, before you read about the design we are providing. You may have some very different—and better—ideas. It may also be more rewarding and fun to work through your own design.

Here are the changes our design would require you to make to **Kernel.h**. These will be discussed below as we describe our suggested approach, but all the changes are given here, for your reference.

The following should already be in your **Kernel.h** file:

```
const
  SERIAL_GET_BUFFER_SIZE = 10
  SERIAL_PUT_BUFFER_SIZE = 10

enum FILE, TERMINAL, PIPE
```

The following should also be there; uncomment it.

```
var
  serialDriver: SerialDriver
```

Add a new global variable:

```
var
  serialHasBeenInitialized: bool
```

Add a new class called **SerialDriver**:

```
------------------------  SerialDriver  ----------------------------
--
--  There is only one instance of this class.
--
const
  SERIAL_CHARACTER_AVAILABLE_BIT                = 0x00000001
  SERIAL_OUTPUT_READY_BIT                       = 0x00000002
  SERIAL_STATUS_WORD_ADDRESS                    = 0x00FFFF00
  SERIAL_DATA_WORD_ADDRESS                      = 0x00FFFF04

class SerialDriver
  superclass Object
  fields
    serial_status_word_address: ptr to int
    serial_data_word_address: ptr to int
    serialLock: Mutex
    getBuffer: array [SERIAL_GET_BUFFER_SIZE] of char
    getBufferSize: int
    getBufferNextIn: int
    getBufferNextOut: int
    getCharacterAvail: Condition
    putBuffer: array [SERIAL_PUT_BUFFER_SIZE] of char
    putBufferSize: int
    putBufferNextIn: int
    putBufferNextOut: int
    putBufferSem: Semaphore
    serialNeedsAttention: Semaphore
    serialHandlerThread: Thread
  methods
    Init ()
    PutChar (value: char)
    GetChar () returns char
    SerialHandler ()
endClass
```

The following field should already be present in class **FileManager**:

```
serialTerminalFile: OpenFile
```

The following field should already be present in class **OpenFile**:

```
kind: int               -- FILE, TERMINAL, or PIPE
```

The serial device driver code will go into the class **SerialDriver**, of which there will be exactly one instance called **serialDriver**. In analogy to the disk driver, the single **SerialDriver** object should be created in **Main** at startup time and the **Init** method should be called during startup to initialize it. You'll need to modify **Main.c** accordingly.

The **SerialDriver** has many fields, but basically it maintains two FIFO queues called **putBuffer** and **getBuffer**. The **putBuffer** contains all the characters that are waiting to be printed and the **getBuffer** contains all the characters that have been typed but not yet requested by a user program. The **getBuffer** allows users to type ahead.

There will be only two methods that users of the serial device will invoke: **PutChar** and **GetChar**. **PutChar** is passed a character, which it will add to the **putBuffer** queue. If the **putBuffer** is full, the **PutChar** method will block; otherwise it will return immediately after buffering the character. **PutChar** will not wait for the I/O to complete.

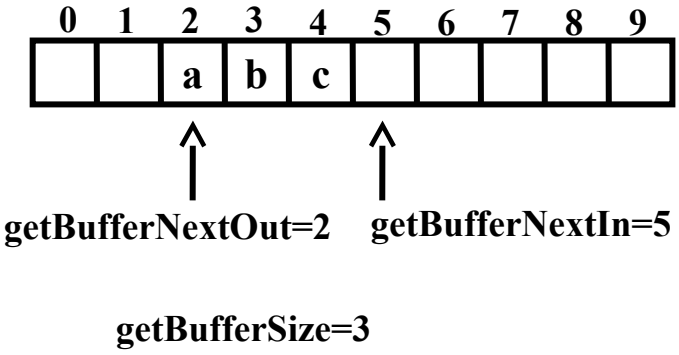
The **GetChar** method will get a character from the **getBuffer** queue and return it. If the **getBuffer** queue is empty (i.e., there is no type-ahead), **GetChar** will block and wait for the user to type a character before returning.

