Tutorial 10

CS2106: Introduction to Operating Systems

Question 1

Virtual Memory: Working Set Model

Question 1(a)

Given the following memory reference string

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

$W(T, \Delta)$ gives the working set at time T, with a window size of Δ.

Give the working set for the following:

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

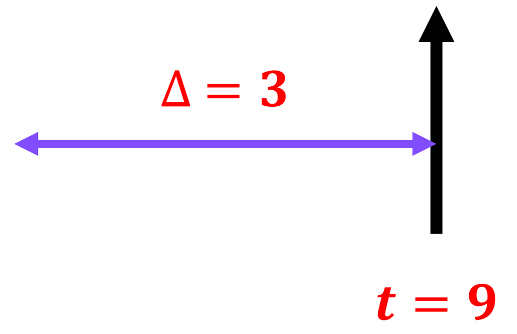
Note:
Over here, the working set is the **active pages** in the interval at time T with a window size of Δ.

Question 1(a)

$W(9, 3)$

- $t = 9$
- $\Delta = 3$

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

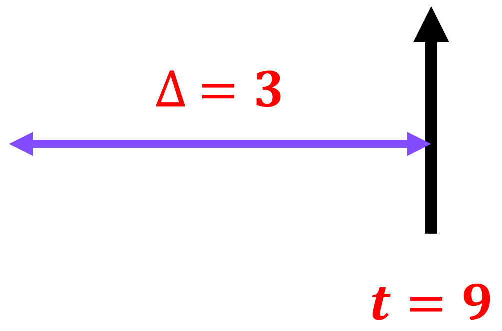


Question 1(a)

$W(9, 3) = \{3, 4, 5\}$

- $t = 9$
- $\Delta = 3$

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

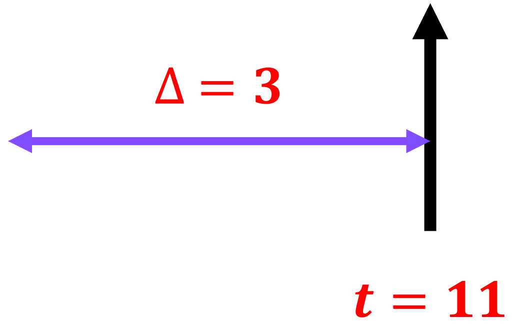


Question 1(a)

$\boldsymbol{W(11, 3)}$

- $\boldsymbol{t = 11}$
- $\boldsymbol{\Delta = 3}$

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

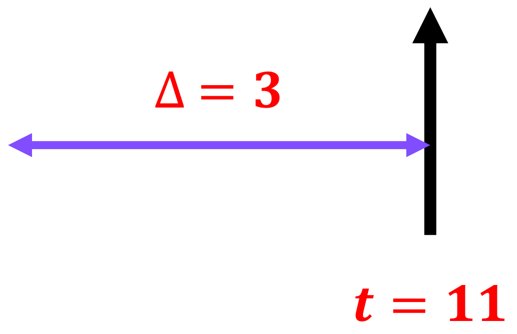


Question 1(a)

$W(11, 3) = \{2, 3, 5\}$

- $t = 11$
- $\Delta = 3$

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

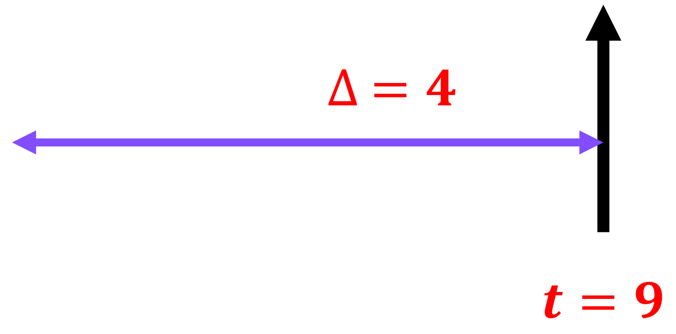


Question 1(a)

