Kunnskap for en bedre verden

Department of Computer Science

TDT4186 - Operating Systems

Solution 6

I/O Scheduling, DMA & RAID

Professor:
Di Liu
Teaching Assistant:
Roman K. Brunner
Student Assistants:
Eik Hvattum Røgeberg
Halvor Linder Henriksen
Ole Ludvig Hozman
Daniel Hansen

Handout: 20.03.2023

Introduction

So far, we only worked on volatile parts of the operating system, meaning that once power is gone, the state is also gone. While this is fine for some small systems, for many systems, we want to save state persistently, e.g. photos you have taken. For that we use disks but might also use other devices such as printers and USB sticks. Communicating with those devices is called input/output, as information flows in and out of the system. The I/O subsystem handles this, and many decisions depend on the type of device and the desired properties. This exercise will focus on hard disks, memory mapping, and direct memory access I/O.

1 I/O Scheduling

For the following exercises, assume the disk organisation as in Figure 37.7, Page 10[1].

1. Assume that the arm and head are in the position as shown in figure 37.7 to begin with. The entries to be accessed are 31, 32,33, 7, 8, 12, and 23. Assume that it takes the same amount of time to move the arm by one track, as it takes for one track to move to the next entry (e.g. if we read 30, the next entry that we can read is 20 on the middle track, as we need one time unit to read out and another time unit to move the arm, thus 19 is in the read position while the arm is moving).

   (a) Given the list of I/O operations, in which order would they be accessed given the Shortest Seek Time First (SSTF) approach? How many timeunits does it take to complete all I/O operations?

   Answer: *Since we are accessing the entries first that have the shortest amount of seek time, we order the list by the time needed to seek. In this case, we are already on the innermost track, thus seek time for 31-33 are 0, so we start with them. After that, 12 and 23 are the next smallest seek-time. In the end, we will scheduler 7 and 8 as those had initially the longest seek time. So overall, the access will happen in the following order: 31, 32, 33, 23, 12, 7, 8. The overall time required is 14 time units (tu) (reading 31-33 (3 tu), switching track (1 tu), reading 23 and 12 (2 tu), switching track (1 tu), waiting for 7 to arrive (5 tu), reading 7 and 8 (2 tu))*

   (b) Would any other of the presented I/O scheduling algorithms behave differently?

   Answer: *In this case not, as SSTF behaves the same as a SCAN of SPTF algorithm in this particular case, as it finds both the shortest positioning time and the SCAN algorithm would behave the same in this position of the arm (moving in one direction only).*

2. Given an average seek time of 4ms, what would be the worst-case seek time you would expect to see?

   Answer: *The worst seek time would be three times the average seek time, so we would expect it to be 12ms. The reason is that the average seek time is calculated to be one-third of a full seek, see equations 37.4 to 37.8. Since a full seek is the worst-case scenario for seek time, the max seek time will be three times the average seek time.*

2 RAID

Even though many systems nowadays run on SSD's, magnetic drives are still in high demand due to the fact that their storage is very cheap and we don't always need the raw speeds of SSD's. While RAID configurations used to be about making slow magnetic HDDs a bit faster and more reliable, the focus is now mainly on the reliability part of the equation.

Assume the numbers given in Figure 37.5 in chapter 37.4[2] for the following exercises.

[1] https://pages.cs.wisc.edu/~remzi/OSTEP/file-disks.pdf
[2] https://pages.cs.wisc.edu/~remzi/OSTEP/file-disks.pdf, page 7

1. For the first subtask we are still looking at how much more throughput we might be able to get out of a raid configuration. Assume a RAID-0 configuration, how many additional disks do we need when using the Barracuda HDD compared to the Cheetah in order to match the throughput? Assume two RAID-0 systems one with only Cheetah's, the other with only Barracudas. We are considering the steady-state sequential throughput. How does the amount of storage compare between the two systems?

   Answer: *To match the speed, we can calculate the least common multiple of the throughputs (125 and 105 MB/s). This leaves us with with 21 Cheetah's disks and 25 Barracuda disks and an overall throughput of 2625 MB/s or roughly 2.5 GB/s. In terms of storage the Cheetah system results in roughly 6 TB, whereas the Barracuda system will have 25 TB of storage.*

2. Assume that our system runs RAID-0 on 20 disks. We assume an annualized failure rate (AFR) of 1% for every single disk, which means there is a 1% probability that any given disk fails during the period of one year. How likely is it that we will lose data over a period of three years?