The **Read** syscall handler should invoke the **GetChar** method and the **Write** syscall handler should invoke the **PutChar** method.

Each of these buffers is a shared resource and the **SerialDevice** class is a monitor, regulating concurrent access by several threads. The buffers will be read and updated in the **GetChar**, **PutChar** and **SerialHandler** methods, so the data must be protected from getting corrupted. To regulate access to the shared data in the **SerialDriver**, the field **serialLock** is a mutex lock which must be acquired before accessing either of the buffers. (Our design uses only one lock for both buffers, but using two locks would allow more concurrency.)

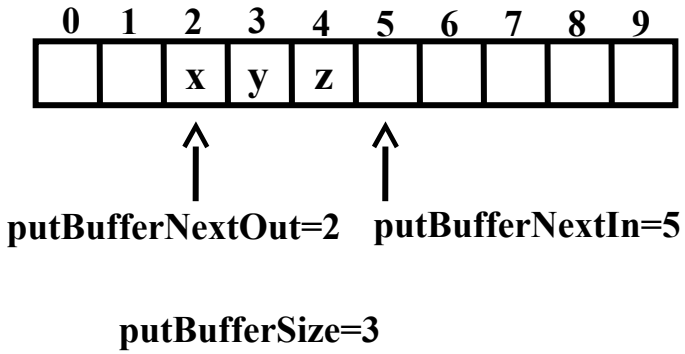
Look at **getBuffer** first. The **GetChar** routine is an entry method. As such it must acquire the **serialLock** as its first operation. The variables **getBufferSize**, **getBufferIn**, and **getBufferOut** describe the status of the buffer.

Here is a **getBuffer** containing "abc". The next character to be fetched by **GetChar** is "a". The most recently typed character is "c".



If the **getBufferSize** is zero, then **GetChar** must wait on the condition **getCharacterAvail**, which will be signaled with a **Down()** operation after a new character is received from the device and added to the buffer. After getting a character, **GetChar** must adjust **getBufferNextOut** and **getBufferSize** before releasing **serialLock** and returning the character.

Next look at **PutChar**. There is a similar buffer called **putBuffer**. Here is an example containing "xyz".



The **PutChar** method must first wait until there is space in the buffer. To handle this, You can use the semaphore called **putBufferSem**. This semaphore is initialized with

```
putBufferSem.Init (SERIAL_PUT_BUFFER_SIZE)
```

which starts the semaphore with one excess signal per available buffer entry. Each time a character is removed from the buffer, it will create an additional space, so every time a character is removed (by the **SerialHandler** method, discussed later), the semaphore will be signaled with an **Up** operation. By waiting on the semaphore with **Down**, **PutChar** ensures that there is at least one free space. Then it acquires the **serialLock**. (Note that even though other threads may sneak in and run between the completion of the **Down** and the acquisition of the **serialLock**, there will always be at least one free space.)

PutChar will then add its character to the next "in" spot in the buffer and adjust **putBufferNextIn** and **putBufferSize**. Then it will release the lock. Finally, before returning, it will signal another semaphore, called **serialNeedsAttention**, which will wake up the **SerialHandler** thread.

In our design, the serial device will be controlled by a kernel thread called "SerialHandler". Every time the serial device interrupts the CPU, this thread will be awakened. Also, every time **PutChar** adds a character to the **putBuffer**, this thread will be awakened.

The **SerialHandler** thread has two tasks. (1) If a new character has been received (i.e., the user has pressed a new key), the new character must be fetched from the device and moved into the **getBuffer**.

(2) If the serial transmission channel is free (i.e., done transmitting the previous character) and there are more characters waiting in **putBuffer** to be printed, the outputting of the next character must be started. The thread must also wake up any other threads waiting on the **getBuffer** (becoming non-empty) and the **putBuffer** (becoming non-full).