$\boldsymbol{W(9,4)}$

- $\boldsymbol{t = 9}$
- $\boldsymbol{\Delta = 4}$

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

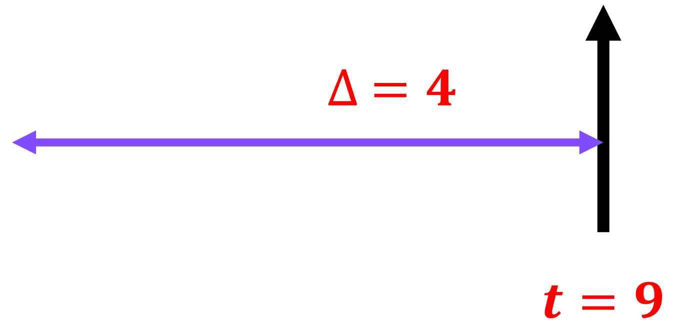


Question 1(a)

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

- $t = 9$
- $\Delta = 4$

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

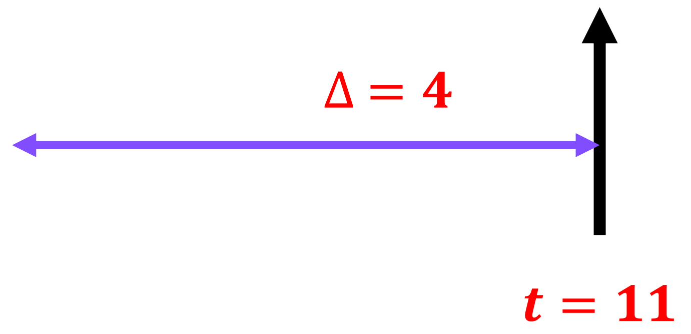


Question 1(a)

$\boldsymbol{W(11, 4)}$

- $\boldsymbol{t = 11}$
- $\boldsymbol{\Delta = 4}$

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

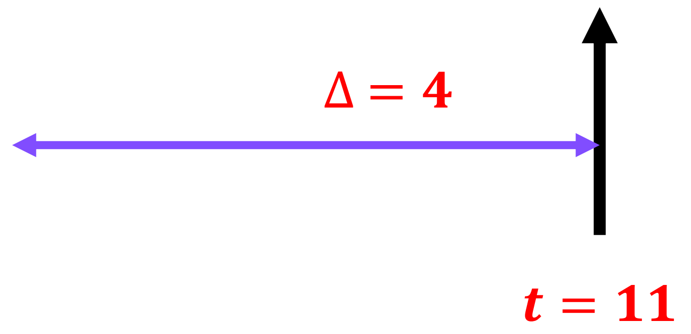


Question 1(a)

$\boldsymbol{W(11,4)} = \{2, 3, 5\}$

- $\boldsymbol{t = 11}$
- $\boldsymbol{\Delta = 4}$

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



Question 1(a)

Given the following memory reference string

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

$W(T, \Delta)$ gives the working set at time T, with a window size of Δ.

Give the working set for the following:

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

Question 1(b)

Hint: In demand paging, the OS only loads a page into RAM upon a page fault. In a new locality, there will be many page faults, is there a way, now that we know the working set for all processes to reduce the number of page faults?

Suppose we know the $W(T, \Delta)$ value for all processes, how can OS use this information?

- $W(T, \Delta)$ gives the number of frames required by a process as well as pages in the working set given time T and window size Δ
- The OS can utilize this value when a process becomes active (e.g. blocked $\rightarrow$ running) to load all working set pages into frames for that process.
- Similarly, when the process become inactive (e.g. from running $\rightarrow$ blocked), the OS can migrate those pages from physical frame into secondary storage.

Question 1(b)

Hint: In demand paging, the OS only loads a page into RAM upon a page fault. In a new locality, there will be many page faults, is there a way, now that we know the working set for all processes to reduce the number of page faults?

Suppose we know the $W(T,\Delta)$ value for all processes, how can OS use this information?

- The OS can also calculate the number of pages required from the working sets of all “active” processes.
- If the total number exceeds the physical frame number, the OS can stop allowing more processes to become runnable or else it would induce thrashing.