   Answer: *We lose data when a single disk fails, as only that disk contains its chunk of data. Therefor we calculate the probability of a failure as follows* $P[SYSTEM\ FAILS] = 1 - P[NO\ DISK\ FAILS]$*. The probability that no disk fails during a period of three years is calculated by raising the probability that no disk fails in a single year to the power of 3. And the probability that no disk fails in a single year is given by* $P[NO\ DISK\ FAILS] = 0.99^{20} = 0.8179$*, so the probability that our system keeps our data safe over the period of three years equals* 54.72%*. But that also means that over a three year period we have a probability of* 45.28% *that we lose data, compared to* 2.9% *chance of losing data with a single disk.*

3. We want to improve the reliability of our system by splitting up our 20-disk system into 4 RAID-4 systems with 5 disks each. How much memory do we have overall and what is the reliability of such a configuration compared to the single 20-disk RAID-0 system in the question above? Assume we don't replace broken hardware.

   Answer: *For RAID-4 we can tolerate a single disk failure at any given point in time. In order for the system to lose data, we need to disks to fail "simultaneously" (before the defect disk has been replaced and the data has been reconstructed). By the assumption of not replacing hardware, we can now calculate the probability that two disks fail in a single year in a 5-disk RAID-4 system. The probability of dataloss in a single year for a single of our RAID-4 systems is* $P[DATALOSS] = 1 - P[NO\ DATALOSS] = 1 - (\binom{5}{1} * 0.01 * 0.99^4 + 0.99^5) = 0.00098$*, so we have an yearly probability for dataloss of 0.098%. Taking the no dataloss parameter to the power of 3 for three years yields a failure probability of 0.29% for a single RAID-4 system. Raising* $P[NO\ DATALOSS]$ *to the power of 4 to adjust for four systems we end up with* $P[DATALOSS] = 1 - P[NO\ DATALOSS_{single\ system}]^4 = 1 - ((\binom{5}{1} * 0.01 * 0.99^4 + 0.99^5)^3)^4 = 0.0117$*. So the overall failure probability for all our 4 RAID-4 systems now is 1.17%. The storage amount that we have is reduced by the amount that 4 disks can save, so we lose 20% storage, while gaining roughly 38 times higher reliability.*

3 I/O Interaction

You are given a new piece of I/O hardware that enables the user to input and output odours. It has a very simple interface, with one 64-bit register each for Status, Command and Data. The Status can either be `UNKNOWN`, `BUSY`, `READY`, and `MODE`. If `MODE` is set to 1 it is reading odours, if it is set to 0 it is set to outputting.

1. Given the description above and the three commands `INIT`, `WRITE` and `READ`, what would a simple programmed I/O driver for the device look like? Assume that `INIT` initializes the device to whatever flags have been written to the DATA registers.

   Answer: *We use a simple busy-wait loop to wait for the device to come online (and wait for the device to not be busy anymore), reading and/or writing directly to and from the data register. We assume that there is a physical address assigned to the device and the registers are addresses relative to the base address of the device. For brevity, we omitted device locking* *(only ever one process should be allowed to access the device simultaneously).*

```
void ready(void *dev_ptr) {
    while (dev_ptr[STATUS] == BUSY);
}
void init_dev(void *dev_ptr, uint64_t mode){
    dev_ptr[DATA] = mode;
    dev_ptr[COMMAND] = INIT;
    ready(dev_ptr);
}
void read_odour(void *dev_ptr, void *buf, uint64_t size) {
    dev_ptr[COMMAND] = READ;
    assert(dev_ptr[STATUS] & MODE);
    for (uint64_t i = 0; i < size; ++i) {
        ready(dev_ptr);
        buf[i] = dev_ptr[DATA];
    }
}
void read_odour(void *dev_ptr, void *buf, uint64_t size) {
    dev_ptr[COMMAND] = WRITE;
    assert(!(dev_ptr[STATUS] & MODE));
    for (uint64_t i = 0; i < size; ++i) {
        ready(dev_ptr);
        dev_ptr[DATA] = buf[i];
    }
}
```

