CPSC 415
Assignment 3

Due: Monday April 2, 10:00PM
Hard-copy cover sheet and write-up , in class April 4
May be done in groups of two.

1 Motivation

The purpose of this assignment is to develop an understanding of the how the three layers of the device driver interact and how the kernel can signal events to an application. In this assignment you will extend the kernel from the second assignment to implement signal handling and a keyboard device. In doing this you will learn how a device driver is constructed and how the kernel can asynchronously signal events to an application.

2 To Start

Your starting point is the kernel resulting from the second assignment. If you did not complete the assignment, or don't wish to use your kernel, you may use someone else's kernel or the partial solution kernel that is currently available on Blackboard/Vista. (You could also start with an assignment 1 solution to avoid time sharing problems, but in the end it needs to run on the assignment 2 kernel.) You will modify the code in several modules and add additional modules to extend the kernel.

2.1 Signal

The task is to implement a mechanism that allows the kernel to asynchronously signal an application. The signaling system will support 32 signals, numbered 0 to 31. Signals are to be delivered in priority order, with signals of a higher number having a higher priority. For example, if signal 23 is being handled by a process, that signal's processing can be interrupted by signals numbered 24 to 31 and signals 0 to 23 would be held in abeyance until the handling of signal 23 finished. Also, in order to simplify things, the only way a signal can be posted (signaled) is via the syskill() call.

2.1.1 Signal system calls

To `syscall.c` add the following system calls:

```
int syssighandler( int signal, void (*handler)(void *))
```

This call registers the provided function as the handler for the indicated `signal`. If `signal` is invalid or it can be determined that the handler resides at an invalid address then -1 is returned otherwise 0 is returned. The function being registered as the handler takes a single argument. When the handler is called, the trampoline code will set things up so that the handler's parameter will point to the start of the context at the time the kernel decides to deliver a signal to the process. (Note this means actually delivering the signal, not marking it for delivery.) Essentially, this is a pointer to the start of the context that was to become active if a signal was not going to be delivered. If a null pointer is passed in as the handler then signal delivery for the identified signal is disabled, which means the signal is ignored. The default action for all signals is to ignore the signal.

```
void sigreturn(void *old_sp)
```

This is to be used only by the signal trampoline code. It takes as an argument the location, in the application stack, of the context frame to switch this process to. Typically, this is the value that was passed to the trampoline code. On the kernel side, the action performed by this call is to replace the stored stack pointer for the process with the pointer passed as the parameter to this system call, to adjust the mask indicating what signals are being accepted, and to recover any return code pushed onto the stack. This call does not return.

```
int syskill(int PID, int signalNumber)
```

This system call requests that a signal be delivered to a process. The `PID` argument is the process ID of the process to deliver the signal to. The `signalNumber` is the number of the signal to be delivered (i.e. 0 to 31). On success this call returns 0, if the target process does not exist then -15 is returned and if the signal number is invalid -1 is returned.

```
int syssigwait(void)
```

This system call simply causes the process to be suspended until a signal is delivered to it. The dispatcher will mark this processes as blocked (if your kernel keeps track of the process state) and make sure that the process is not put on the ready queue. The only way the process will return to the ready queue is when the process becomes the target of a signal. Once a signal is delivered, this call completes and returns 1.

Keep in mind that the above calls will have some corresponding code in the dispatcher and it is your responsibility to determine what it needs to be.

2.1.2 Signal processing Code

In the file `signal.c` implement the function

```
sigtramp(void (*handler)(void *), void *cntx, void *osp)
```

When the kernel decides to deliver a signal to a process, the kernel modifies the application's stack so that upon a the context switch the sigtramp() code is executed. The sigtramp() codes runs in user space as part of the application and is used to control the signal processing in the application. When executed it is the responsibility of sigtramp() to call the provided handler. When the handler returns, sigtramp() calls `sigreturn()` to complete the signal processing. The `sigreturn()` call never returns.

```
int signal(int pid, int sig_no))
```

This function, implemented in signal.c, is internal to the kernel, and is called when a signal is to be delivered to a process. The function takes as arguments the PID of the process to deliver the signal to and the signal to deliver. The function returns -1 if the PID is invalid, -2 if the signal is invalid and 0 otherwise. If a process is blocked on a system call when it is targeted to receive a signal then that system call is unblocked and returns an error indication to the application, **after** the signal processing finishes. (i.e. When blocked on a system call if a signal goes off the established signal handler for that call is run before the call returns the error. Unless otherwise specified use -32 as the return code for a system call interrupted by a signal.)

