CS2106 Operating Systems

Semester 1 2020/2021

Week of 5th April 2021

Tutorial 10

File Abstraction

1. [Working Set Model] Working set model allows OS to make better decision for frame allocation among processes. Let us consider several aspects of the idea in this question.
   a. [Basic Idea] Given the following memory reference string (from lecture 9 – page replacement section):

Time	1	2	3	4	5	6	7	8	9	10	11	12
Page	2	3	2	1	5	2	4	5	3	2	5	2

   Given that W(T, Δ) gives the working set at time T with a window size of length Δ, give the following:

   W(9, 3), W(11, 3), W(9, 4), W(11, 4)

   b. [Application] Suppose we know the W(T, Δ) value for all processes, how can OS use this information? Hint: think about virtual memory + demand paging. Demand paging refers to the idea that OS only loads a page into RAM upon page fault instead of trying to predict and pre-load pages for processes.

   (Exploration Question) Using the same idea, is there any easy way to dynamically adjust the Δ value?

   c. [Implementation] Consider the following "mysterious" algorithm:

> 1. Every Page Table Entry has an additional K-bit mysterious value, M_0, M_1, … M_{K-1}. All K-bit are initialized to '0'.
>
> 2. Whenever a page P is referenced, its corresponding Mysterious value are updated as follows:
>   a. $M_{K-1} \leftarrow M_{K-2}$
>   b. …
>   c. $M_2 \leftarrow M_1$
>   d. $M_1 \leftarrow M_0$
>   e. $M_0 \leftarrow 1$
>
> 3. All other Non-P pages' Mysterious value are updated as follows:
>   a. $M_{K-1} \leftarrow M_{K-2}$
>   b. …
>   c. $M_2 \leftarrow M_1$
>   d. $M_1 \leftarrow M_0$
>   e. $M_0 \leftarrow 0$

   What does the mysterious value represents? If the above algorithm is handled by the hardware, what should be done every K memory accesses?

2. [Wrapping File Operation] 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 has high latency access time.

   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. write one byte 100 times takes longer than writing 100 bytes in one go.

   So, most high level programming languages provide **buffered file operations** that wraps around the primitive file operations. The buffered version essentially maintains an internal intermediate storage in memory (i.e. buffer) to store user read/write values from/to the file. 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.

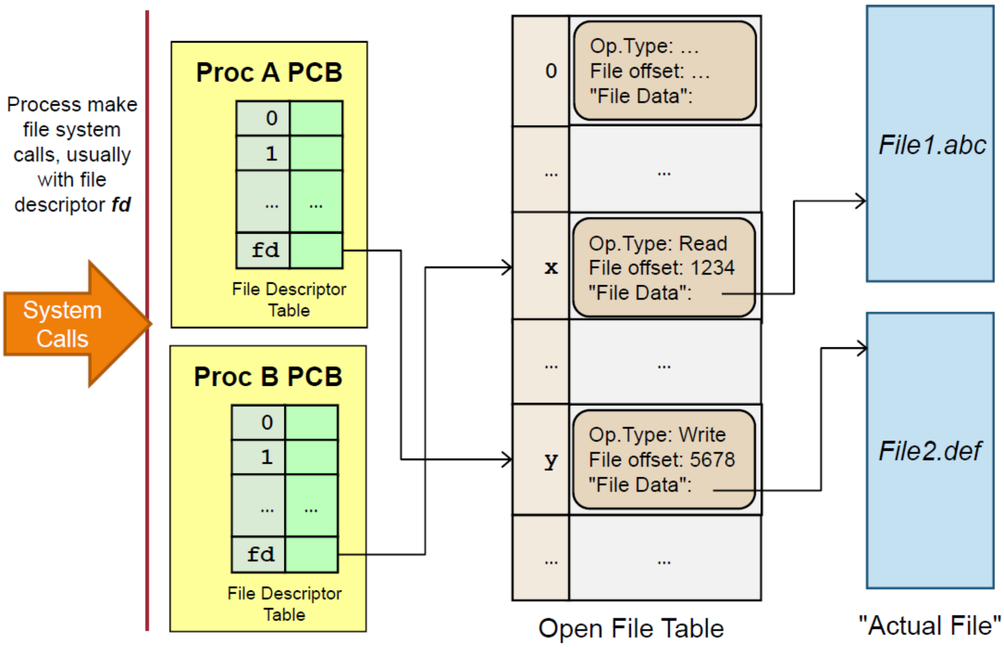
   a. [Generalization] Give one or two examples of buffered file operations found in your favorite programming language(s). Other than the "chunky" read/write benefit, are there any other additional features provided by those high level buffered file operations?

   b. [Application] Take a look at the given "**`weird.c`**" source code. Compiles and performs 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 the ***`printf`***() statement, how does the output pattern changes? How can this information be useful?

   c. [Design] Give **a high level pseudo code** to provide buffered file read operation. Use the following "function header" as a starting point:

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

3. [Open File Table] Below is the illustration taken from lecture 10:



Discuss how this organization helps OS to handle the following scenarios. Your answer should refer to the relevant structure(s) if possible.

a. Process A tries to open a file that is currently being written by Process B.
b. Process A tries to use a bogus file descriptor in a file-related system call.
c. Process A can never "accidentally" access files opened by Process B.
d. Process A and Process B reads from the same file. However, their reading should not affect each other.
e. Redirect Process A's standard input / output, e.g. " **`a.out < test.in > test.out`**". (Hint: *nix processes has 3 default file descriptors: 0 = stdin, 1 = stdout, 2 = stderr)

4. [Cover if time permits - 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?

Questions for your own exploration

1. Explain the following concepts in your own words clearly; the shorter, the better! You 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. (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?