In the **SerialDriver** class there is a semaphore called **serialNeedsAttention**, which will be signaled (with **Up**) to wake up the **SerialHandler** thread. The code for **SerialHandler** is an infinite loop which waits on the semaphore, then checks things, and then repeats (going to sleep until the next time the semaphore is signaled).

The **SerialInterruptHandler** routine should be modified to signal the **serialNeedsAttention** semaphore and thereby wake up the serial handler thread. Unfortunately, this semaphore will not be initialized when the OS first begins. Although the semaphore will be initialized as part of the OS startup, serial interrupts may occur from the very beginning. An attempt to signal a semaphore that has not yet been initialized will result in an "uninitialized object" error. You don't want that!

To deal with serial interrupts that might occur before the semaphore has been initialized, You can add a new global variable

```
serialHasBeenInitialized: bool
```

In KPL, global variables are always initialized to their zero values; for a boolean this is "false". The code in **SerialDriver.Init** should create the semaphore and initialize it. As the last thing it does, **Init** should set

```
serialHasBeenInitialized = true
```

Here is the code for the **SerialInterruptHandler** function:

```
currentInterruptStatus = DISABLED
if serialHasBeenInitialized
  serialDriver.serialNeedsAttention.Up()
endIf
```

Next let's look at **SerialDriver.Init**. First it might want to print the message "Initializing Serial Driver..." The two fields called **serial_status_word_address** and **serial_data_word_address** are pointers to the memory-mapped addresses of the two serial device registers. They should be initialized here and will never change. (KPL cannot handle "const" values that are pointers, so instead, you can make them fields of **SerialDriver**.)

Next, **Init** must initialize the **serialLock**. Then **Init** must initialize the fields associated with the input buffer. They are: **getBuffer**, **getBufferSize**, **getBufferNextIn**, **getBufferNextOut**, and the **GetCharacterAvail** condition. Next, **Init** must initialize the fields associated with the output buffer: **putBuffer**, **putBufferNextIn**, **putBufferNextOut**, and the **putBufferSem** semaphore. As mentioned above, the argument to **putBufferSem.Init** indicates one initial signal per buffer slot. Next, **Init** must initialize the **serialNeedsAttention** semaphore.

Then **Init** must create a new thread (a kernel thread) which will monitor the serial terminal device. This is the **serialHandlerThread** field in **SerialDriver**. As you know, the **Thread.Fork** function requires a pointer to a function and a single integer argument. You should create a function (called **SerialHandlerFunction**) which ignores the integer argument and immediately invokes **serialDriver.SerialHandler** method. This method contains all the code and it never returns.

Finally, the **Init** method should set **serialHasBeenInitialized** to true and return.

As mentioned before, the **SerialHandler** method contains an infinite loop. The first thing in the loop is a wait on the **serialNeedsAttention** semaphore. This semaphore can be signaled by either the **PutChar** method (when a new character is added to the output queue) or when a serial interrupt occurs. In either case, the thread will wake up, check things, and then go back to sleep, waiting for the next signal.

When awakened, the **SerialHandler** thread will need to look at the serial device to see if a character has arrived at the device (i.e., a key has been pressed). So it must query the serial device status register and check the "character available" bit. If set to 1, it must get the character from the serial device data register. Then the **SerialHandler** must add it to the input buffer. This requires first acquiring the **serialLock**.

It is possible that the input buffer is full and this must be checked for. If the **getBuffer** is full, we have a case of the user typing too many characters ahead, before the program has asked for them. In our design, the character is simply dropped (i.e., do not add it to the buffer). Instead, you should print out a message containing the character. This will be very helpful in debugging.

```
print ("\nSerial input buffer overrun - character '")
printChar (inChar)
print ("' was ingored\n")
```

After adding the character to the buffer, the **SerialHandler** needs to signal the **getCharacterAvail** condition, then release the **serialLock**.

After dealing with the input stream, the **SerialHandler** needs to look at the output stream. (It would also be correct to handle the output before the input.)

