CSCI 447 - Operating Systems, Winter 2020
Assignment 7
The Serial I/O Device Driver
Pipe and Final ToyFS System Calls

Due Date: Friday, March 13, 2020
Points: 225

1 Overview and Goal

In this assignment, 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 a similar way Unix users interact with a Unix shell.

2 Download New Files

The files for this assignment are available in:
`https://facultyweb.cs.wwu.edu/~phil/classes/w20/447/a7`
The following files are new to this assignment:

```
Environ.h
Environ.k
KernelChanges
TestProgram5.h
TestProgram5.c
UserLib.h
UserLib.k
cat.h
cat.k
chmode.h
chmode.k
cp.h
cp.k
du.h
du.k
df.h
df.k
echoargs.h
echoargs.k
exn.h
exn.k
expr.h
expr.k
fileA
fileB
fileC
fileD
grep.h
grep.k
hello.h
hello.k
help
ln.h
ln.k
ls.h
ls.k
mkdir.h
mkdir.k
more.h
more.k
primes.txt
pwd.h
pwd.k
rm.h
rm.k
rmdir.h
rmdir.k
script.h
script.k
script.sh
setexit.h
setexit.k
sh.h
sh.k
shutdown.h
shutdown.k
stat.h
stat.k
tee.h
tee.k
test-447
test-script
```

```
test.h
test.k
wc.h
wc.k
```

The following files have been modified from the last assignment or are present to make sure your copy is the same as the reference version is using:

```
Main.k
Syscall.h
System.k
UserSystem.h
UserSystem.k
makefile
```

The makefile has been modified to compile the new programs and create a correct sized DISK file. Three new system calls were added to Syscall.h, UserSystem.h and UserSystem.k. Two of these you will implement and one is give to for completeness.

All remaining files are unchanged from the last assignment except you have one more method for your Kernel OpenFile class. Take the file RemoveEntry and add this method to the OpenFile class just after the AddEntry method. You will need to add the method definition to your Kernel.h file.

3 Changes in Kernel.h

In your Kernel.h file, update the INIT_NAME to be TestProgram5. Also, the file "KernelChanges" has a bunch of additions to both Kernel.h and Kernel.k. Not all are required, but if you add all the changes, you can change them later if you elect to do something different. (This refers to the pipe object.) This will be talked about later in this document. Also, make sure your constants in Kernel.h match the following:

```
PAGE_SIZE = 8192                                  -- in hex: 0x0000 2000
PHYSICAL_ADDRESS_OF_FIRST_PAGE_FRAME = 1048576    -- in hex: 0x0010 0000
NUMBER_OF_PHYSICAL_PAGE_FRAMES = 512              -- in hex: 0x0000 0200

MAX_NUMBER_OF_PROCESSES = 20
MAX_STRING_SIZE = 255
MAX_PAGES_PER_VIRT_SPACE = 48
MAX_FILES_PER_PROCESS = 10
MAX_NUMBER_OF_FILE_CONTROL_BLOCKS = 48
MAX_NUMBER_OF_OPEN_FILES = 48
USER_STACK_SIZE_IN_PAGES = 1
NUMBER_OF_ENVIRONMENT_PAGES = 0
```

```
SERIAL_GET_BUFFER_SIZE = 10
SERIAL_PUT_BUFFER_SIZE = 10
```

The final kernel changes are to let the user land programs know that a system call is not implemented. For all the system calls that are listed as extra credit, add the following code as the body of the Handle_Sys_XXXX call. For ones you implemented in the last assignment do not add the following code and if you choose to implement the extra credit in this assignment, you can replace the following code with your implementation.

```
currentThread.myProcess.lastError = E_Not_Imp
return -1
```

4 Work Summary

There are a number of implementation tasks needed to complete this assignment. They are the following:

Serial I/O device driver	– 125 points
Chdir system call	– 10 points
Dup system call	– 15 points
Pipe system call	– 55 points
Link system call	– 20 points (Extra credit)
UnLink system call	– 35 points (Extra credit)
Mkdir system call	– 35 points (Extra credit)
Rmdir system call	– 15 points (Extra credit)

You should implement the Serial driver first. Then the remaining can be done in any order. Doing pipes second can maximize your points.

Note: Please keep track of your time on this assignment. Please keep track of time for each of the 4 jobs separately as best as possible. Include it in your submission. See the "What to Hand In" section later. Also, if you do extra credit items, record the time spent on each of them.

5 Implementing Serial I/O

In this assignment, 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 assignment you will not add any new syscalls to implement serial I/O.

