CS2106 Operating Systems
Semester 1 2020/2021

Week of 5th April 2021
Tutorial 10 **Suggested Solutions**
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?

ANS:

a.

W(9, 3) = { 3, 4, 5 }
W(11, 3) = { 2, 3, 5 }
W(9, 4) = { 2, 3, 4, 5}
W(11, 4) = { 2, 3, 5 }

b.

W() gives the number of frames required by a process. So, we can utilize the W() value to:

- When a process become active (e.g. from blocked ➔ running), load all working set pages into frames for that process. Similarly, when the process become inactive (e.g. from running ➔ blocked), we start to migrate those pages from physical frame into secondary storage.
- We can also use the total W() values among all "active" processes, if the total exceeds the physical frame number, we may want to stop allowing more process to become runnable as that may induce thrashing. It is common to refer to the two sets of processes the "active set" and the "inactive set" in this context.

(Optional) The second point above suggests that if the CPU is well utilized and page fault activity is low ➔ **Δ value is well chosen** as the pages in memory are good representation of working set pages for all processes. If the page fault activity is very high (coupled with low CPU utilization) ➔ **Δ may be too small**. Similarly, if both page fault activity and CPU utilization are low ➔ **Δ may be too large** (which prevented more processes to be runnable to utilize the CPU).

c.

The mysterious value indicates whether a page has been used in K memory accesses window. Every K-accesses, the hardware should trigger an interrupt to bring in the OS so that the OS can take note of the W() (essentially all pages with non-zero K-bit).

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"
```

ANS:

a.

C: printf, scanf, fprintf, fscanf

Java: FileInputStream, FileOutputStream

C++: "<<", ">>" stream operators

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.

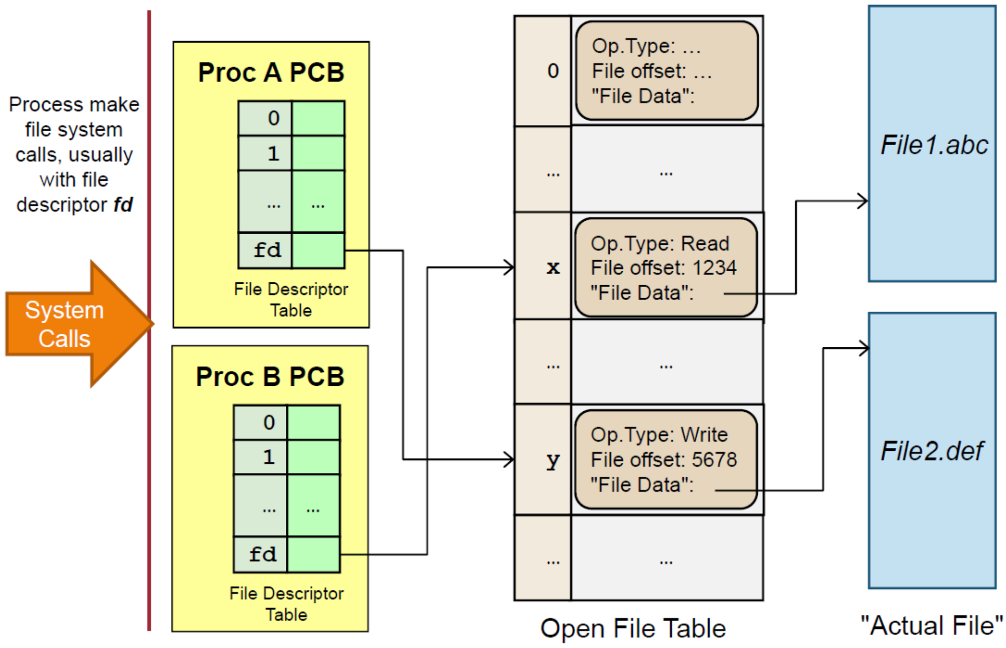
Only key points are mentioned.

BufferedFileRead(File, outputArray, requestSize) //pseudo code
// Read arraySize bytes from File, place the file content 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!
        read(File, Buffer, Buffer.size – Buffer.availableItems)   //low level file reading
        Buffer.availableItems = Buffer.Size   //Buffer is full now

    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 >> Buffer.size
//2. When the file "File" is exhausted (i.e. cannot refill buffer anymore)

//These scenarios can be easily supported by building on the key points above.
```