First, you need to query the status register and check the "output ready" bit. A 1 bit indicates the device is ready to transmit another character, so next you need to check to see if there are any characters queued for output. Before you check **putBufferSize**, you'll need to acquire the **serialLock**. If there is at least one character in the queue, you can remove it (adjusting **putBufferSize** and **putBufferNextOut**) and move it into the serial device data register. Finally, you'll need to signal **putBufferSem**, to wake up any **PutChar** threads waiting to add characters to a full buffer. And don't forget to release the **serialLock** no matter what the code does or your OS will freeze up.

Regarding buffers and pointers, here is a little trick. Assume you have the following code (not part of this project):

```
var buffer: array [MAX_SIZE] of char = ...
    nextPos: int
```

To add an element to the buffer, you'll need to increment the **nextPos** index variable. The following code uses the **mod** operator when it adds 1, which cause the buffer to be a "circular" buffer.

```
buffer[nextPos] = x
nextPos = (nextPos + 1) % MAX_SIZE
```

If **MAX_SIZE** = 100, then this code will add 1, going from 99 back to 0. The same trick works when decrementing index values:

```
nextPos = (nextPos - 1) % MAX_SIZE
```

The specification says that a user program can open "terminal" just like any other file. The file descriptor array associated with each process points to **OpenFiles**, with a null value indicating that the given file descriptor is not an open file.

Since the **Open** syscall must assign a new file descriptor when called for "terminal" and the new file descriptor must point to an **OpenFile**, you will need two kinds of **OpenFile**. One kind is for files and one is for the terminal. Later, we mention the possibility of implementing pipes, so there are really three kinds of **OpenFiles**, but we'll ignore pipes for now.

The **kind** field in **OpenFile** will have one of the following values...

```
enum FILE, TERMINAL, PIPE
```

Since there is only one terminal, there will only be one **OpenFile** whose **kind** is **TERMINAL**. Since there is exactly one **OpenFile** for the terminal, you can pre-allocate this **OpenFile** object. The logical place to do this is in **FileManager.Init**. You can use the field called **serialTerminalFile** in the **fileManager** object. In **Init**, you can create this unique **OpenFile** object, set its **kind** to **TERMINAL**, and make **serialTerminalFile** point to it.

The **serialTerminalFile** is pretty much a dummy place holder. None of its other fields (**currentPos**, **fcb**, **numberOfUsers**) will be needed.

The **Open** syscall handler requires very few changes. Presumably in the previous project, you began by copying the String argument (**filename**) from the virtual address space to a kernel buffer (and aborting if problems). Then you found the next free entry in the **fileDescriptors** array (and aborting if none).

At that point, you can check to see if the **filename** is equal to "terminal" (see the **StrEqual** function from the **System** package). If so, you can just make the **fileDescriptor** entry point to the **OpenFile** called **serialTerminalFile** and return.

The syscall handler for **Close** is straightforward. You'll need to reclaim the entry in the **fileDescriptors** array, but that is all. In particular, **FileManager.Close** should not do anything if called on **serialTerminalFile**. Or perhaps you simply avoid ever invoking **FileManager.Close** on the **serialTerminalFile**.

Modifying the **Read** syscall handler will require a little more effort. Presumably in the last project your **Handle_Sys_Read** function began by checking the **fileDesc** argument and locating the **OpenFile** in question (and aborting if problems). After possibly dealing with a **sizeInBytes** of zero or less, you can insert code to see if you are dealing with the **serialTerminalFile** object, instead of a regular disk file.