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 assignment, the Open syscall was passed a filename. In this assignment, the behavior of Open will be modified slightly: if the filename argument happens to be the special string "/dev/serial", 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.

When the Close syscall 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 transferred, 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.

5.1 Resources

To help with implementing the terminal, please refer to the following sections from the document titled "The BLITZ Emulator":

- 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. You may also want to review assignment 1 where you read characters from the serial device with the GetCh() function. This assignment uses the same device, but controlling it from KPL instead of assembly.

5.2 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.

5.2.1 Serial Driver

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, DIRECTORY
```

The following should also be there; uncomment it.

```
var
  serialDriver: SerialDriver
```

Add a new class called SerialDriver: (This definition goes in Kernel.h and you will need to add the implementation in Kernel.k. It is not given here.)

```
------------------------  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
    initialized : bool
    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, DIRECTORY, 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. The Main.k distributed with this assignment 7 has it initialized there. (If you do a different implementation, make sure you initialize it correctly in your Main.k.)

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. (You could add helper functions like ReadTerminal() and WriteTerminal() to OpenFile to localize your terminal functionality outside of Handle_Sys_Read and Handle_Sys_Write.)

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".

Index	0	1	2	3	4	5	6	7	8	9
Array			a	b	c					
Variables		out	↑		in	↑				

```
getBufferNextOut = 2
getBufferNextIn  = 5
getBufferSize = 3
```

If the getBufferSize is zero, then GetChar must wait on the condition getCharacterAvail, which will be signaled with a Signal() 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".

Index	0	1	2	3	4	5	6	7	8	9
Array	y	z								x
Variables		in	↑						out	↑

```
putBufferNextOut = 9
putBufferNextIn  = 2
putBufferSize = 3
```

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 put-BufferNextIn 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, use the “initialized” field in the SerialDriver (shown in the above outline). In KPL, global variables are always initialized to their zero values; for a boolean this is “false”. Since the “serialDriver” is a global variable, the “initialized” field is set to “false”. The code in SerialDriver.Init should create the semaphore and initialize it. As the last thing it does, Init should set the “initialized” field to true.

Here is the code for the SerialInterruptHandler function:

```
currentInterruptStatus = DISABLED
if serialDriver.initialized
  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.

5.2.2 Buffer Manipulations

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

```
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
```

5.2.3 Open File Manipulations

Remember that a user program can open "/dev/serial" 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. You have already seen an open directory (the ToyFS root directory and OpenDir) and later we talk about the implementation of pipes, and so there are really four kinds of OpenFiles, but we'll ignore pipes and directories for now.

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

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

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 assignment, 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 "/dev/serial" (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 assignment 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 assignment 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.

If it works better for you, adding helper functions for reading and writing using the terminal will allow the Handle_ functions to delegate the actual work. This also allows the Handle_ functions to check for errors and then call functions to do the actual work.

5.3 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 Read fast enough to prevent the type-ahead buffer from overflowing, but that is a different problem.

5.4 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 assignments, 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 assignment, 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 assignment 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...!)

5.5 Dealing With Newline and Carriage Return

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. For this assignment, I am expecting your code to work in raw mode, but not do all the processing of cooked mode. The shell is written to work well in raw mode.

5.6 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.

UserLib.h and UserLib.k have several utility routines for use by "user land" programs that use "proper" I/O that use system calls. Several of the programs available to run by the shell (see below) use this library to do their jobs. The following are definitions in UserLib.h:

```
const stdin  = 0
      stdout = 1
      EOF = '\xFF'

functions
  ReadLine ( line: String) returns bool
  dReadLine (fd:int, line: String) returns bool
  GetChar () returns char
  dGetChar (fd: int) returns char
  PutChar (c: char)
```

```
dPutChar (fd: int, c: char)
Print (str: String)
dPrint (fd: int, str: String)
PrintInt (num: int)
dPrintInt (fd: int, num: int)
StringToInt (str: String) returns int
IntToString (val: int str: String)
```

6 The Shell Program

After completing the Serial I/0 portion of this assignment, your kernel will have enough functionality to support a Unix-like shell. Specifically, the serial I/O allows the shell run interactively. This assignment has a shell included in the user programs provided. It supports the following features: (Some may require you to complete this assignment, including the extra credit system calls.)

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. If you implemented command line arguments, commands like:

   ```
   % ls -l /bin
   % cat file1
   ```

5. Nested shell invocation, for example

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

   or