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)

ANS:

a. OS uses the Open File Table to check for existing opened file. Since the file is already opened for reading, it can reject the file open system call from process A.
b. Since Process A passed the file descriptor (fd for short) to OS as parameter, OS can check whether that particular entry is valid (or even exists) in the PCB of A. If the fd is out of range, non-existent etc, OS can reject the file-related system calls made by Process A.
c. Since the fd index into process specific PCB, there is no way Process A can access Process B's file descriptor table.
d. As discussed in lecture, Process A and Process B can have their own **fd**s, which refers to two distinct locations in the open file table. Each entry of the open file table keep track of the current location separately. This enables Process A and Process B to read from the same file independently.
e. Key points: There are 3 standard file descriptors for every program in Unix (0 = stdin, 1 = stdout, 2 = stderr). These are "opened" automatically and linked to the

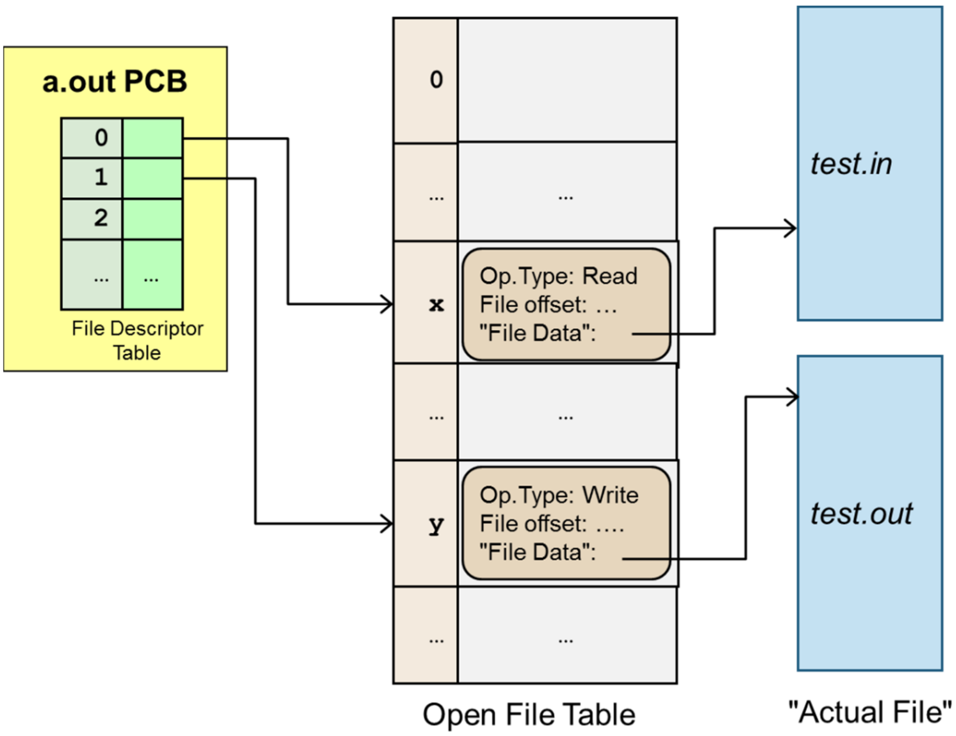
corresponding files. Note that screen (terminal), keyboard are represented as *special files* in unix.

So, for all file redirections, it is a simple question of:

a. Opening and possibly creating the file.
b. Replace the corresponding file descriptor to point to the entry from (1) in the open file table.

The following illustrate the last case:

`a. out < test.in > test.out`



Q4. [Deferred to Tutorial 11]