Question 1(b): Extension

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

CPU Utilization	Page Fault Activity	Interpretation
High	Low	Δ value is well chosen
Low	High	Δ value may be too small
Low	Low	Δ value may be too big

Question 1(b): Extension

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

- High CPU Utilization, Low Page Fault Activity → Δ value is well chosen
  - The pages resident in memory consist of pages that are actively being used by all processes.
- Low CPU Utilization, High Page Fault Activity → Δ may be too small
  - Some pages that are part of the current locality may not be resident in memory, results in high page fault activity which is I/O bound, so CPU would idle.
- Low CPU Utilization, Low Page Fault Activity → Δ may be too large
  - Working set now includes pages that are not activity used by each process
  - Total demand for frames exceed actual physical memory
  - Fewer processes can fit in physical memory at the same time.
  - Not enough runnable processes to keep CPU busy.

Question 1(c)

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 values 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 values 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. $M_0 \leftarrow 0$

Question 1(c)

Mysterious Algorithm Visualized

T = 0

Page	Frame	3 Bits
1	6	000
2	3	000
3	3	000
4	1	000

T = 1, Access Page #3

Page	Frame	3 Bits
1	6	000
2	3	000
3	3	001
4	1	000

T = 2, Access Page #1

Page	Frame	3 Bits
1	6	001
2	3	000
3	3	010
4	1	000

T = 3, Access Page #2

Page	Frame	3 Bits
1	6	010
2	3	001
3	3	100
4	1	000

Entries with non-zero bits, were pages that were recently accessed.

Question 1(c)

What does the mysterious value represent?

The mysterious value indicates whether a page has been used in K memory accesses windows.

If the above algorithm is handled by the hardware, what should be done every K memory accesses?

Every K-accesses, the hardware should trigger an interrupt to bring in the OS so that the OS can take note of the working set $W()$ (essentially all pages with non-zero K-bit).

Question 2

File Systems: Buffered File Operation

Q2: Context

- File operations are expensive in terms of time
  - Each file operation is a system call, need to change from user to kernel mode.
  - High latency access time to secondary storage
- This results in a strange phenomenon
  - The time taken to perform 100 file operations for 1 item far exceeds the amount of time needed to perform a single file operation for 100 items

Q2: Context

- This motivates the implementation of buffered file operations which can be implemented with 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.

Question 2(a)

- Give one or two examples of buffered file operations found in your favourite programming language(s).
  - C: `printf, scanf, fprintf, fscanf`
  - Java: `FileInputStream, FileOutputStream`

- Other than the "chunky" read/write benefit, are there any other additional features provided by those high-level buffered file operations?
  - Error checking
  - Packing / unpacking of data types.

Question 2(b)

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.

i. Can you explain both the behaviour and the significance of the "trigger" value?

ii. If you add a new line character **`"\n"`** to the **`printf()`** statement, how does the output pattern changes?

iii. How can this information be useful?

Question 2(b)

i. Can you explain both the behaviour and the significance of the "trigger" value?
   - `printf` buffers the user output until the internal buffer is full before actually sending the output to the screen.
   - The trigger indicates the probable size of the internal buffer.

Question 2(b)

ii. If you add a new line character "\n" to the printf() statement, how does the output pattern changes?

- If a newline character is added, the output is performed **immediately**.

iii. How can this information be useful?

- 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 lead to the wrong conclusion ("the while loop is not executed!").
- An interesting way to solve the above issue is to use fprintf and "stderr" (the standard error). Use it and see what happens with the buffering.

Question 2(c)

