CS 452 (Fall 22): Operating Systems

Phase 4 - Device Drivers

due at 5pm, Thu 17 Nov 2022

NOTE NOTE NOTE

Students should read the **USLOSS Supplements** that I have provided before they read this spec. This document assumes that you already understand how to use both types of devices. (The clock is simple enough that it did not need a supplement.)

1 Phase 4 Overview

In this Phase, you will implement "device drivers" for the clock, terminal, and disk devices, along with system calls to make simple requests on them. The clock device driver will implement a `Sleep()` system call; the terminal will implement read and write functions; the disk will implement a basic query (to read the size of the disk) as well as read/write operations.

Common requirements from previous Phases (such as the requirement to implement an `init()` function, how to register system calls, etc.) will not be repeated here; refer to previous Phase specs as necessary.

2 Autograder Updates

This Phase implements system calls that allow a user to read and write to the various terminals. While USLOSS is designed to allow you to run an interactive terminal, our testcases will be far simpler.

I have provided 4 input files, `term[0123].in`, which are the standard input to any testcase. Essentially, as soon as USLOSS starts, the data is "typed" into the various terminals, at whatever rate USLOSS allows.

Some testcases, of course, also write to the terminals. The output generated by the testcase is saved in the files `term[0123].out`. You will notice that, in the `.out` files for each testcase, I also have a dump of the terminal output files. To pass the testcase, you must both have the correct output generated by `USLOSS_Console()`, but also have the correct output on teach of the 4 terminals.

3 Limitations?

In Phase 3, you were supposed to use mailboxes to build mutexes, or to perform queuing of processes. I encourage you to continue the practice in this Phase. However, I am not *requiring* it; therefore, if you prefer to use raw calls to `blockMe()` and `unblockProc()`, you are free to do so - but you will have to

implement your own queueing mechanisms, and almost certainly create another shadow process table.

(A real OS would not use mailboxes; they would change the process state directly - as well as using raw (spin) locks and/or disabling interrupts directly. But I think that getting practice using mailboxes is a handy learning experience for students at your level.)

Similarly, while you are **allowed** to mask off interrupts for terminals if you wish, I don't expect you to. In a real OS, we would mask off interrupts (at least, the write interrupts) on any terminal that was not in use, to improve performance. However, in this project, I'm not terribly worried about that.

4 Clock

You will only implement one system call related to the clock: `Sleep()`. This call simply asks that the process be blocked for a certain number of seconds.

You will recall that, in Phase 2, you already implemented a clock, which will wake up `waitDevice(CLOCK)` roughly every 100 ms. Therefore, since `Sleep()` asks for a delay in seconds, it's easy to wait a certain number of clock cycles.

However, `waitDevice()` is not designed to handle multiple processes waiting on the same device; therefore, you should have only one process calling `waitDevice()`. While there are a variety of ways that are permissible, I recommend that you have a simple process, in an infinite loop, which increments a counter each time that another interrupt is received. It then uses mailboxes to wake up sleeping processes, when their wakeup time (as measured in clock cycles) arrives.

5 Terminal

There are only two terminal syscalls: `TermRead()` and `TermWrite()`. They operate **entirely** independently; nothing about your readers should affect your writers, or vice-versa. However, since each terminal device can only have one waiting process (in `waitDevice()`), your device driver will need to, on each interrupt, pass information to both the read-side and the write-side.

5.1 `TermWrite()`

A call to `TermWrite()` should complete atomically; that is, while the buffer that you give the syscall is being flushed to the terminal, no other buffers can attempt to write to the same terminal. (Writes on other terminals, and reads on all terminals, are unaffected.) Therefore, your code will likely want to use some sort of mutex to control access to the terminal - not a mutex that protects data (which you would lock and unlock often), but a mutex that protects the write to even **attempt** to write - which you grab once, and hold for a very long time.

