CS2106 Operating Systems
Semester 2 2019/2020

Tutorial 10 Solutions
File Abstraction

1. (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 changes? 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:

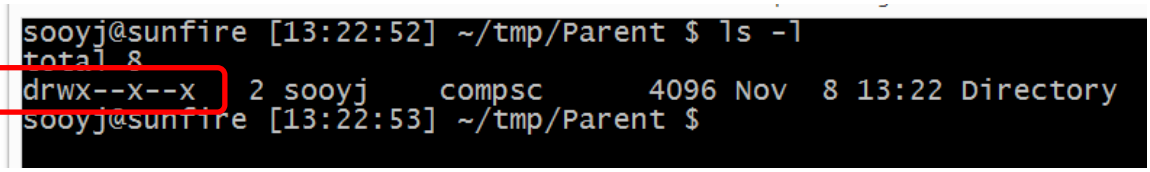
      ```
      BufferedFileRead(file, outputArray, arraySize)
      // Read arraySize items from file and place the items in outputArray
      ```

2. (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 f`read` and `printf` are removed instead?
      ii. What happens when c() is run with multiple threads instead of processes?

3. (Adapted from [SGG]) Why do many operating systems have a system call to "open" a file, rather than just passing a path to the read or write system calls each time?

4. (Understanding directory permission) In *nix 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 *nix 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?

Questions for exploration:

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 the two basic 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. (Case Study) Apache Kafka is a distributed message system for log processing. In summary, its main purpose is to stream log messages to log consumers as they are created. In this question, we will look at how logs are stored and managed in Kafka.

   For parts a and b, assume that each stream of logs is implemented as a file that is constantly appended to by a log writer.

   a. Suppose each log message is of a **fixed size (64 bytes)** and Kafka receives a subscription request to a stream of logs. Briefly describe how you would implement a handler for this request.

   b. Suppose that each log message can be of **variable size**. Briefly describe how you can modify the way you store each log message and your answer in part a to accommodate this.

   A stream of logs be virtually never-ending and stretch up to tens of gigabytes. Storing everything in a single file in the file system poses plenty of inconveniences – imagine if you want to delete older logs. Instead, Apache Kafka separates each stream of logs into multiple contiguous segments, storing each segment as an actual file.

   c. Suppose you need to navigate to a particular message log, located at a stream offset (similar to a file offset, but relative to the whole stream). How can we achieve this, given that streams can be located in different segment files?