If so, you'll need to call **SerialDriver.GetChar** once to get each character from the device. We'll leave the details to you, but perhaps you'll use a single loop which calls **GetChar** once per iteration. You'll need to keep track of the virtual address in which to store the next incoming character. You'll need to perform the virtual-to-physical translation and check to make sure (1) that the virtual page number is between 0 and the top legal page in the address space, (2) that the page is valid (In this project all pages should be valid, since we haven't yet implemented paging to disk.), and (3) that the page is writable. You'll also need to set the page to dirty and referenced, under the assumption that this would be needed if we were swapping out pages. The **Read** syscall must return the number of characters gotten from the input stream and moved into the user-space buffer. The reading will stop just after a newline (**\n**) character, but the user program will always get at least one character unless there is an error with the arguments to the syscall, or the end-of-file (EOF) character (control-D, ASCII value 0x04) is typed as a first character on a read. When EOF is typed, the read buffer should not be modified any further and the number of characters read until before EOF was pressed should be returned. In particular, if EOF is the first character on a read, the **Read** syscall should return 0 without modifying its buffer.

The modifications to the **Write** syscall handler are quite similar. A single loop can call **SerialDriver.PutChar** once per iteration. You'll need to have the same checks on the arguments and the same checks on the virtual address pointer. Of course the code should not set the page to dirty for write operation.

The KEYBOARD-WAIT-TIME Simulation Constant

One of the simulation constants used by the emulator is

KEYBOARD_WAIT_TIME

The value of this number tells the emulator how fast the serial terminal device is. In particular, it tells about how many instructions are to be executed between serial interrupts.

If, for some reason, your kernel does not retrieve an incoming character from the terminal device fast enough, the character might get lost when the next character comes in. If this happens, the emulator (which checks for various program errors) will notice that your OS is failing to get incoming characters fast enough and will print out a message such as:

ERROR: The serial input character "g" was not fetched in a timely way and has been lost!

If you see this message, it indicates that your kernel has an error. It is not getting the incoming characters when it should.

The default value for KEYBOARD_WAIT_TIME (30,000) should be more than enough to give your device driver time to process each character and add it to the type-ahead buffer. If you run into this error, the solution is to fix your kernel, not modify the simulation constant!

Of course the user program may fail to call **Write** fast enough to prevent the type-ahead buffer from overflowing, but that is a different problem.

Raw and Cooked Input

Review the material in the document "The BLITZ Emulator" regarding "raw" and "cooked" input. You should play around with your program using both "raw" and "cooked" mode. See the **raw** and **cooked** commands or the **–raw** command line option.

In cooked mode, which is the default, the host Unix system will echo all characters as you type them. Only after you hit the "enter" key will any characters get delivered to the emulator and hence to the BLITZ serial device and to your BLITZ kernel code.

In general, cooked mode is very nice because it lets the user edit his/her input (using the backspace key) and relieves most Unix programs from the burden of echoing keystrokes and dealing with the backspace character.

But be aware that with cooked mode, the BLITZ emulator may get frozen, waiting for you to hit the enter key. Or it may not. Since this may be rather confusing, the BLITZ emulator will print a message whenever it stops executing BLITZ code and is just waiting for user input.

The emulator takes a command line parameter **–wait** that tells it what to do when there is nothing more to do. If you go back and look at the code in the thread scheduler, you'll see that when a thread goes to sleep and there are no remaining threads on the ready list, the "idle thread" will execute the "wait" instruction, which suspends CPU execution and waits on an interrupt.

In the past projects, we did not use the **–wait** option, so when a "wait" instructions was executed, the emulator would print out the familiar message:

```
A 'wait' instruction was executed and no more interrupts are scheduled...
halting emulation!
```

With this project, the user is now able to type input so we don't want the emulator to just quit. We want the kernel to wait for incoming events—keystrokes, in particular—wake up, service the interrupts, and possibly resume execution in some user-level thread.

So in this project you'll need to use **–wait** on the command line, e.g.,

```
% blitz -g os -wait
```

or

```
% blitz -g os -wait -raw
```

Now, you'll might see a different message:

```
Execution suspended on 'wait' instruction; waiting for additional user
input
```

When you see this message, the emulator has stopped executing instructions and is waiting for you to enter something. This message only appears when the emulator is running in cooked mode; in raw mode the emulator will just quietly wait for the next keystroke.

