Pale INF422 – Composants d'un système informatique
INF422 Final Exam – Components of a Computing System

2008–2009

- L'examen dure 3 heures.
  The exam lasts 3 hours.
- Tous documents autorisés.
  All documents are authorized.
- Il est impératif de commenter les programmes et de justifier vos réponses.
  Programs must be commented and every answer must be justified.

English Text

We study how to read files stored on an optical disk such like a CDROM.

A (simplified) CDROM is modeled as a long spiral *track* composed of a series of N "holes" representing data bits. The data can be accessed through a unique reading head capable of *radial* translations. The track circles C times around the center of the disk, progressing inwards of P microns at every full circle (P for track Pitch). We assume the disk rotates at a constant pace of T rounds per minute, and that the radial translation of the head operates at P micron per microsecond. Note that the head moves slightly when reading consecutive buts: we assume its maximal speed is much higher than what is required to follow the spiral track along with the disk rotation. Let us give some realistic figures for a classical CDROM:

- $N = 5.6 \times 10^9$ bits or 7×10^8 bytes;
- $P = 2\,\mu m$;
- $C = 20000$, hence every disk rotation exposes 280000 bits on average;
- and $T = 300\,mn^{-1}$.

These figures are given for illustrative purposes, and we will hereafter prefer symbolic computations (parametered by N, P, C and T).

The *controller* of the optical drive is a microprocessor embedded on the device itself. It manages the radial placement of the head, the waiting period before a specific disk location passes below it, and can read a block of consecutive bytes.

The operating system manages the optical device through a dedicated driver. This driver runs in privileged (kernel) mode. When a user process attempts to read an open file located on the CDROM, the process uses the `read()` call, which delegates the operation to the device driver.

1 Usage Scenario

We assume that the device is known by the operating system through the *major number* 11, itself being associated with the device file `/dev/cdrom`.

Question 1.1

The command `ls -l /dev/cdrom` prints:

```
brw-rw---- 1 root cdrom 11, 0 2008-11-07 09:04 /dev/cdrom
```

Explain this message thoroughly.

Question 1.2

We assume that a CDROM has been inserted in the drive, and that it has been formatted with an `iso9660` file system (the standard for CDROM) containing a unique file called `record`.

We execute the following UNIX commands as `root`, assuming the `media` directory is initially empty:

```
mkdir /media/cdrom
ls /media/cdrom
mount -t iso9660 /dev/cdrom /media/cdrom
ls /media/cdrom
```

What does this sequence of commands print? What is the role of the `mount` command in this sequence?

In the following, we assume that this sequence has been executed.

Question 1.3

By default, the `mount` command run as `root` only provides limited access rights. What is the command to run (as `root`) to grant read access to the `record` file to all users?

Observe the access rights of the `/dev/cdrom` file and those of the `record` file. Is there a problem? If so, which one? If not, why?

Question 1.4

In the following, we assume that all Java applications are executed in a Java virtual machine run with the rights of a plain user. Give two reasons why it is not desirable to run a classical application as `root`.

Question 1.5

What is the purpose of the following Java method?

```
static byte[] whatisit(String name) {
    byte[] buffer;
    try {
        File file = new File(name);
        int size = file.length();
        buffer = new byte[size];
        FileInputStream in = new FileInputStream(file);
        while (size > 0) {
            /* Tries to read ‘size’ bytes into the array ‘buffer’
               starting at the offset ‘position’, and return
               the number of bytes that were effectively read */
            n = in.read(buffer, position, size);
            position = position + n;
            size = size - n;
        }
        in.close();
    } catch (Exception e) {
        System.exit(1);
    }
    return buffer;
}
```

Using this method, write a `main` method which reads the `record` file from the CDROM.

Question 1.6

Processing `in = new FileInputStream(file)` opens the `record` file and creates a dedicated `in` object to read it. What are the respective roles of the Java virtual machine and of the operating system in this operation?

Question 1.7

Describe the main steps that the operating system kernel must go through to discover that the device in charge of the `record` file is the CDROM drive.

Question 1.8

Cite two advantages of mounting the CDROM in the `/media/cdrom` directory, as opposed to letting applications access the `/dev/cdrom` file directly.

Question 1.9

When calling `in.read()`, the Java virtual machine performs a `read()` system call, using the file descriptor that was assigned when opening the `record` file. This system call tells the kernel how many bytes to read and the memory address where to store those bytes. The file descriptor lets the kernel identify the data blocks containing the requested bytes; indicate the main steps and data structures involved in this identification.

2 Non-Blocking Reads

In the following, we consider file system blocks whose size is 1024 bytes.

