Phase 4: Device Drivers and System Calls

1 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 another process's. Therefore, the kernel specification requires 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 the previous phase. All the services in this phase 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 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. It is suggested, however, to use the provided libraries for the previous phases.

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, with priority 3. If you are not using the provided libraries and using your own phase 3 library, which spawns a user process running **start3**, you will need to adjust your code for this change.

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

2 Device Drivers

You need to implement *two* device drivers for this phase of the project. The device drivers are kernel processes that run at priority **2** with interrupts enabled. Using priority 2 will allow you 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 phase 2) to wait for interrupts to occur, and **device_output** (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 I suggested in phase 3. The system calls in Section 4 use an initial capital letter: i.e., **Sleep**. The function name given to **sys_vec** would then use the same name with a lower case initial letter; e.g., **sleep**. This would be the function that takes a **sysargs *** parameter. This function would extract the needed values from the **sysargs** structure, and then call the corresponding **real** function; e.g., **sleep real**.

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.

2.2 Disk Driver

The disk driver is responsible for reading and writing sectors to the disk. Your implementation needs to queue several requests from user processes simultaneously, rather than simply handling one request at a time. (Note that you may choose to implement any disk scheduling algorithm in your implementation to improve disk utilization. Or just simply queue the requests.)

3 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**.

4 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 **usloss/include/phase2.h**:

```
typedef struct sysargs
{
      int number;
      void *arg1;
      void *arg2;
      void *arg3;
```

```
        void *arg4;

        void *arg5;

    } sysargs;
```

An interface that makes these calls look more like regular procedure calls is provided in **libuser.c**. You will need to translate it to **libuser.o** when making a runnable test case.

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

Sleep

Delays the calling process for the specified number of seconds. Use the equivalent of the **Terminate** syscall to terminate the process if it is zapped while asleep.

Input.

Output

Create a user-level process. Use fork1 to create the process, then change it to user-mode. If the spawned function returns, it should have the same effect as calling **Terminate**.

Input:

arg1: number of seconds to delay the process.

Output:

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

DiskRead

Reads one or more sectors from a disk. Use the equivalent of the **Terminate** syscall to terminate the process if it is zapped while waiting for the completion of the disk read operation.

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.

`DiskWrite`

Writes one or more sectors to the disk. Use the equivalent of the **Terminate** syscall to terminate the process if it is zapped while waiting for the completion of the disk write operation.

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 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.

`DiskSize`

Returns information about the size of the disk. Use the equivalent of the **Terminate** syscall to terminate the process if it is zapped while waiting for the completion of the disk size operation.

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.

5 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: **MboxCreate, MboxRelease, MboxSend, MboxReceive, MboxCondSend, MboxCondReceive,** and **waitdevice**.

For phase 4, you can use some (not all) of the phase 1 functions. Specifically, you can make use of: **fork1, join, quit, zap, is zapped, getpid, and dump processes**. You cannot use: block_me, unblock_proc, read_cur_start_time, and time_slice.

For **phase 4**, you can make use of the phase 3 functions. The file: provided_prototypes.h, which should be included in usloss/include, contains the function prototypes for the _real functions from **phase 3**. 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 cannot disable interrupts in **phase 4**. Instead, use semaphores or mailboxes for mutual exclusion when necessary.

6 Initial Startup

In phase 4 you should implement the routine **start3** that first initializes the necessary data structures. **start3** then should spawn a user-mode process running **start4**. This initial user process should be allocated 8 * *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 Phase 1, 2 and 3 Libraries

I will grade your phase 4 using the phase 1, 2, and 3 libraries that I supply. The provided libraries **liblxuphase1.a**, **liblxuphase2.a**, and **liblxuphase3.a** can be stored in your phase4 working directory. You may use either your earlier libraries or the ones the I supply while developing your phase 4. However, it will be graded using the phase 1, 2 and 3 libraries that I supply.

8 Submitting Phase 4 for Grading

For the turnin, you will need to submit all the files that make up your phase 4. Design and implementation will be considered, so make sure your code contains insightful comments, variables and functions have reasonable names, no hard-coded constants, etc. You need to turn in a **README** file to help me understand your solution. Your **README** file should include
information as below:

- Your group members and working system you used to develop your phase 4
- If it requires special handling to run your phase 4 code with test cases, please describe.
- For each test case, provide the output as well as a self-evaluation whether the test case is passed or not. For special case, you may argue that the test case is partially parsed.

Your **Makefile** must be arranged so that typing ‘make’ in your directory will create an archive named **libphase4.a**. Your Makefile must also have a target called ‘clean’ which will delete any generated files, e.g. .o files, .a files, core files, etc.