But now there is another problem: How can you stop the emulator? The answer is by hitting **control-C**.

Hitting control-C once will suspend BLITZ instruction emulation and put you back in the debugging command line loop. You might see something like this:

```
Beginning execution...
==================  KPL PROGRAM STARTING  ==================
Initializing Thread Scheduler...
Initializing Process Manager...
Initializing Thread Manager...
Initializing Frame Manager...

*****  Control-C  *****
Done!  The next instruction to execute will be:
026C5C: A3FFFFF8    bne   0xFFFFF8   ! targetAddr = _Label_168_2
>
```

Control-C behaves a little funny when in "cooked" mode. You may need to hit the ENTER/RETURN key one or two times after hitting control-C before you see the ">" prompt.

Hitting control-C twice in a row will terminate the BLITZ emulator, which could be useful if the emulator has a bug. (As if...!)

Dealing With \n and \r

In the **Read** syscall handler, you should replace any incoming **\r** characters by **\n** and treat the character just like the **\n** character (i.e., return from the **Read** syscall immediately without waiting for additional characters).

Why? Because if you are running the emulator in "raw" mode, some terminals will send a **\r** character whenever the key marked "enter" or "return" is struck. By substituting **\n** for **\r**, the BLITZ user-level program will never see a **\r** character and can work only with **\n** characters.

In the **Write** syscall handler, whenever the user-level program tries to send the **\n** character to the serial device, you should insert an additional call to send a **\r** as well. Perhaps this code will work:

```
if ch == '\n'
  serialDriver.PutChar ('\r')
endIf
serialDriver.PutChar (ch)
```

This may be helpful in raw mode and should not have any effect in cooked mode. You might enjoy experimenting to see what your terminal does if this additional **\r** is left out. (Try hitting control-J which will send a **\n** to your program. Try hitting control-M which will send **\r** to your program.)

Ideally, an OS would perform character editing and everything associated with cooked mode in the terminal driver code. Ideally, you would run your kernel using the **–raw** option in the emulator and your driver would perform all echoing and character editing. You might wish to experiment with such modifications, but they are not part of the assignment.

The "Print" Functions

Up to now, you have been using functions such as **print**, **printInt**, **printIntVar**, and **printHex** to assist in debugging your kernel code. These functions do not work like normal I/O on any real computer. Instead, these functions all make use of a BLITZ instruction called "debug2" which would not be found on any real computer. This magic little instruction will cause some string or number to be immediately printed out. There are no devices to interface with, no delays, and no interrupts. The output occurs "atomically" (i.e., all at once, with no intervening instructions) which turns out to be *very, very* useful in being able to read output from concurrent programs.

In a real kernel, there is a similar mechanism for printing to facilitate debugging. However, the output is written to an in-memory buffer (rather than displayed), where it can be examined (after the kernel has crashed) by some simpler program that copies the "output" sitting in memory to somewhere where it can be read by a human. A real nuisance, but debugging kernel code is only for the strongest of programmers!

In order for user-level programs to print, they should call **Write** on the serial terminal file. Technically, any use of the "debug2" instruction ought to be removed, but we have left it in. The **print**, **printInt**,

printIntVar, etc. functions are for debugging use only; they are not like anything found in a real kernel. To print, a user-level program needs to call a function (such as **printf** in C) which in turn will invoke the **Write** syscall.

In our test programs, we will dispense with library functions like **printf** which call **Write** and just invoke **Write** directly. Likewise, we will not get around to implementing input functions like **scanf**, but will just invoke the **Read** syscall directly.

The Shell Program

After completing this project, your kernel will have enough functionality to support a Unix-like shell. In particular, your kernel can support the following features:

(1) Print a prompt (such as %), read in the name of a program, and execute that program loaded. For example:

```
% prog
```

The file called "prog" will be executed with file descriptor 0 (stdin) pointing at the terminal and file descriptor 1 (stdout) pointing at the terminal.