Question 2.1

To implement the `read()` call, the kernel calls a standard function of the device driver, dedicated to the retrieval of a unique data block. This function, called `read_block()`, takes the number of the file system block to read and the address of a memory area where to store this block's data. It transfers these parameters in the form of a standard *request* into a system interconnection bus, towards the controller of the optical drive.

Why should users be prevented from directly calling such a function? Indicate three important reasons.

Question 2.2

Considering one of the worst situations (block location, head location, disk rotation angle), compute the maximal latency between a call to `read_block()` and the complete reading of the block (we will neglect the driver's execution time, including the time it takes to store the block in the computer's memory).

Question 2.3

While a request is handled, it is not desirable that to stall the `read_block()` function until its termination, but on the contrary it is expected that the driver prepares the follow-up request or returns to the kernel for other tasks to be performed (including the execution of other applications).

When the controller of the optical drive signals the processor of the end of a request, the latter interprets this signal as an *interrupt*. What needs to be done by the device driver to prepare the reception of such an interrupt? Which actions should be performed when handling the interrupt itself?

Question 2.4

It is important not to trigger long computations inside an interrupt handler, or the reactivity of the system is likely to drop. A modern approach to this problem avoids direct interactions with the controller by the device driver, forwarding the requests to a dedicated process instead, called `kcdromd`, also running in privileged mode. The `kcdromd` process only interacts with the associated driver, and manages the optical drive controller directly.

Request forwarding is implemented through a dedicated memory area, implementing a queue (first-in-first-out). To implement such a mechanism, we can use the `pause()` system call which suspends the execution of a process until a signal is delivered (`kill()` system call). Define and describe a protocol to allow:

- the `kcdromd` process to suspend its execution when the request queue is empty;
- the driver to wake this process when new requests are enqueued;
- the driver to proceed without waiting for the completion of the requests.

Question 2.5

In this question, we use Java to simulate the previous protocol.[1]

The queue is stored in an array of objects of a `Request` class (to be defined); to simplify the code, the array can be considered infinite.

Let us view the driver calling `read_block()` and the `kcdromd` process as plain concurrent *threads*, and ignore (for now) the *race condition* issues that may arise among accesses to the array of requests

Implement the `Request` class as well as a `FIFO` class featuring static `enqueue()` and `dequeue()` functions (and auxiliary static variables) such that the driver may enqueue requests by calling `FIFO.enqueue()`, the `kcdromd`

[1] An absurdity for a piece of code running in privileged mode, but this is only for the sake of illustration.

process may handle requests in the same order calling `FIFO.dequeue()`, suspending itself when the queue is empty. You may use `pause()` and `kill()` functions performing the corresponding system calls.

Question 2.6

Race conditions among threads may lead to inconsistent states. Describe a race condition in your code involving concurrent enqueue and dequeue operations. Exhibit a real interleaving of calls to `read_block()` et execution of the `kcdromd` process that reveals the problem.

Question 2.7

Under certain conditions, a race condition may also occur when multiple requests are enqueued. Which are these conditions? What hypothesis should be made on the execution of system calls in the kernel?

Question 2.8

How does the Java language help fixing these race conditions? Tell how your program should be modified (without rewriting it).

3 Optimized Request Handling

Given the slow motion of the head and the slow rotation of the disk, one often introduce a heuristic, called SCAN, analogous to the *elevator principle*. The SCAN heuristic can optimize the time to perform *a list of pending requests*. Therefore, the ability to initiate multiple requests without waiting for their completion is essential.

Question 3.1

The principle is the following: the head starts at the outermost location, then the `kcdromd` process dequeues the requests according to increasing block numbers. When reaching the innermost block of the sequence, the traversal switches to the opposite direction and requests are scheduled according to decreasing block numbers; and so on.

Of course, the profitability of this heuristic depends on the order in which requests are emitted by `read_block()` calls. Does it also depend on the distribution of the accessed block numbers? Provide a quantitative analysis in the case of a sequence of uniformly distributed blocks, or squeezed against one of the outer or inner end. You may construct a typical sequence corresponding to each case, rather than conducting a statistical study in the general case.

Question 3.2

Show that the heuristic does not fully model the temporal features of the CDROM drive (it derives in fact from a popular heuristic for hard drives). Describe informally an improved method that exploits the elevator principle but according to a different ordering.

Question 3.3

Show that the SCAN heuristic is fair, meaning that any pending request will eventually be handled within a bounded delay.

Question 3.4

Build a sequence of request handling and `read_block()` calls such that the delay to complete a specific read operation may grow much beyond the delay of pending requests at any given time.

This property of the SCAN heuristic shows that formal fairness is not sufficient to guarantee a “reasonable” bounded reactivity.

Question 3.5

Propose a variation of the SCAN heuristic such that the maximal waiting time for any request depend only on the number of requests in the queue at the time it is emitted.