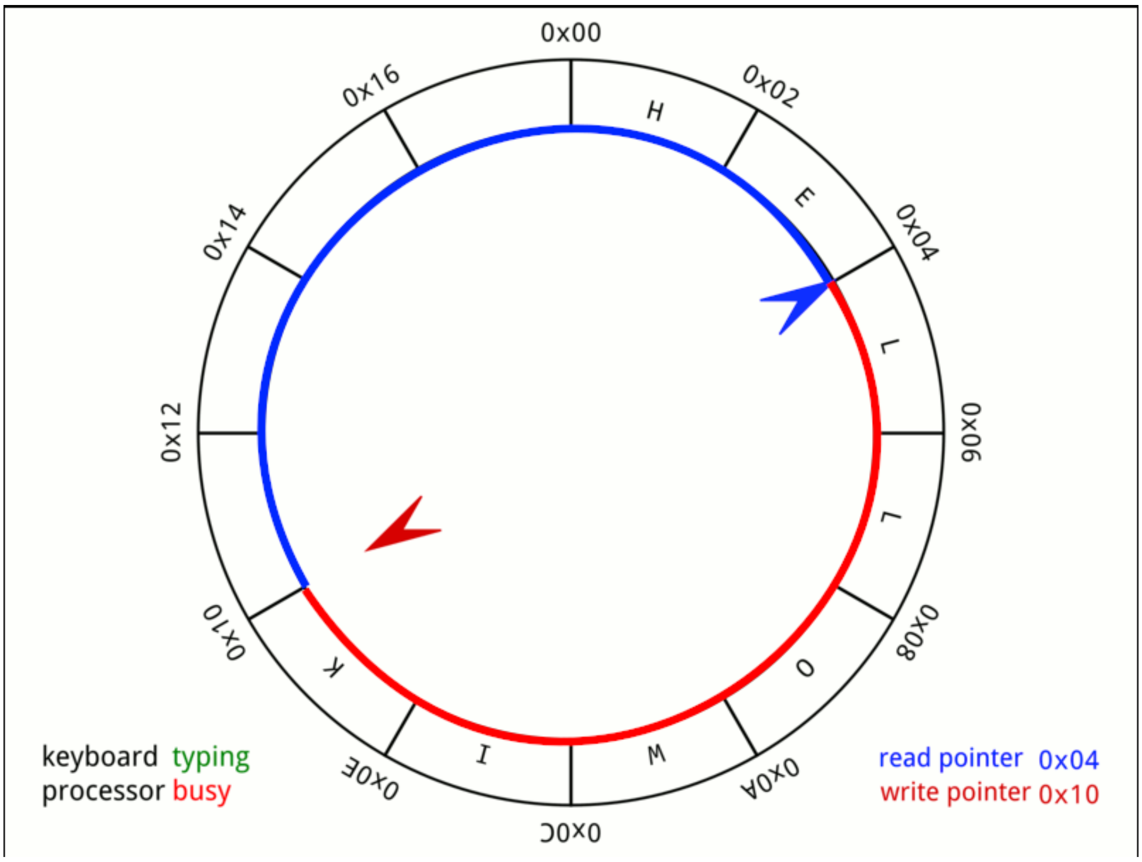
Listing 1: Possible simple driver for our odour device

2. After releasing the device with your simple driver, you receive user complaints that when watching a movie with odour enabled, the machine's performance greatly degrades (The machine gets very hot, stuttering images and low responsiveness of the system overall). What could cause that problem and how could you improve your driver? What changes would you need in the interface of the device?

   Answer: *The driver is most likely using up a lot of CPU resources if the device is used constantly to output odours when watching a movie with odour information. Thus, other processes that would need the CPU, e.g. to decode DRM-protected frames, don't get enough CPU time to perform the tasks that they are supposed to in a timely fashion. Thus, we need to free some CPU time for those tasks. One way of achieving that is adding interrupts to the odour device, such that the odour device can notify the CPU once it has either read or written out an odour. This means that instead of busy looping on the status register, we could yield the CPU for other processes to take over. To further reduce the load on the CPU, we would need a direct memory access engine and adapt our driver accordingly. E.g. we could provide a pointer and a size through the data register before calling write. After that, we would yield the CPU and wait for the finished interrupt. That means we now only have to do a single constant operation per read or write without having to loop over the buffer on the CPU itself.*

3. For DMA engines, we need to allocate a buffer that the engine can either read from or write to. This means that we need to provide a way of how the DMA engine (and the program) can detect if it is reading valid data from the buffer. Imagine our device is writing new data to the buffer, how can the CPU know, which parts of the buffer are already ok to be read? Propose a solution to that problem.

   Answer: *A very common solution is a so-called circular buffer. That is still a linear buffer in memory, but the pointers are calculated modulo the buffer size, so the pointer moves "around" the buffer in a circular fashion. Aside the buffer, we now have two pointer locations, where both the CPU and the device indicate where they are currently with reads, respectively writing. Now whenever the CPU wants to see if it can read, it just compares the values of the pointers* *and checks if there is a gap between the read/write pointer. If there is a gap, the CPU can process on as many bytes as the gap is big. On the other side, the write pointer can check how many bytes it can still write to the buffer by checking the difference between the read pointer and the write pointer.*



Red indicates the valid reading zone for the reader, where as blue indicates the already read entries, that the writer can use to write new data to the buffer. See the animated version under https://w.wiki/6SQe.

Adapted from ”Circular Buffer Animation” by MuhannadAjjan, licensed under CC BY-SA 4.0, Screenshot and highlighted zones, is licensed under CC BY-SA 4.0 by Roman Brunner