(2) Redirect input using the **<** character, so that stdin comes from a file. For example:

```
% cat < myFile
```

(3) Redirect stdout using the **>** character, so that stdout goes to a file. For example:

```
% ls > temp
```

(4) Deal with stderr (file descriptor 2), perhaps using **>2** for redirection.

(5) Deal with starting jobs, without waiting for their completion. For example:

```
% cat < myFile > temp &
[1] 723
%
```

In this example, 723 is the process id. We could also implement some other job-control functions like **fg** and **bg**.

(7) Nested shell invocation, for example

```
% sh < script > output
```

(8) We even have enough to implement some fancy (and complex) features such as some shell programming constructs and command-line editing and history.

(9) If we include the implementation of pipes (as discussed below), we could also add the ability to pipe the output of one program to another program. For example:

```
% cat | wc
```

What we don't have yet is any ability to pass command-line arguments to the program, as in:

```
% blitz -g os -raw
```

As part of the testing suite, we are providing a shell program called **sh**. After finishing this project, see if your kernel can execute this shell!

Pipes

As an extension (if you have enough time) you might consider adding pipes to your kernel.

You'll need to add a new syscall, **Pipe**, which takes no arguments. This syscall will create a new pipe and return a file descriptor which refers to it. It is much like the **Open** syscall. It will need to allocate a new **OpenFile** object. The **kind** of this new **OpenFile** object will be neither FILE nor TERMINAL, but PIPE. You'll need to modify the **OpenFile** class to add some sort of a buffer to it. (How many bytes should the buffer be? Only one byte is really necessary but more will allow greater efficiency for programs using pipes when a producer outruns a consumer.)

You'll also need to add something to control the concurrency and synchronization between producers and consumers. (When the buffer is full, any process trying to write to it must be suspended. When the buffer is empty, any process trying to read from it must be suspended.) The code in **Read** and **Write** will be quite similar to the code for dealing with the terminal file.

What to Hand In

The **p8** directory contains a new user-level program called:

TestProgram5

Please modify your code to load **TestProgram5** as the initial process.

TestProgram5 is structured like the previous test programs, but in addition to the individual test functions, it also contains a menu-driven interface. Once you get the serial device code more-or-less functioning, you can change the **Main** function to invoke the **Menu** function. Then you can run the

various tests interactively from a menu, instead of recompiling each time. This should make your life easier when debugging and playing with raw and cooked modes.

After you have finished coding and debugging, please run each test (except **Menu** and **Shell**) and hand in the output from each test. A separate document, called **DesiredOutput.pdf**, shows what the correct output should look like. (Due to all the funny characters involved, a file created with script may be a little confusing!)

For the tests named **KeyTest** and **EchoTest**, **LineEchoTest**, don't obsess on getting your output to exactly match the **DesiredOutput**; these tests are more for you to play around with to understand how terminal I/O works. For the other tests (**BasicSerialTest**, **EOFTest**, **OpenCloseTest**, **TerminalErrorTest**) your output should match the **DesiredOutput** file.

Be sure to *use the same code to execute all tests*. Please hand in only one copy of your **Kernel.c** code and do not hand in any output that was produced by a different version of your code!

*Do not change **TestProgram5***, except to uncomment one of the lines in the **main** function.

During your testing, it may be convenient to modify the tests as you try to see what is going on and get things to work. Before you make your final test runs, *please recopy **TestProgram5.c** from our directory*, so that you get a fresh, unaltered version.

Please hand in hardcopy of your **Kernel.c** code. You only need to hand in those functions and methods that you created/modified. This will probably include:

Handle_Sys_Open
Handle_Sys_Close
Handle_Sys_Read
Handle_Sys_Write
Handle_Sys_Seek
SerialInterruptHandler
All SerialDriver code

Please circle or highlight the code you have written for this project.

Please attempt to run the **sh** shell program, but do not hand in any output from that. Instead, please write directly on your output something like...

The shell program ran okay.

or

I was unable to run the shell program.