- 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" bytes from "File" and place the file content in
// "outputArray"
```

- Suppose, there is already an internal buffer called **`Buffer`** with the following attributes
  - **`size`**: capacity of the buffer
  - **`availableItems`**: how much data is current stored in the buffer

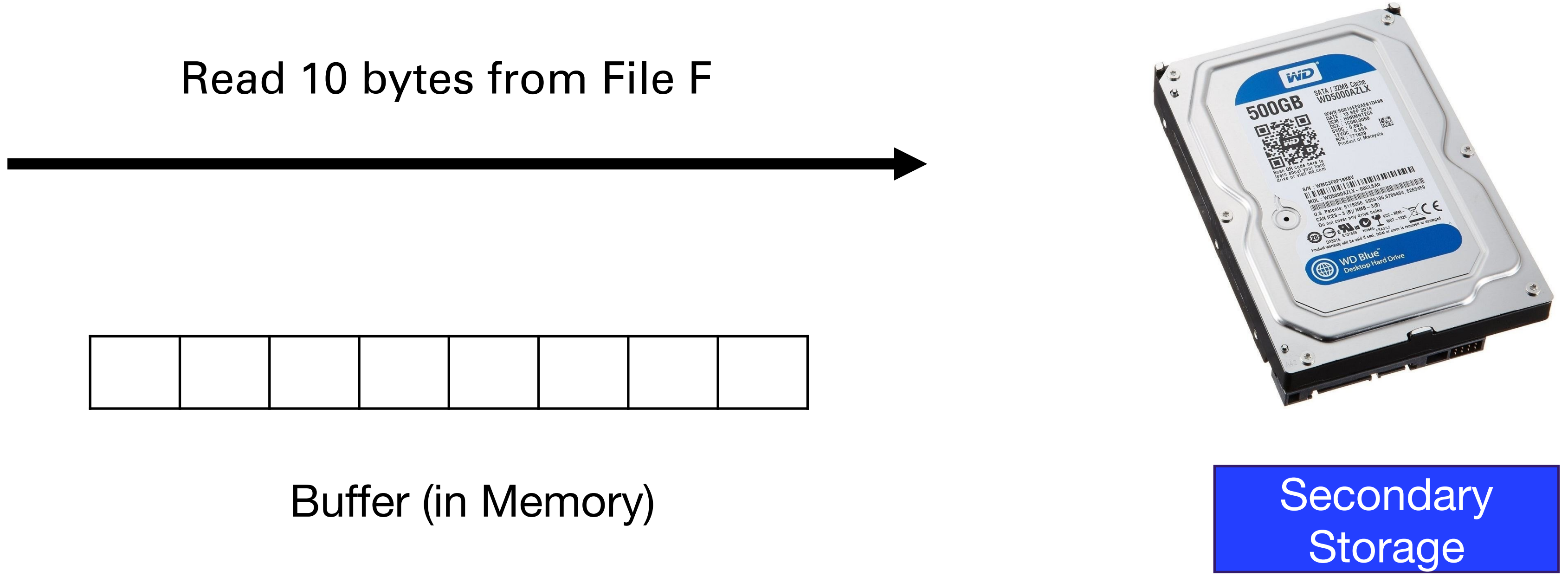
Read Operation: `read()`

- Function Call

  `int read(int fd, void *buf, int n)`

- Purpose
  - reads up to `n` bytes from current offset into buffer `buf`
- Return
  - number of bytes read, can be `0...n`
  - `<n` – end of file is reached
- Parameters
  - `fd` – file descriptor (must be opened for read)
  - `buf` – an array large enough to store `n` bytes
- `read()` is ***sequential read***
  - starts at current offset and increments offset by bytes read

Question 2(c): BufferedFileRead



