CS2106 Introduction to Operating Systems

Semester 2 2023/2024

Questions

File Abstraction

1. Explain the following concepts in your own words clearly; the shorter, the better! Your explanation should be easily understandable for non-CS2106 students.

   a. What is a file?

   b. Name and describe some classifications of files.

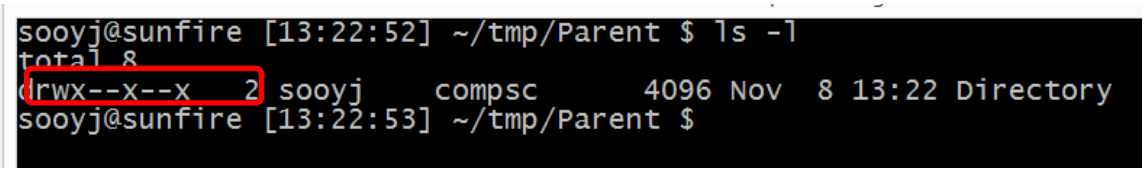
   c. Distinguish between a file type and a file extension.

   d. What does it mean to open and close a file?

   e. What does it mean to truncate a file?

---

2. [Understanding directory permission] In Unix system, a directory has the same set of permission settings as a file. For example:



You can see that directory **`Directory`** has the read, write, execute permission for owner, but only execution permission for group and others. It is easy to understand the permission bits for a regular file (read = can only access, write = can modify, execute = can execute this file). However, the same cannot be said for the directory permission bits.

Let's perform a few experiments to understand the permission bits for a directory.

Setup:

- Unzip **`DirExp.zip`** on any unix platform (Solaris, Mac OS X included).
- Change directory to the **`DirExp/`** directory, there are 4 subdirectories with the same set of files. Let's set their permission as follows:

`chmod 700 NormDir`	`NormDir` is a normal directory with read, write and execute permissions.
`chmod 500 ReadExeDir`	`ReadExeDir` has read and execute permission.
`chmod 300 WriteExeDir`	`WriteExeDir` has write and execute permission.
`chmod 100 ExeOnlyDir`	`ExeOnlyDir` has only execute permission.

Perform the following operations on each of the directory and note down the result. Make sure you are at the **`DirExp/`** directory at the beginning. DDDD is one of the subdirectories.

a. Perform "**`ls -l DDDD`**".
b. Change into the directory using "**`cd DDDD`**".
c. Perform "**`ls -l`**".
d. Perform "**`cat file.txt`**" to read the file content.
e. Perform "**`touch file.txt`**" to modify the file.
f. Perform "**`touch newfile.txt`**" to create a new file.

Can you deduce the meaning of the permission bits for directory after the above? Can you use the "directory entry" idea to explain the behavior?

---

3. [Wrapping File Operations] File operations are very expensive in terms of time. There are several reasons: a) As we learned in the lecture, each file operation is a system call, which requires an execution mode change (user ➔ kernel); b) Secondary storage mediums have high access latencies.

This leads to a strange phenomenon: it is generally true that the total time to perform 100 file operations for 1 item each is **much longer** than performing a single file operation for 100 items instead. e.g. writing one byte 100 times takes longer than writing 100 bytes in one go.

Most high-level programming languages therefore provide **buffered file operations** that wrap around primitive file operations. The buffered version maintains an internal intermediate storage in memory (i.e. buffer) to store values read from/written to the file by the user. For example, a **buffered write operation** will wait until the internal memory buffer is full before doing a large one-time file write operation to ***flush*** the buffer content into file.

a. (Generalization) Give one or two examples of buffered file operations found in your favorite programming language(s). Other than the "chunky" read/write benefit, are there any other additional features provided by these high-level buffered file operations?

b. (Application) Take a look at the given "**`weird.c`**" source code. Compile and perform the following experiments: Change the trigger value from 100, 200, … until you see values printed on screen **before the program crashes**. Can you explain both the behavior and the significance of the "trigger" value? If you add a new line character "**`\n`**" to the **`printf()`** statement, how does the output pattern change? How can this information be useful?

c. (Design) Give an algorithm in **high-level pseudo-code** to provide a buffered read operation. Use the following function header as a starting point:

```
BufferedFileRead(file, outputArray, arraySize)
// Read arraySize items from file and place the items in outputArray
```

---

4. [Wrapping File Operations, again] Study the attached program `weird_read.c`. Note that it is written to run with the given input `alice.txt`, but you can modify it to read your own file.

a. Uncomment `a();` in `main`, then compile and run the program. What do you observe and why?

b. Uncomment `b();` in `main`, then compile and run the program. What do you observe and why?

c. Uncomment `c();` in `main`, then compile and run the program. What do you observe and why?

   i. What happens when the first `read` and `printf` are removed instead?

   ii. What happens when c() is run with multiple threads instead of processes?
