Phase 4: User Processes and System Calls
Complete program due: Friday, November 13th, 9:00 pm

1.0 Introduction

The kernel constructed in Phases 1 & 2 provides valuable system services to the levels above it, but is inappropriate for direct use by user processes. For example, if a user process calls **`waitDevice`** directly, there is no guarantee that the process will get the interrupt associated with its I/O operation, rather than one intended for another process. This is why the kernel specification required that a process be in kernel mode before calling its procedures.

For this phase, you will continue the implementation of the support level that was begun in phase 3. All of the services in phase 4 will be requested by user programs through the syscall interface. For this phase, the support level will contain system driver processes to handle the pseudo-clock, terminal input, and disk I/O. These system processes run in kernel mode and with interrupts enabled.

You may use the libraries that you constructed in Phases 1, 2, and 3. I will also make libraries available that I think are correct. To use this, link with *libpatrickphase1.a*, *libpatrickphase2.a* and *libpatrickphase3.a* in */home/cs452/fall15/lib*.

Recall that your phase 3 started running a process named **`start3`** after it had completed initialization. Thus, you should name the code that is to receive control to initialize this level **`start3`**. Note that **`start3`** needs to run in kernel mode, not user mode; if you are using your own phase 3 library, you will need to make this change to use **`fork1`** rather than **`spawnReal`**.

The header file *phase4.h* can be found in */home/cs452/fall15/include*. You should include this header file in all of your C files.

2.0 Device Drivers

You need to write drivers for three types of devices for this phase of the project. These device drivers are kernel processes that run at priority 2 with interrupts enabled. Typically device drivers would run at the highest priority -- using priority 2 will allow you (and us) to experiment with running other processes at higher priority than the device drivers. You must use semaphores and/or mailboxes to provide mutual exclusion and synchronization between the device drivers and other processes in the system. The device drivers use **`waitDevice`** (from phase1) to wait for interrupts to occur, and **`USLOSS_DeviceOutput`** (see the USLOSS manual) to write the device control and status registers. The interface routines for the drivers may only be invoked in kernel mode. The prototypes shown in this section are suggestions only; you may use them, modify them, or use something completely different. The naming convention used here follows what Patrick suggested in phase 3. The system calls (see section 4 below) use an initial capital letter; i.e., **`Sleep`**. The function name given to **`systemCallVec`** would then use the same name with a lower case initial letter; i.e., **`sleep`**. This would be the function that takes a **`systemArgs *`** parameter. This function would extract the needed values from the **`systemArgs`** structure, and then call the corresponding **`Real`** function; i.e., **`sleepReal`**.

Syscall interfaces are provided that allow user processes to interact with the driver processes; see section 4.

2.1 Clock Driver

The clock device driver is a process that is responsible for implementing a general delay facility. This allows a process to sleep for a specified period of time, after which the clock driver process will make it runnable again. The clock driver process keeps track of time using **`waitDevice`**. Since the kernel controls low-level scheduling decisions, all the clock driver can ensure is that a sleeping process is not made runnable until the specified time has expired.

The interface to the clock driver process is through the following routine:

`int sleepReal(int seconds)`

> Causes the calling process to become unrunnable for at least the specified number of seconds, and not significantly longer. The seconds must be non-negative.
>
> **Return values**:
>
> 1: **`seconds`** is not valid
> 0: otherwise

2.2 Disk Driver

The disk driver is responsible for reading and writing sectors to the disk. The driver uses the circular scan seek optimization algorithm to improve disk utilization. This means that the driver queues several requests simultaneously, rather than simply handling one request at a time.

The interface to the disk device driver is through the following routines. These routines are synchronous, meaning that they should not return until the requested disk transfer has been completed.

```
int diskRealReal(int unit, int track, int first, int sectors,
                 void *buffer)
```

> Reads sectors **`sectors`** from the disk indicated by **`unit`**, starting at track **`track`** and sector **`first`**. The sectors are copied into **`buffer`**. Your driver must handle a range of sectors specified by **`first`** and **`sectors`** that spans a track boundary (after reading the last sector in a track it should read the first sector in the next track). A file cannot wrap around the end of the disk.
>
> **Return values**:
>
> -1: invalid parameters
> 0: sectors were read successfully
> >0: disk's status register

```
int diskWriteReal(int unit, int track, int first, int sectors,
                  void *buffer)
```

> Writes sectors **`sectors`** to the disk indicated by **`unit`**, starting at track **`track`** and sector **`first`**. The contents of the sectors are read from **`buffer`**. Like **`diskRead`**, your driver

must handle a range of sectors specified by **`first`** and **`sectors`** that spans a track boundary. A file cannot wrap around the end of the disk.

Return values:

-1: invalid parameters
0: sectors were written successfully
>0: disk's status register

`int diskSizeReal(int unit, int *sector, int *track, int *disk)`

Returns information about the size of the disk indicated by **`unit`**. The **`sector`** parameter is filled in with the number of bytes in a sector, **`track`** with the number of sectors in a track, and **`disk`** with the number of tracks in the disk.