```
% sh script.sh > output
```

6. When your kernel can process pipelines, the shell will use them from the command line. For example:

```
% cat < file1 | wc
% cat file1 | grep Hello | wc
```

7. Command expansion:

```
% set x $( expr ${X} + 1 )
```

8. If and While statements:

```
% if true
> echo Yes
> else
> echo No
> end

% set X 1
% while test ${X} -lt 10
> echo ${X}
> set X $( expr ${X} + 1 )
> end
```

9. There are many many other features that just require them to be implemented in the shell but do not require any more kernel support. These include a wildcards (*), "stderr" file, multiple commands on one line, and other standard shell features. Features like inherited environment variables would require the Exec system call to be changed to pass an environment as well as the arguments. Also, a number of file based commands standard in UNIX can be implemented without extra kernel support.

This shell, called sh, will be able to use command line arguments, if implemented, but will not crash if they are not implemented. This shell has "shell variables" and has the following built-in commands:

echo args	- echo the arguments
cd dir	- cd to the requested directory
exit [value]	- exit the shell with the value, no value means 0
set var value	- set the shell variable to the value
unset var	- remove the definition of the variable
true	- set the $? variable to 0
false	- set the $? variable to 1
read var	- set the variable's value by reading a line from stdin

This shell also processes expansions on the command line. They are:

$(command)	- replace with the output of the command
$?	- replace with the exit value of the last command
${name}	- replace with the variable's value

Along with this shell, there are a number of utility programs that are included in the files for this assignment. When the shell tries to Exec a program that does not start with "/", it prepends "/bin/" to the name and tries the Sys_Exec. If that call fails, it then tries to exec the name of the program as given. The utility programs are placed on the DISK by the makefile in the directory /bin. These utility programs are:

cat	concatenate files
chmode	change the mode of a file
cp	copy files
du	calculate the disk usage on a directory tree or file
	flags: -k for Kbytes, -s for summary
df	print disk statistics
echoargs	just echo the command line arguments, numbered and one per line.
exn	a test utility for testing Sys_Exec
expr	calculate expressions, supports integer +, -, *, /, % with (and).
grep	print lines that match a simple pattern
hello	print "Hello World"
ln	link a file to a new name
ls	list file names, flags: -a all, -d directory, -l long
mkdir	make a directory
more	page through a file or standard input
pwd	print the name of the current working directory
rm	remove a file
rmdir	remove a directory
script	run a sub-shell capturing all input and output to a file
setexit	converts the argument to a number and exits with that number
sh	the shell
stat	print the stat information for a file or directory
tee	a pipe fitting
test	evaluate parameters: operators = != < > (strings)
	-ne -eq -lt -le -gt -ge (integers) -a -o ! (boolean)
	file ops: -X name: d (directory) e (exists) f (file) r (readable)
	s (size greater than zero) w (writable) x (executable)
wc	character, word and line count, flags: -c, -w, -l

7 Implementing more System Calls

The assignment to this point is worth 125 points. For the final 100 points, implement code that adds the following system calls to your kernel: (Doing them in this order is the best but not required. If you get stuck on one, try the others. For example, even if you get stuck on Mkdir, you can use the program "toyfs" to make a directory on your "DISK" and then test Rmdir.)

Chdir	- 10 points
Dup	- 10 points
Pipe	- 55 points
Link	- 20 points (Extra)
Unlink	- 35 points (Extra)
Mkdir	- 35 points (Extra)
Rmdir	- 20 points (Extra)
"turn-in"	- 25 points

7.1 Chdir

Chdir is relative easy. After verifying the argument as usual, try to open the directory and get an OpenFile. If you can't open the directory or the OpenFile is not a directory, close the new file if needed and return an error. After no errors may occur, close the working directory and assign the new OpenFile as the current working directory.

7.2 Dup

Dup is a new system call added for assignment 7. The prototype for the user level call is:

```
Sys_Dup  (fd: int) returns int
```

The changes to Syscall.h, UserSystem.h and UserSystem.k are done for you and the new versions are distributed with this assignment. But, in your Kernel.k, you will need to add the code to service the call. Dup should find the first free entry in the file descriptor table, starting at 0, and make a new reference to the original file descriptor in the new location. This call fails only if there are no free file descriptor entries and should return -1. If this call is successful, return the new file descriptor number. The shell depends on this for some of its features. Remember the OpenFile.NewReference method.

7.3 Pipe

The Pipe system call creates a pipe for use by the process. The prototype for the user level call is:

```
Sys_Pipe (fds: ptr to array [2] of int) returns int
```

This syscall will create a new pipe and return the file descriptors which refers to the pipe in the array. If the pipe is successfully created, the system call returns 0. If something goes wrong, the system call returns -1. On success, the file descriptor of the "read end" is put into fds[0] and the file descriptor of the "write end" is put into fds[1].

Pipe has some similarities to the Open syscall. It will need to allocate two new OpenFile objects. The kind of this new OpenFile object will be PIPE. One of the OpenFile objects will have flags contain O_READ and the other one has O_WRITE. (An alternative is to have two file kinds, PIPE_R and PIPE_W, but with the flags you shouldn't have to do that.) You will need to modify the OpenFile class to add a reference to a new Pipe that manages the pipe and has a buffer for pipe data. How many bytes should be in the buffer? Only one byte is really necessary but more will allow greater efficiency for programs using pipes when a producer generates data faster than the consumer is reading. Using a frame is a good idea. You can get a single frame from the FrameManager. On final Close of your Pipe object, you need to return that frame back to the FrameManager. You use PutAFrame() to return the pipe buffer.

For the operation of the pipe, you will need some way to control the concurrency and synchronization between producers (writes on the write end) and consumers (reads on the read end). A writer adds data to the buffer. When the buffer is full, any process trying to write to it must be suspended. If the writer is writing more than can fit in the buffer, it should do a partial write, go to sleep, and then continue the write after it starts running again. This could happen multiple times if it is writing a lot of data, for example more than a PAGE_SIZE of data. A reader takes data out of the buffer and returns it to the user process. It will copy the maximum of the bytes requested and the number of bytes in the buffer. After a copy of the bytes available, the number of bytes transferred is returned to the user. When the buffer is empty, any process trying to read from it must be suspended. When a read completes, it makes room in the buffer and if a writer is suspended, it must restart the writer. When a write to the buffer happens, even if the writer must be suspended because there is not enough room in the buffer to complete the write, a waiting reader must be restarted. Also, readers may need to restart other readers and writers may need to restart other writers.

The semantics of closing a pipe in our file system will be a bit different than in UNIX. Similar to UNIX, if all write ends of a pipe are closed, a Read should return 0 when the buffer is empty. (If the buffer is not empty, the call returns characters from the buffer.) Note, if the last writer closes the write end, all suspended readers need to be restarted so they can return 0 to the user processes. If all read ends are closed, a Write on the pipe returns -1. (We do not have signals in our Blitz OS.) This means that when the last reader closes the read end, all writers that are suspended need to be restarted so they can return a -1 to the user process.

In the file manager, you would then need to add an array of pipes and a method to get a free pipe. (Implement something similar to OpenFile management.) Don't block if you can't get a free pipe, just return an error. The Init would be run once from the FileManager's Init method and the FileManager would need a way to keep track of free pipes. Opening a pipe would then get a pipe from the FileManager and should get a new frame for the buffer and a Close should free that frame. (If you wanted to improve this a bit, you could add an argument to FrameManager.GetAFrame that asks to block or not. Pipe.Open then would not block if a frame was not available, but return a failure to open the pipe.) You will also need to add a method to FileManager to return an unused pipe when both ends have been closed. This return a pipe function should be a non-blocking function. (Most likely, you won't need an FCB for pipes because you have a Pipe object instead. The FCB is directly related to the ToyFS.)

To help you on the pipe, you could implement a pipe object like:

```
class Pipe
    superclass Listable
    fields
        bufferFrame: int  -- Buffer frame, needs to be acquired at open time
        head, tail: int   -- Circular buffer
        pipeMutex: Mutex
        charsInPipe: int
        readQueue: Condition
        writeQueue: Condition
        writer: Condition

    methods
        Init ()
        Open () returns bool
        Read (buffer: ptr to char, sizeInBytes: int) returns int
        Write (buffer: ptr to char, sizeInBytes: int) returns int
        Close ()
  endClass
```

Handle_Sys_Read function should just call the pipe Read function if it is reading from a pipe. It should do a similar thing for Write. All the synchronization for the pipe would then be done in the Pipe object.

If you choose to not implement Pipe or Dup, add the following code in your kernel Syscall-TrapHandler function so you kernel won't have a fatal error with an unknown syscall.