To be able to deliver signals in priority order it is suggested that you extend each PCB entry with a field to indicate that a signal or signals are pending (Hint use a bit mask with each bit in the mask corresponding to a signal number). Then, each time a process is dispatched this field is checked to see if there are pending signals. The number of a signal indicates its priority with 0 being the lowest priority. If a signal is pending then a "signal frame" is put onto the stack before doing the actual context switch.

With respect to a specific process, if a higher priority signal occurs while processing a signal then the current signal processing is suspended and the new signal handler run. When the higher priority handler completes, execution returns to the point where the lower priority signal handler was interrupted. If a signal of the **same** or lower priority occurs while a signal handler is executing then that signal is deferred until completion of this handler and all higher priority handlers. Signals that are to be ignored because there is no handler never get recorded for delivery. If, before a process gets a chance to process a specific signal, the same signal is delivered the signal handler for that signal is run only once.

2.1.3 Signal handling implementation suggestions

To deal with the signal prioritization issue you might want to maintain three sets of bit masks. The first, as indicated above is a bit mask that records all of the signals currently targeted to the process. The second is a mask with 1s in the locations of all the bits of the signals we are willing to accept (i.e. a handler is installed for that signal) and the third is a mask with 0s in the bits corresponding to the signals being ignored because the priority of the corresponding signal is too low and 1s everywhere else.

The 2nd mask can be used to determine if a new signal should be recorded. This can be done by and'ing the new signal, represented as an integer with the bit corresponding to its signal number turned on, with the 2nd mask and or'ing the result into the first mask. Then, as a simple, first approximation, if the kernel wants to determine if there is a signal to deliver it ands the first and third mask together and if the value is non-zero a signal needs to be delivered.

As part of delivering the signal the kernel will need to save the current value of the 3rd mask and then change the 3rd mask so that only signals that have a higher priority than the signal being delivered are visible. This mask needs to be saved because when the signal processing for the current signal is finished the old mask will need to be restored. Since a signal can occur while processing a signal this third mask may need to be saved multiple times. It will be the responsibility of the sigreturn() call to restore the third mask to its old value.

One problem is where to store the "old" mask. You might want to consider putting the old mask on the application stack, just before the context for the trampoline code is added. (i.e. Put

it on the stack as the 4th parameter for sigtramp. Sigtramp won't use it, but when sigreturn() is called the kernel can compute where the mask was stored and retrieve it. (Suggestion: Consider defining a struct that can be used to setup the stack for the switch to sigtramp and then having the appropriately defined fields in it such as the new context, return address, arguments to sigtramp() and then any extra fields.)

Also, you have to deal properly with the return code of system calls that are interrupted due to signals paying particular attention to what needs to happen if one signal interrupts another. Basically the return value needs to be saved so that it can be recovered later. It is suggested that you use the same strategy outlined above for the 3rd mask.

2.2 Device Driver

You are to implement the *keyboard* device using the standard design pattern discussed in class.

2.2.1 System Calls

To `syscall.c` and `disp.c` add the following system calls:

```
extern int sysopen(int device_no)
```

This call will be used to open a device. The argument passed in is the major device number and can be used to index, perhaps after some adjustment, into the device table. The call returns -1 if the open fails, and a file descriptor in the range 0 to 3 (inclusive) if it succeeds.

```
extern int sysclose(int fd)
```

This call takes as an argument, the file descriptor (`fd`) from a previous open call, and closes that descriptor. Subsequent system calls that make use of the file descriptor return a failure. The call returns 0 for success and -1 for a failure.

```
extern int syswrite(int fd, void *buff, int bufflen)
```

This call performs a write operation to the device associated with the provided file descriptor, *fd*. Up to *bufflen* bytes are written from *buff*. The call returns -1 if there is an error, otherwise it returns the number of bytes written. Depending upon the device, the number of bytes written may be less than *bufflen*.

```
extern int sysread(int fd, void *buff, int bufflen)
```

This call reads up to *bufflen* bytes from the previously opened device associated with *fd* into the buffer area pointed to by *buff*. The call returns -1 if there is an error, otherwise it returns the number of bytes read. Depending upon the device, the number of bytes read may be less than *bufflen*. A 0 is returned to indicate end-of-file (EOF).

```
extern int sysioctl(int fd, unsigned long command, ...)
```

This call takes a file descriptor and executes the specified control command. The action taken is device specific and depends upon the control command. Additional parameters are device specific. The call returns -1 if there is an error and 0 otherwise.