Return values:

-1: invalid parameters
0: disk size parameters returned successfully

2.3 Terminal Driver

The terminal drivers are responsible for managing the terminals. Each terminal has its own terminal driver process(es). User processes may read and write lines of text (strings that end with '**`\n`**') to and from the terminals. The terminal drivers are responsible for buffering terminal input and output, and for providing mutually exclusive access when a process reads or writes a line (e.g., writing a line to a terminal is an atomic action). The maximum length of a line, including the terminating newline, is **`MAXLINE`** (defined in *phase4.h*).

The interface to each terminal device driver is through the following routines:

`int termReadReal(int unit, int size, char *buffer)`

This routine reads a line of text from the terminal indicated by **`unit`** into the buffer pointed to by **`buffer`**. A line of text is terminated by a newline character ('**`\n`**'), which is copied into the buffer along with the other characters in the line. If the length of a line of input is greater than the value of the **`size`** parameter, then the first **`size`** characters are returned and the rest discarded.

The terminal device driver should maintain a fixed-size buffer of 10 lines to store characters read prior to an invocation of **`termRead`** (i.e. a read-ahead buffer). Characters should be discarded if the read-ahead buffer overflows.

Return values:

-1: invalid parameters
>0: number of characters read

`int termWriteReal(int unit, int size, char *text)`

This routine writes **`size`** characters — a line of text pointed to by **`text`** to the terminal indicated by **`unit`**. A newline is not automatically appended, so if one is needed it must

be included in the text to be written. This routine should not return until the text has been written to the terminal.

Return values:

-1: invalid parameters
>0: number of characters written

As a helpful hint, you may want to consider implementing a terminal driver using more than one process. Sending characters to a terminal and receiving characters from a terminal are independent operations that can happen simultaneously, a natural situation for using separate processes.

3.0 Zap/Quit Handling

In the kernel, a process could be zapped by another process, indicating that it should quit as soon as possible. The device drivers in the support level must follow this convention; if they are zapped, they should clean up their state and call **`quit`**. The device driver interface routines (e.g. **`sleep`**, **`diskRead`**, **`diskWrite`**, **`diskSize`**, **`termRead`**, **`termWrite`**) should all simply return prematurely if the process calling them was zapped while waiting. The I/O routines will report fewer bytes read and written than asked for, and the **`sleep`** routine will simply return too soon.

4.0 System Calls

The following system calls are supported. Executing a syscall causes control to be transferred to the kernel SYSCALL handler, which in earlier phases terminated USLOSS for undefined syscall's. You must now extend the system call vector to include these syscall's. System calls execute with interrupts enabled. You will need to use mailboxes and/or semaphores to provide appropriate exclusion for shared data structures.

USLOSS allows only a single argument to a system call, and no return value. Therefore, all communication between a user program and the support layer will go through a single structure; the address of this structure is the argument to the syscall. The structure is defined as follows in */home/cs452/fall15/include/phase2.h*:

```
typedef struct systemArgs {
      int number;
      void *arg1, *arg2, *arg3, *arg4, *arg5
} systemArgs;
```

An interface that makes these calls look more like regular procedure calls is provided in */home/cs452/fall15/phase4/libuser.c* This file will make it easier for you to write test programs.

A user process that attempts to invoke an undefined system call should be terminated using the equivalent of the **`Terminate`** syscall (syscall 3). Code to handle this was part of phase 3.

4.1 Sleep (syscall SYS_SLEEP)

Delays the calling process for the specified number of seconds (sleep).

Input

arg1: number of seconds to delay the process.

Output

arg4: -1 if illegal values are given as input; 0 otherwise.

4.2 DiskRead (syscall SYS_DISKREAD)

Reads one or more sectors from a disk (**`diskRead`**).

Input

arg1: the memory address to which to transfer
arg2: number of sectors to read
arg3: the starting disk track number
arg4: the starting disk sector number
arg5: the unit number of the disk from which to read

Output

arg1: 0 if transfer was successful; the disk status register otherwise.
arg4: -1 if illegal values are given as input; 0 otherwise.

The arg4 result is only set to -1 if any of the input parameters are obviously invalid, e.g. the starting sector is negative.

4.3 DiskWrite (syscall SYS_DISKWRITE)

Writes one or more sectors to the disk (**`diskWrite`**).

Input

arg1: the memory address from which to transfer.
arg2: number of sectors to write .
arg3: the starting disk track number.
arg4: the starting disk sector number.
arg5: the unit number of the disk to write.

Output

arg1: 0 if transfer was successful; the disk status register otherwise.
arg4: -1 if illegal values are given as input; 0 otherwise.

The arg4 result is only set to -1 if any of the input parameters are obviously invalid, e.g. the starting sector is negative.

4.4 DiskSize (syscall SYS_DISKSIZE)

Returns information about the size of the disk (**`diskSize`**).

Input

arg1: the unit number of the disk

Output

arg1: size of a sector, in bytes
arg2: number of sectors in a track
arg3: number of tracks in the disk
arg4: -1 if illegal values are given as input; 0 otherwise.