Question 2(c): BufferedFileRead

Read 10 bytes from File F

$$\longrightarrow$$

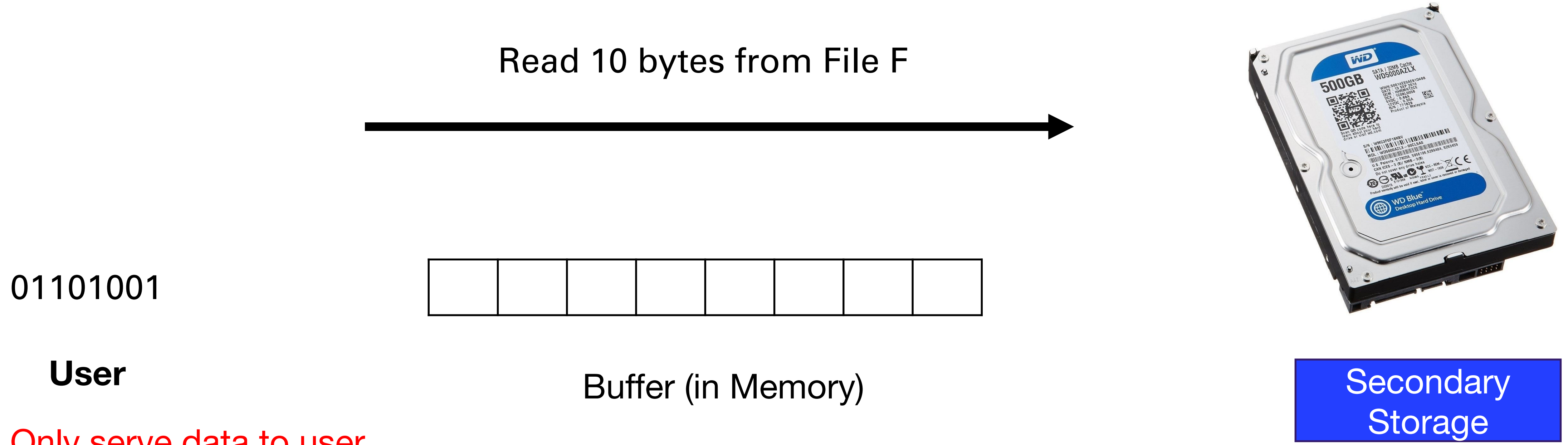
0	1	1	0	1	0	0	1

Buffer (in Memory)

Secondary Storage

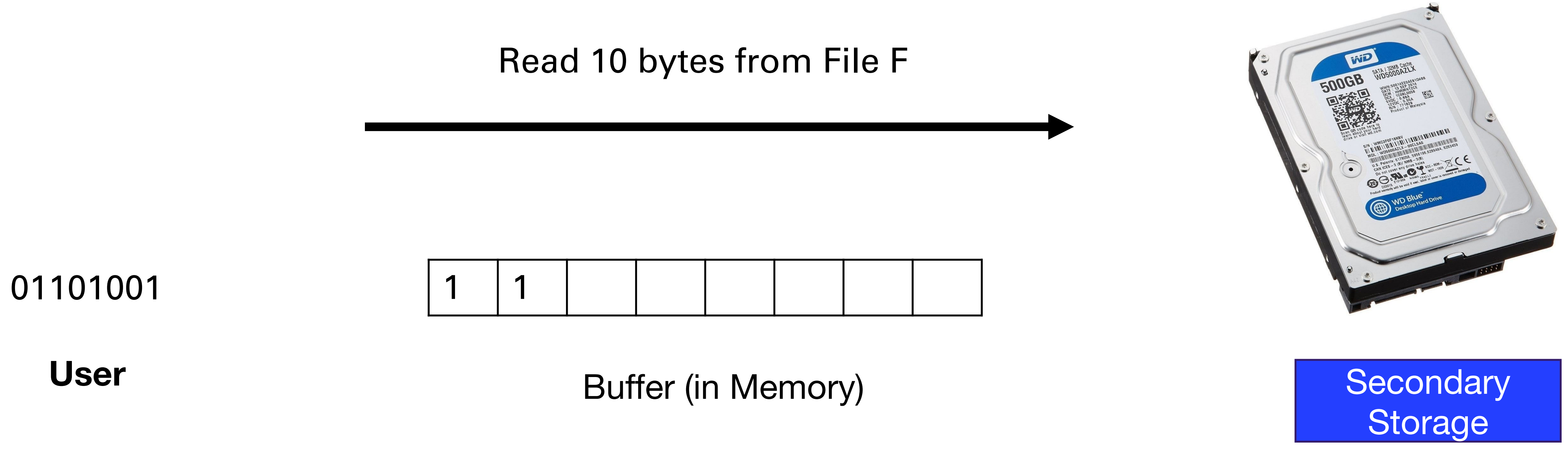
Only serve data to user when buffer is full!

Question 2(c): BufferedFileRead



Only serve data to user when buffer is full!

Question 2(c): BufferedFileRead



Question 2(c)

- Give a high-level pseudo code to provide buffered file read operation.
- Suppose, there is already an internal buffer called **`Buffer`** with the following attributes **`size`** and **`availableItems`**

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

If Buffer.availableItems < arraySize
    read(File, Buffer, Buffer.size – Buffer.availableItems)
    Buffer.availableItems = Buffer.size

copy( outputArr, Buffer, arraySize )
Buffer.available -= arraySize
```