Once you hold the mutex, how should you write to the terminal? You cannot simply have some simple `for()` loop, which writes each character, since you are not allowed to write until the terminal is ready to receive it. Since the terminal may already be busy, you will need to wait when you begin writing - and you will have to wait in-between each character as well. But unfortunately, the process writing to the terminal is probably not the device driver, and thus it cannot call `waitDevice()` directly.

I will not tell you how you must implement this; you have freedom. But I've presented a couple options below.

Strategy A

One option is to have the syscall process post the buffer to global variables; each time that an interrupt occurs, the device driver process can directly send another character to the output.

In this strategy, the syscall process blocks, waiting for a single mailbox message from the device driver; the driver sends it when the entire buffer has been flushed to the terminal.

Strategy B

Another option is to have the syscall process wait on a mailbox message, sent by the device driver, for **every single** character. In this way, we are implementing something a bit like `waitDevice()` - but only for the write side - a process wanting to write simply waits for the message, and immediately writes out a character when it receives the message.

Of course, this would require the device driver to use `MboxCondSend()` to send the message, since it might encounter interrupts when there is no user trying to write to the terminal.

5.2 `TermRead()`

Reading is more complex than writing, because you must buffer up an unknown amount of data, for an unknown amount of time.

When a terminal has a new character to deliver to the CPU, it sends an interrupt, and when the CPU reads the Status Register, it will contain the character. Your device driver code will receive this information from `waitDevice()` (remember that `waitDevice()` reads the status of the device for you, and delivers it to you); thus each time that `waitDevice()` returns, you must check to see if the status indicates a new character has been read; if so, you must save it into the current working buffer.

`TermRead()` requires buffering because it always reads an entire line of data. Your read code will save up to `MAXLINE`[1] characters into a buffer; you deliver the buffer when either (a) a newline is typed; or (b) you fill the buffer. (In the

[1] usloss.h

latter case, the rest of the characters in the line - any beyond the `MAXLINE` limit - will be delivered in the next line.)

Moreover, since the user might be typing to a terminal before a process is attempting to read it, you must save up exactly 10 of these buffers in memory, ready to deliver. If `TermRead()` is called, you may immediately return the first saved buffer; if `TermRead()` is called many times, the backlog will eventually get flushed. (If, at any point, you end up with more than 10 lines of buffered input on a single terminal, discard any lines past the 10th.)

NOTE 1: A single call to `TermRead()` can only read a single line of input, even if many are buffered.

NOTE 2: If a user asks for less data, in `TermRead()`, than is avaialble in the next buffered line, then deliver what you have space to deliver, and discard the rest. Reading short, in `TermRead()`, is **not an error.**

HINT: Do you have a handy mechanism, which allows you to store small-to-medium sized buffers, in the order that they were created, and automatically discards extra buffers if you have already hit the maximum number allowed?

6 Disks

Disks have three system calls: `DiskSize()`, which reads a number of parameters about the disk (not just the number of tracks); and `DiskRead()/DiskWrite()`. The disk itself supports 4 operations: `TRACKS, SEEK, READ, WRITE.`

The `DiskSize()` syscall simply asks how large the disk is; you can find details in the formal spec below. You can either send a new `TRACK` operation for each request, or you can save the information in a table, and deliver it directly to syscall processes. However, if you do the latter, note that you will need some way to handle race conditions - the testcase might call `DiskSize()` before the `TRACKS` operation completes.

Unlike the low-level `READ,WRITE` operations, the `Read(),Write()` system calls can access any block, anywhere on the disk, and can access multiple blocks as a single call. To handle this, you will need to automatically generate `SEEK` operations to move the "disk head" around; you will often `SEEK` at the start of an operation, and occasionally, it may be necessary to `SEEK` again in the middle of the sequence, if the range of blocks crosses from one track to another.

If **any** operation, anywhere in a sequence, reports `ERROR` status, then you should report the error to the user (see the formal spec below to see how). Note that this means that data may be **partially transferred** - in a `WRITE`, only some of the disk blocks may have been updated, and in a `READ`, only some of the memory. This is **normal** - you should not attempt to make the changes atomic.