```
        case SYSCALL_PIPE:
          return -1
        case SYSCALL_DUP:
          return  -1
```

7.4 Link (Extra Credit)

The Link system call will need to check for write permission to the "new" directory and then add a new entry in that directory. (This is very similar to file create except you already have the inode.) You need to lookup the inode and get a FCB to be able to modify the inode information. You should increment the link count in the inode and save the inode information.

7.5 Mkdir (Extra Credit)

Mkdir is similar to creating a new file since a new inode will need to be allocated. Then, you need to create the "empty" directory which includes the entries "." and "..". Notice, if created correctly, you can use openfile.AddEntry to add these names. This adds a link to the parent directory and has two links for the newly created directory. The two links are the name of the new directory in the parent directory and the "." entry. And with the ".." entry, you have a new link to the parent directory.

7.6 Unlink (Extra Credit)

Unlink only removes the link in the directory. This should decrement the link count in the inode for the file. (You should only be changing inode information via the FCB class.) In doing this, you will need to get the fcb for the file. Decrement the link count and save the inode.

The complicating feature of adding this system call is that now, it is possible to end up with a link count of 0 in some inode. It is at this point that the space for the file may be reclaimed. When a file has a link count of zero and the file is closed for the last time, it can't be opened again or looked up. We must then delete the file. The includes freeing all data blocks associated with the file. This includes all blocks in the direct block entries (non-zero entries are allocated blocks), all non-zero entries in the indirect block (if there is one) and the indirect block itself. Once all blocks are freed, you then can free the inode.

In review, any time you want to use a ToyFS file, you should have a FCB. Even if two processes open the same file, they should have one FCB. Therefore, if a file is open, there should be a unique FCB for that file with the "number of users" showing how many times it is being used. As long as there is a FCB for the file, the file must not be deleted. Consider the Release method of the FCB class. It currently does:

```
----------  FileControlBlock . Release  ----------

  -- Must be called with fileManagerLock locked.

  method Release (lock: ptr to Mutex)
     numberOfUsers = numberOfUsers - 1
     if numberOfUsers <= 0
        -- Final close, mark unused and release any indirect frame
        relativeSectorInBuffer = -1
        inode.FreeIndirect ()
        inode.number = -1
        fileManager.fcbFreeList.AddToEnd (self)
        fileManager.anFCBBecameFree.Signal ( lock )
     endIf
    endMethod
```

You need to add code to this method to detect that a file needs to be deleted. This would include a directory file if it had zero links. This happens at the "Final close" time.

7.7 Rmdir (Extra Credit)

Rmdir is similar to unlink but works on directories. First, you need to make sure that the directory contains only the two entries, "." and "..". You should be able to detect that by looking at the fsize for the inode. (Note: RemoveEntry is not completely implemented. Both AddEntry and RemoveEntry have been tested only on directories that have at most one data block in the directory. RemoveEntry has a comment where code should exist to delete a block when the last entry has been removed from that block. This should not be a problem for this assignment and you don't need to implement it.) When deleting the directory, remember that you are removing a link to the parent directory and need to decrement the parent's link count. Also, you should update the directory's link count to zero so that the directory inode and all data blocks will be freed by releasing the associated FCB.

8 What to Hand In

The a7 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). 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!) Please collect all the script output into the file named "a7-script".

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, Open-CloseTest, TerminalErrorTest) your output should match the DesiredOutput file. (The DesiredOutput file does not contain any output from the menu or pipe tests.)

Be sure to use the same code to execute all tests. 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 make sure you have an unaltered version of TestProgram5.k

You may notice that TestProgram5 has tests only for the Dup and Pipe system calls. All the other new system calls should be tested using the shell. Chdir can be tested by the shell. Link, Unlink, Mkdir, and Rmdir can be tested by the programs that directly use those system call, specifically, ln, rm, mkdir and rmdir.

The final test after everything is working is a test-447 file found in the directory /testdir. Please run the following commands from your BLITZ shell in the directory /testdir:

```
% script test-output
% sh /testdir/test-447
.... output ...
% exit
```

Please extract the test-output file from your BLITZ disk and commit it to your "a7" branch. You can get a file from your BLITZ disk and save it in a UNIX file using the command:

```
% toyfs -g blitz_name unix_name
```

where “blitz_name” is the actual name of your BLITZ file (test-output in this case) and “unix_name” is what you want the UNIX file name to be.

Please hand in a cover sheet that includes a list of any problems still existing in your code when you turned it in. Your cover sheet should also report the time it took you to do this assignment reported for the eight parts. As usual, create an “a7” branch in your gitlab repository and commit the a7-script and the test-output to the a7 branch.

Final Note:

If you don’t have command line arguments working, get them working for this assignment so you can have a fully functional OS when you finish with this assignment.