4.5 TermRead (syscall SYS_TERMREAD)

Read a line from a terminal (**`termRead`**).

Input

arg1: address of the user's line buffer.
arg2: maximum size of the buffer.
arg3: the unit number of the terminal from which to read.

Output

arg2: number of characters read.
arg4: -1 if illegal values are given as input; 0 otherwise.

4.6 TermWrite (syscall SYS_TERMWRITE)

Write a line to a terminal (**`termWrite`**).

Input

arg1: address of the user's line buffer.
arg2: number of characters to write.
arg3: the unit number of the terminal to which to write.

Output

arg2: number of characters written.
arg4: -1 if illegal values are given as input; 0 otherwise.

5.0 Phase 1, 2 and 3 Functions

In general, in a layered design such as this one, you should not have to refer to global variables or data structures from a previous phase. For phase 4, you can make use of any of the phase 2 functions.

For phase 4, you can use some (not all) of the phase 1 functions. In particular, you can make use of: **`fork1`**, **`join`**, **`quit`**, **`zap`**, **`isZapped`**, **`getpid`**, **`readtime`**, and **`dump_processes`**. You can not use: **`blockMe`**, **`unblockProc`**, **`readCurStartTime`**, and **`timeSlice`**.

For phase 4, you can make use of the phase 3 functions. I will provide a file: */home/cs452/fall15/include/provided_prototypes.h* that will contain the function prototypes for the **`Real`** functions from my phase 3 solution. These are the kernel mode versions of these functions. In particular, this will make it possible for you to use semaphores as well as mailboxes for synchronization purposes in phase 4.

You can not disable interrupts in phase 4. Instead, use semaphores or mailboxes for mutual exclusion when necessary.

6.0 Initial Startup

After your phase 4 has completed initializing the necessary data structures, it should spawn a user-mode process running **`start4`**. This initial user process should be allocated **`2 * USLOSS_MIN_STACK`** of stack space, and should run at priority three. **`start3`** should then wait

for **start4** to finish; **start3** will then take care of stopping all the phase 4 driver processes before quitting.

7.0 Phase 1, 2 and 3 Libraries

We will grade your phase 4 using the phase 1, 2, and 3 libraries that we supply. The provided libraries will be named: *libpatrickphase1.a*, *libpatrickphase2.a*, and *libpatrickphase3.a*. These libraries will be available in */home/cs452/fall15/lib*. For possible help in debugging, there will also be available debug versions of each library.

You may use either your earlier libraries or the ones the we supply while developing your phase 4. However, it will be graded using the phase 1, 2 and 3 libraries that we will supply.

8.0 Submitting Phase 4 for Grading

Note: Programs are submitted via D2L only.

Before submitting to D2L, you will need to create a **phase4.tgz** file. There is a target in the provided phase1 **Makefile** (*/home/cs452/fall15/phase4/Makefile*) named **submit**. It is at the very end of the **Makefile**.

Doing make submit will create a **phase4.tgz** file that contains:

- The **.c** files for your phase4. This will include **phase4.c**, **p1.c**, and any other **.c** files that you have created. If you have created additional **.c** files, you should have already modified **COBJS** to include these additional **.o** files.
- The **.h** files for your phase1. This will include **driver.h**. If you have created other **.h** files, you should have already modified **HDRS** to include the additional **.h** file(s).
- The **Makefile**.

You will *not* submit the usloss library or directory. You will *not* submit the phase 1 library or the phase 2 library. We will put a link for usloss and the phase1 and 2 libraries into the grading directory after we extract your files. The same is true for the **testcases** directory — do not submit the testcases. We will add links for the **testcases** and **testResults** directories.

You are encouraged to create the **phase4.tgz** file, then copy it to a different area of your home directory, extract the file using **tar xvfz phase4.tgz** and do a compile to see if everything has been included.

Any file(s) that are missing from your submission that we have to request from you will result in a reduction of your grade!

To submit the **phase4.tgz** file to D2L

- Log on to D2L, and select CSc 452.
- Select "Dropbox".
- You will find *Phase4*, and *Phase4-late*. Only one of these will be active. Click on the active one — will be *Phase4* if you are turning in the work on time.
- You should now be at the page: "Submit Files - Phase4". Click on the "Add a File" button. A pop-up window will appear. Click on the "Choose File" button. Use the file browser that will appear to select your file(s). Click on the "Upload" button in the lower-right corner of the pop-up window.

- You are now back at "Submit Files - Phase4". Click on "Upload" in the lower-right of this window. You should now be at the "File Upload Results" page, and should see the message **`File Submission Successful`**.

D2L will send you a confirmation email after each item is placed in the Dropbox. This email is sent to your UA email (which is <yourUANetId>@email.arizona.edu). Retain these emails at least until you have received a grade for the assignment; it is your confirmation that you did submit the program(s).

You may submit phase4 more than once. We will grade the last one that you put in the Dropbox.

9.0 Groups

As advertised, you can change groups at this time is you wish. If you choose to do so, send mail to "452fall15@cs.arizona.edu" giving us details of how groups are being changed. You may use Piazza if you wish to advertise that you are looking for a new partner.