However, as regards other system calls, each `READ,WRITE` operation **must be atomic.** That is, once you start performing operations for a given `READ` or `WRITE`, no other `READ,WRITE` operation should be able to begin.

6.1 Optimizing Head Seeks

Your disk must reorder operations, if there are multiple processes attempting reads and writes at the same time. You are required to use a one-directional "elevator algorithm" (called C-SCAN in our zyBooks textbook). Basically, this means that the head will always seek forward (to higher track numbers), but will attempt to always make the shortest jumps possible; therefore, it will access the tracks in order. When it finishes the highest (current) request in the queue, the head seeks all the way down, to the very first request in the queue, and begins moving slowly upward again.

In our system, we don't care what order blocks are accessed within a track; we don't bother sorting the queue, since we assume that the blocks within the current track can **all** be accessed quickly.[2]

Both reads and writes should be kept in the same queue (since we're trying to minimize head seek time).

7 Required Syscalls

7.1 `int Sleep(int seconds)`

Pauses the current process for the specified number of seconds. (The delay is approximate.)

System Call: `SYS_SLEEP`

System Call Arguments:

- arg1: seconds

System Call Outputs:

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

(spec continues on next page)

[2]Sharp eyed readers may notice an ambiguity in the spec: what happens if there are multiple operations, all targetting the same track, but one (or more) of them will spill over into the next track? What happens then?

Good question! I'm going to ignore it for now - you can assume that the testcases won't ever do this.

7.2 `int TermRead(char *buffer, int bufSize, int unit, int *lenOut)`

Performs a read of one of the terminals; an entire line will be read. This line will either end with a newline, or be exactly `MAXLINE` characters long. (It will literally write `MAXLINE` characters, and so your buffer probably needs +1 character, if you plan to add a null terminator.) If the syscall asks for a shorter line than is ready in the buffer, only part of the buffer will be copied, and the rest will be discarded.

System Call: `SYS_TERMREAD`

System Call Arguments:

- arg1: buffer pointer
- arg2: length of the buffer
- arg3: which terminal to read

System Call Outputs:

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

7.3 `int TermWrite(char *buffer, int bufSize, int unit, int *lenOut)`

Writes characters from a buffer to a terminal. All of the characters of the buffer will be written atomically; no other process can write to the terminal until they have flushed. Of course, "atomically" in this case does not mean "fast," since it will take many seconds to write out even a short line.

System Call: `SYS_TERMWRITE`

System Call Arguments:

- arg1: buffer pointer
- arg2: length of the buffer
- arg3: which terminal to write to

System Call Outputs:

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

(spec continues on next page)

7.4 `int DiskSize(int unit, int *sector, int *track, int *disk)`

Queries the size of a given disk. It returns three values, all as out-parameters: the number of bytes in a block[3]; the number of blocks in a track; and the number of tracks in a disk.

System Call: `SYS_DISKSIZE`

System Call Arguments:

- arg1: the disk to query

System Call Outputs:

- arg1: size of a block, in bytes
- arg2: number of blocks in track
- arg3: number of tracks in the disk
- arg4: -1 if illegal values were given as input; 0 otherwise

7.5 `int DiskRead(void *buffer, int unit, int track, int firstBlock, int blocks, int *statusOut)`

Reads a certain number of blocks from disk, sequentially. Once begun, the entire read is atomic; no other syscalls will be able to access the disk.

System Call: `SYS_DISKREAD`

System Call Arguments:

- arg1: buffer pointer
- arg2: number of sectors to read
- arg3: starting track number
- arg4: starting block number
- arg5: which disk to access

System Call Outputs:

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

[3] or "sector"

7.6 `int DiskWrite(...)`

See `DiskRead()` above.

8 Turning in Your Solution

You must turn in your code using GradeScope. While the autograder cannot actually assign you a grade (because we don't expect you to match the "correct" output with perfect accuracy), it will allow you to compare your output, to the expected output. Use this report to see if you are missing features, crashing when you should be running, etc.