CS2106 Operating Systems
Semester 1 2021/2022

Tutorial 10 Solutions
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?
   It is a named collection of data, used for organizing secondary storage.

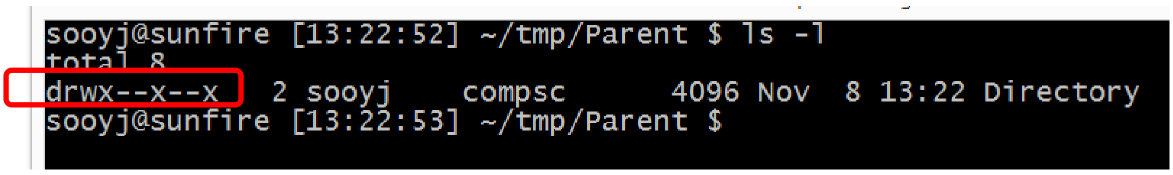
   b. Name and describe some classifications of files.
   Regular files: contain user information
   Directories: system file for FS structure
   Special Files: character/block oriented

   c. Distinguish between a file type and a file extension.
   A file type is a description of the information contained in the file. A file extension is a part of the file name that follows a dot and can help identify the file type.

   d. What does it mean to open and close a file?
   Operating systems keep a table of currently open files. The open operation enters the file into this table and places the file pointer at the beginning of the file. The close operation removes the file from the table of open files.

   e. What does it mean to truncate a file?
   Truncating a file means that all the information on the file is erased but the administrative entries remain in the file tables. Occasionally, the truncate operation removes the information from the file pointer to the end.

2. (Understanding directory permission) In *nix system, a directory has the same set of permission settings as a file. For example:



```
sooyj@sunfire [13:22:52] ~/tmp/Parent $ ls -l
total 8
drwx--x--x    2 sooyj     compsc       4096 Nov   8 13:22 Directory
sooyj@sunfire [13:22:53] ~/tmp/Parent $
```

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?

ANS:

	ReadExeDir	**WriteExeDir**	**ExeOnlyDir**
a	ok	nope	nope
b	ok	ok	ok
c	ok	nope	nope
d	ok	ok	ok
e	ok	ok	ok
f	nope	ok	nope

The responses may seem arbitrary unless you apply the understanding that "Directory == the list of directory entries (file/subdir)".

Then, the permission can be understood as:

Read = Can you read this list?   (impact: ls, <tab> auto-completion)

Write = Can you change this list? (impact: create, rename, delete file/subdir). Note the interaction with the execute permission bit.

Execute = Can you use this directory as your working directory? (impact: cd).

Since modifying the directory entries (i.e. with write permission) requires us to set the current working directory, execute bit is needed for the write-related operations under the target directory.

Some interesting scenarios:

a. Directory permission is independent from the file permission. So, you still can modify a file under a "read only" directory if the file allows write.

b. If you want to allow outsider to access a particular deeply nested file, e.g. A/B/C/D/file, you **only need** execute bit on A, B, C, D directory (i.e. read permission is not needed). This is a great way to hide the content of the directory and only allow access to specific file given the full pathname.

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?

Answer:

- Either the read/write system calls return a handle to a file descriptor (like open normally does), or they do not and we pass a path to read/write each time.
- In the former case, it would complicate the interface of the system calls, because they would then need to also accept a handle instead of a path somehow.
- In the latter case, we would incur the cost of permission checking and path resolution each and every time the system call is made, which are non-trivial and expensive. Also, we lose the ability to have the OS track our offset within the file.
- In either case, such an interface would not generalise well to file descriptors that do not come from paths on the filesystem, like pipes and anonymous sockets.
- Some history: open, read, write and close were present in the original Unix OS from the start. Although pipes and sockets, etc., didn't come around until much later, the design generalised well to those. See https://unix.stackexchange.com/a/265627, http://www.read.seas.harvard.edu/~kohler/class/aosref/ritchie84evolution.pdf

4. (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
```

ANS:
a.
C: printf, scanf, fprintf, fscanf (printf and scanf are specifically to stdout and stdin, and are specialized versions of fprintf and fscanf. One can specify stdout/stdin as the file pointers. All of these are buffered. The buffer is flushed when (i) full or (ii) fflush is called or (iii) new line character is read/written)
Java: BufferedReader/BufferedWriter, Scanner. (BufferedReader/BufferedWriter have 8kB buffer while Scanner has 1kB)
C++: "<<", ">>" stream operators. Buffered.
Common additional features: error checking, packing/unpacking of datatypes (e.g. low level file operations are usually operating in bytes, it is useful to be able to directly operate on other datatype (int, float, etc) without worrying about how to translate the value into/from human readable string).

b.
- printf buffers the user output until the internal buffer is full before actually output to the screen.
- The trigger indicates the probable size of the internal buffer.
- If a newline character is added, the output is performed "immediately".
- If printf() or similar is used as a debugging mechanism, the buffered output sequence may confuse the coder. e.g. in the original "weird.c", the program can crash without showing any printout, which can easily leads to the wrong conclusion ("the while loop is not executed!").

c.
Only key points are mentioned.

```
BufferedFileRead(file, outputArray, requestSize) // pseudo code
// Read arraySize bytes from file and place the contents in outputArray

// Assume we have an internal buffer buffer with size and availableItems attributes
// The buffer can be implemented as a circular array

  If buffer.availableItems < requestSize // not enough items in buffer
    read(file, buffer, buffer.size – buffer.availableItems) // low level file reading
    buffer.availableItems = buffer.size // Buffer is full now

  memcpy(outputValues, buffer, requestSize) // copy the user requested items over to output
  buffer.availableItems -= requestSize // assume requestSize <= buffer.size
```

Scenarios not considered for the above simple key points:
1. When `requestSize` is larger than `buffer.size`
2. When the file is exhausted (i.e. cannot refill buffer anymore)

These scenarios can be easily supported by building on the key points above. Look at the accompanying BufferedFileRead.c (fread function) file for the complete solution.

5. (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?

   The characters are printed in a seemingly random order.

   The file is opened once before `fork`, so the two processes' file descriptors for `alice.txt` are pointing to the same open file table entry. When a process reads 1 character, the offset would be incremented, and the next read by either processes would be for the next character.

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

   The first 50 characters of the text are printed in order two times.

   As `open` is called after `fork`, each process's file descriptor would point to different open file table entries. Therefore, the offsets which reads are called from are independent between the two processes.

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

   The first 50 characters of the text are printed in order two times (like b).

   `fread` is a buffered IO operation (see 4c)), so it would read as much as its buffer can contain, but only copies to the caller's buffer the number of bytes that the user requests. As the first `fread` is called before the fork, the filled buffer is copied to the child process. The child process then reads from this buffer until it consumes everything before it actually issues a file read.

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

      One process would print the first 50 characters, and the other would print the next (order can be either). As `fopen` is called before the fork, they are sharing a file descriptor like in a). But the underlying `read` syscall is called with buffer size = 50 instead, so every 50 characters read are not interleaved.

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

      The read buffer is shared in process space, so an interleaving similar to (a) would be expected for fread calls. However, since the write buffer in printf to stdout is shared between both threads, the output to stdout will be interleaved as well. Hence, the interleaving behaviour may not be evident in most execution scenarios