USLOSS Supplement - Disks

NOTE:
This document does not give any information that is not already present in the USLOSS documentation. However, this document will (hopefully) present that information in a form that is easier to understand - especially for people without much experience accessing hardware devices.

1 Overview

Each disk in USLOSS is made up of 512-byte blocks[1], organized into tracks of 16 blocks each. Each track is thus exactly 8 KB in size, and each disk will be some exact multiple of the track size.

USLOSS disks are modeled after spinning hard drives, which have a physical "head" which moves back and forth across a spinning disk. It takes several milliseconds (an **eternity** in computer terms) for the head to move to a new location; once there, the disk spins underneath it, and you can read (or write) any data in that **"track."** Thus, you can read many blocks in very little time, provided that they are all in the same track - but it takes a noticeable amount of time to "seek" to another track.

(document continues on the next page)

[1] Also known as "sectors"

2 Disk Operations

USLOSS provides exactly 4 disk operations[2]. Operations can have up to 2 parameters:

- `USLOSS_DISK_READ, USLOSS_DISK_WRITE`

  Reads or writes a single block inside the current track. Always accesses **exactly** 512 bytes. It cannot access any blocks outside the current track.

  You provide a pointer to a buffer, and the hard drive will use DMA to read/write the buffer directly.

  Parameter 1: Block index within the track, must be in range 0-15 (inclusive)

  Parameter 2: Buffer pointer

- `USLOSS_DISK_SEEK`

  Changes the current track. Does not do DMA; it simply gives status (`OK` or `ERROR`) when done.

  Parameter 1: Destination track

  Parameter 2: n/a

- `USLOSS_DISK_TRACKS`

  Queries the disk, to ask how many tracks it has.

  You provide a pointer to an `int`, and the hard drive will use DMA to write the answer into that variable.

  The size of a disk will never change while USLOSS is running, so you have a choice:

  - You may read the disk sizes once, as USLOSS starts up, and save those values for later, to deliver to users;
  - You may read the disk size each time that the user asks you the question (see the project Phase 4 spec).

  Parameter 1: Pointer to `int`

  Parameter 2: n/a

3 No Parallel Requests

While it is certainly possible to have both disks busy at the same time (doing different things), USLOSS **does not support** sending multiple requests to the same disk at the same time. A Real World disk almost certainly has the ability to queue up multiple requests, but in USLOSS, once a disk has begun an operation, the disk will refuse to start any others until the first completes.

[2] usloss.h

Thus, your operating system code will need to handle queueing (deciding which user to work for next), sequencing (doing many operations in a single user request), etc.

4 Sending Requests

Because disk requests are more complex than the terminal, it is not possible to encode them entirely in a single Control Register. Instead, the disk builds requests in memory, and then writes the **pointer** to the request struct to disk. The disk uses DMA to read the request struct, and then performs the operation.

Requests should be built using the struct `USLOSS_DeviceRequest`[3], which has fields `opr,reg1,reg2`. For example, the code snippet below will send a WRITE operation to a disk:

```
int   unit       = ... ;    // 0 or 1
int   blockIndex = ... ;    // 0-15, inclusive
char *buf        = ... ;

USLOSS_DeviceRequest req;
req.opr  = USLOSS_DISK_WRITE;
req.reg1 = (void*)(long)blockIndex;
req.reg2 = buf;

USLOSS_DeviceOutput(USLOSS_DISK_DEV, unit, &req);
```

4.1 Locals or Globals for the Request Struct?

The example above uses a local variable for the request struct. This is certainly permissible, but care is required. Remember that the disk operation will be running asynchronously after you post the request with `DeviceOutput()`; you cannot predict how early (or late) the disk might attempt to read that memory. Thus, it is critical that you **do not free** the memory until after the disk has reported status.

If you use a local variable, then this means that the function which holds the struct **must not return** while waiting for the request to complete. Depending on your design, this might or might not be easy to do. Therefore, remember that it is also permissible to use a global variable for the request struct. But would you need a shadow process table, so that you could have per-process information? Not necessarily. A reasonable implementation might be to simply have a single Request per disk, something like this:

```
struct DiskState
{
    USLOSS_DeviceRequest req;
```

[3] usloss.h

```
    ... other fields ...
} disks[2];
```

This works well because there can only be a single Request ongoing (on each disk) at a time; thus, it is possible, if you wish, to re-use the same struct over and over, for a long sequence of operations.

5 Interrupts

Unlike terminals, USLOSS does not give disks the ability to mask off their interrupts; this makes sense because disks (unlike terminals) will never send interrupts until the OS has asked them to perform some action.

The disk will send an interrupt any time that an operation has completed; you should use `DeviceInput()` to read the status on the device. (Unlike terminals, which have a Status Register made up of many fields, reading a disk returns a single integer value.)

The status will be one of:

- `USLOSS_DEV_READY`

  The previous operation completed successfully.

- `USLOSS_DEV_BUSY`

  The device is still busy. You should never see this right after an interrupt; if you do, this indicates a serious error in your code (or perhaps USLOSS).

- `USLOSS_DEV_ERROR`

  The most recent request encountered an error. If you see this, you should deliver it immediately to the user program that drove this operation, and it indicates to them that some sort of error has happened - such as a failed write, or an invalid seek, etc.

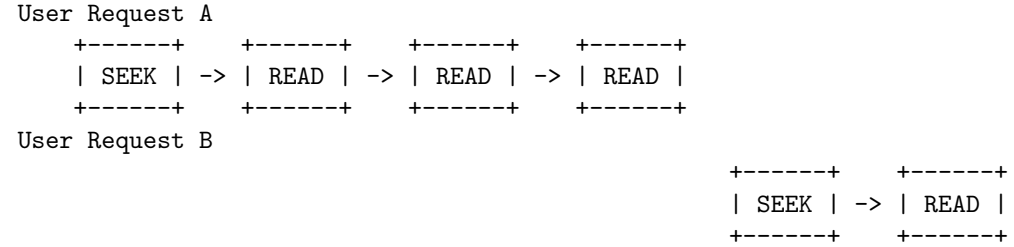
6 Sequencing

Each disk can perform only very limited operations (see above). This means that many user requests may need to be broken down into multiple steps. For instance, a READ may require first performing a SEEK to go to the correct track, followed by several READ requests to the disk (one per block).

Users are thus encouraged to make a distinction between requests made to the disk and requests made by the user to the operating system kernel. In addition, it probably will be important to implement some sort of queueing mechanism, which allows a user request, which is only partially fulfilled, to own the disk entirely until it completes. Otherwise, a SEEK from a new, incoming operation might cause the disk to jump to the wrong location!

(examples, with pictures, on the next page)

In this example, we have two different user requests, on different tracks. The first request accesses three different blocks. (Time advances to the right.)



In this example, we have only one user request, but it spreads across multiple tracks; it starts on the last block of the first track, and continues into the next track.