2.2.2 Device Independent Calls

Each application level system call requires a corresponding call to be invoked by the dispatcher. The calls that you will need to implement are: `di_open()`, `di_close()`, `di_write()`, `di_read()` and `di_ioctl()`. The parameters to these calls are just the parameters of the corresponding system call along with any additional parameters, it any, required by your design. The return values of these calls and their meaning are design dependent. (Recall that the return values are only used by the dispatcher and do not represent the value that the system call returns to the application.) These calls should be implemented in the file called `di_calls.c`

Except for the `di_open()` call, the remaining calls all follow the same implementation pattern. Roughly each DII call does the following:

- Verifies that the passed in file descriptor is in a valid range and identifies an opened device.
- Using the information stored in the file descriptor table of the process determines the the index of the appropriate device in the device table from the there determines the function to call.
- Calls the function.
- You will need to determine the meaning of the return value of the DII call.

We discussed in class the types of things that `di_open()` needs to perform. However, we don't have a file system to perform the name to major device number mapping. To get around this problem have the `di_open()` call take the actual major device number as the argument to identify the device being opened. The `di_open()` call will need to verify that the major number is in the valid range before calling the device specific `open()` function pointed to by the device block and adding the entry to the file descriptor table in the PCB.

2.2.3 PCB Changes

You will need to add a file descriptor table to your PCB. The file descriptor table must allow 4 devices to be opened at once. Each entry in the file descriptor table must somehow (this is up to you) identify the device associated with the descriptor.

2.3 Device Table and Device Structure

Your device table will not be very interesting in that it will have only 2 devices in it. Both entries are for the keyboard. The device 0 version of the keyboard will **not** echo the characters as they arrive. This means that if the characters need to be display the application will have to do it. Device 1 will echo the character. This means that the character could be displayed before the application has actually read the character. Even though these are separate devices, only one of them is allowed to be open at a time.

Each table entry has a device structure similar to that described in class. (i.e. It need not be exactly as described in class.) But, the structure proposed in class serves only as a guide. Keep in mind that the structure you choose must be capable of supporting a wide range of both physical and virtual devices. Also note that the keyboard devices have basically the same functionality except for one routine so do not duplicate code.

The required tables and structures should be defined in kernel.h and initialized in init.c

2.4 Keyboard

For each of the device independent calls you will need to implement a corresponding device specific device driver call. The implementation is to be in the files named `kbd.c` and `kbd.h`.

Since writes are not supported to the keyboard, all calls to `syswrite()` will result in an error indication (-1) being returned.

The behaviour of the sysread() system call is somewhat simpler and less flexible than might be found on a typical Unix system.

If the echo version of the keyboard is opened, then each typed character is echoed to the screen (You can use `kprintf()` to do this, but `kputc()` is probably easier, see the kprintf.c source file) as soon as it arrives. If the non-echo version of the keyboard is being used the responsibility for printing characters, if required, is the application's.

The keyboard implementation is to internally buffer up to 4 characters. If a control-d is typed the underlying driver will take that to mean that no more input follows and return, at the appropriate time, an end-of-file (EOF) indication. A return value of 0 (zero) by sysread() will indicate EOF. Subsequent sysread() operations on this descriptor will continue to return the EOF indication. If the buffer fills up subsequent character arrivals are discarded and not displayed. If a control-d is received while the buffer is full it is discarded like any other character.

When a read call is made it **blocks** until one or more of the following conditions occurs while copying data from the kernel to the application buffer passed as the parameter to sysread():

- The buffer passed in to the read system call is full as specified by the size parameter to the read call. Any unread bytes in the kernel buffer are returned on subsequent reads.
- While copying bytes to the application buffer the "Enter key" (i.e. the carriage return character) is encountered. The ASCII character for a new line ("\n") is put into the buffer and the call returns. The "\n"character is included in the count of the number of characters returned. Characters after the "Enter Key" are returned in subsequent read operations.
- An EOF was detected by the kernel and all unread data has now been copied.

Observe that when filling the application's read buffer, characters are always consumed first from the kernel's read buffer followed by new characters from the keyboard if there are insufficient characters in the in the kernel's buffer. When read operations succeed the bytes returned must correspond to the legal ASCII characters as defined by *man ASCII* on the undergraduate Linux machines.

The only *sysioctl()* command supported by the keyboard device is one to change what character typed at the keyboard indicates an EOF. The command number to request this operation is 49, (see http://www.random.org for how this number was selected.) and the third parameter is the integer value of the character that is to become the new EOF indicator.

2.4.1 Interacting with the keyboard hardware

The chip that communicates with the keyboard is the Intel 8042 and it occupies ports 0x60 to 0x6F, although we won't use all the ports. Port 0x60 is where data is read from. Commands and control information are read/written to port 0x64. To see if there is data present read a byte from port 0x64. If the low order bit is 1 then there is data ready to be read from port 0x60. Links to documents to help you understand the 8042 and how to program it are on the page announcing

the release of this assignment. The keyboard is the 2nd device connected to the PIC/APIC. Since devices are numbered starting with 0, the IRQ for the keyboard controller is 1. Consequently, interrupts for the keyboard controller are enabled with the following call: `enable_irq(1, 0)`. To avoid unneeded processing the kernel should enable the keyboard interrupts through the APIC only upon an open and disable them again on a close. You will need to read the code to determine how to disable the keyboard. I'd suggest starting with `enable_irq()`. Also, don't forget to install your ISR.

When a character is read from the 8042 you actually get back a scan code representing which key was pressed or released. (You get an event when the key is pressed and another event when the key is released.) This means that you will need to convert the scan codes to ASCII and you will also need to keep track of whether the control and shift keys are still being held when subsequent scan codes arrive. Note: keys outside the standard part of the keyboard (e.g. the function keys, keypad, arrow keys etc, can be ignored. The escape key is considered a standard key.) Before getting too involved in this code, you should experiment a bit by having the keyboard interrupt routine simply print the byte read from port 0x60. This will allow you to determine just what happens when a key is pressed/released and how the control and shift keys fit into this. Some code to convert scan codes to ASCII has been provided on the announcement page as well. You may find the code, which doesn't have a lot of comments, to be of some help. Feel free to use the code and modify it for your own purposes. If you choose to use this code, and make changes to it, be sure to clearly identify and document the changes you make.

2.5 A test Program

Write a root process that

1. Prints a greeting.
2. Opens the "echo" keyboard
3. Reads characters, one at a time until 10 characters are read.
4. Attempts to open the "no echo" keyboard
5. Attempts to open the "echo" keyboard
6. Closes keyboard
7. Opens the "no echo"
8. Does three reads of 10 characters and has the application print them (Each typed line is to be at least 15 characters long.)
9. Continues reading characters until an EOF indication is received.
10. Closes keyboard and opens the "echo" keyboard
11. Installs a signal handler for signal 18 that simply prints that it was called.
12. Creates a new process that sleeps for 100 milliseconds and then sends a signal 20 to the root process followed by signal 18.

13. The root process does another read (don't type anything)
14. The read should be interrupted by the signal and get back an error, print out a message to indicate this.
15. Read input until an EOF indication is returned.
16. Attempt to read again and print the result that read returns
17. Print termination message and exit.

In the above be sure to include appropriate print statements to clearly illustrate what is happening.

2.6 Hints to Successfully Completing the Assignment

Here are some hints on how to successfully complete this assignment.

1. Implement the parts you understand first. Do not assume that the best way to implement this assignment is start at the beginning of the assignment and work your way through.
2. To develop some expertise in interacting with the 8042 just try reading from it and printing something out before writing other parts of the code.
3. Write small bits of code and test it right away. **Do not** write the whole assignment and then try to get it working.

3 Testing

You are to produce a testing document organized into sections. Each section will cover one testing scenario, have a title, and be clearly delineated with an obvious separator from adjacent sections. For each test case explain what is being tested, how the test is structured, and how the output demonstrates what is being tested. The test description and interpretation is to be followed by the test output. In some circumstances it may be appropriate to annotate the output to make it clear as to what happened. Annotations, if done neatly, can be done by hand if you don't want to use things like text balloons or other annotation techniques that word processors typically have. Be very careful to distinguish between annotations and actual output. It is acceptable to remove output that is superfluous as long as that is indicated.

1. Test showing prioritization of signals
2. sighandler() test case
3. sigkill() test case
4. syssigwait() test case
5. sysopen() with invalid arguments
6. syswrite() with invalid file descriptor

7. sysioctl() test for invalid commands
8. sysread() when there are more characters buffered in kernel than the read requests
9. Two test cases for scenarios not covered here or in the test program.

4 How the TA's will Test Your Code

To test your code, the TA's will compile and run your code as is, and verify that the test program works. Secondly they may provide their own user land code to test the signal handling and devices.

5 What to Hand In

You need only submit one submission per group.

Be sure to include in the compile directory a working makefile.

Also submit a file called **whoarewe.txt** containing the names, and student numbers of the group members. If your program has any problems or things that the TAs should know about put that information in this file as well.

To submit, copy the files into the subdirectory `~/cs415/as3` and run the handin script:

```
handin cs415 a3
```

See the man page for `handin` for additional information.

Submit the following on paper:

1. A filled in copy of the cover sheet (provided at the end of this assignment) **Make sure it is signed!**
2. The testing documentation as described in the *Testing* section.

All written materials are to be handed in at the start of class on April 4th. Assignments without a handed in cover page will not be marked.

CPSC 415: Assignment 3

Student Name	Student Number	Student Signature

Kernel Used:

	Mark
Functionality/Design	/30
Signaling System	/10
Keyboard	/15
Test Program	/5
Code Documentation	/15
Code clarity	/10
Function descriptions	/5
Test Cases	/20
Total	**/65**