`size`: capacity of the buffer

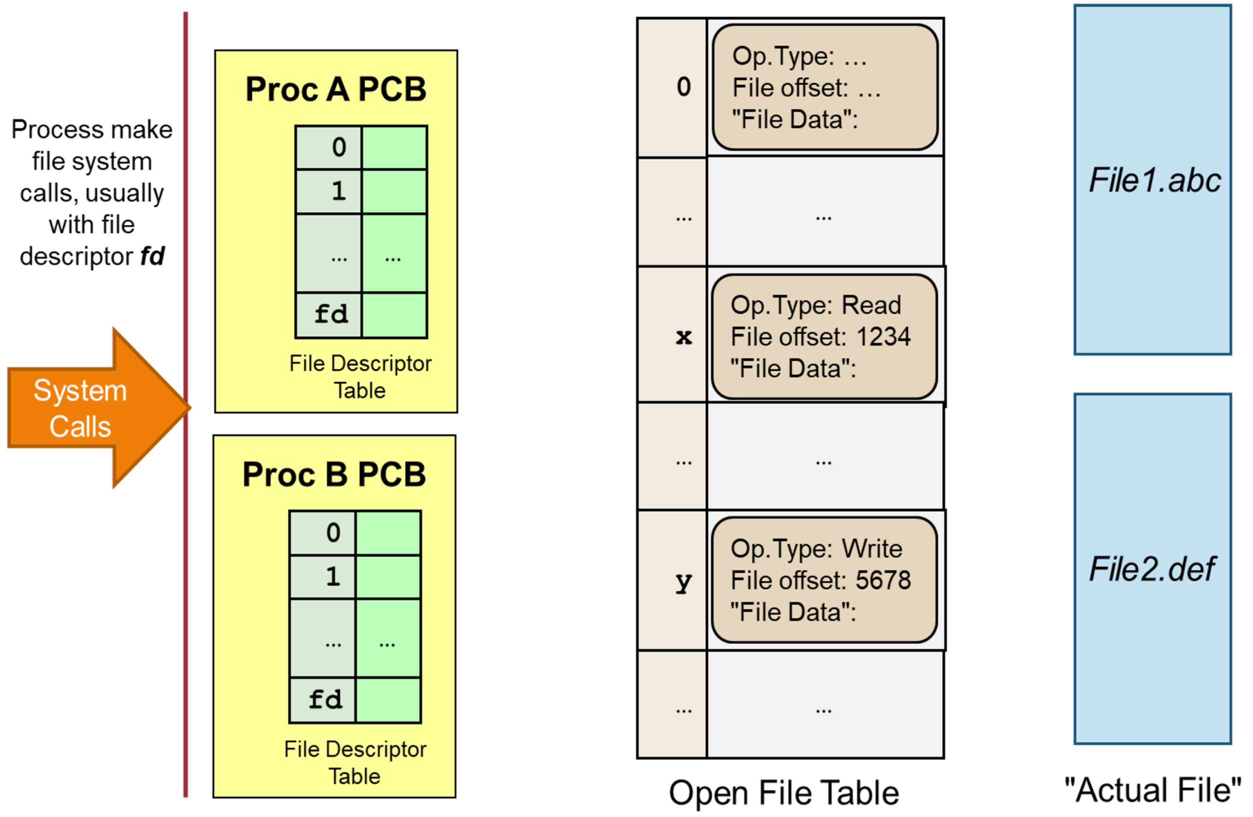
`availableItems`: how much data is current stored in the buffer

Question 3

File Systems: Open File Table

Question 3

Consider two Processes A and B, as well as the following System-Wide Open File Table, with two files **`File1.abc`** and **`File2.def`**.



Question 3

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 read from the same file. However, their reading should not affect each other.

e) Redirect Process A's standard input / output.

Example: "`a.out < test.in > test.out`".

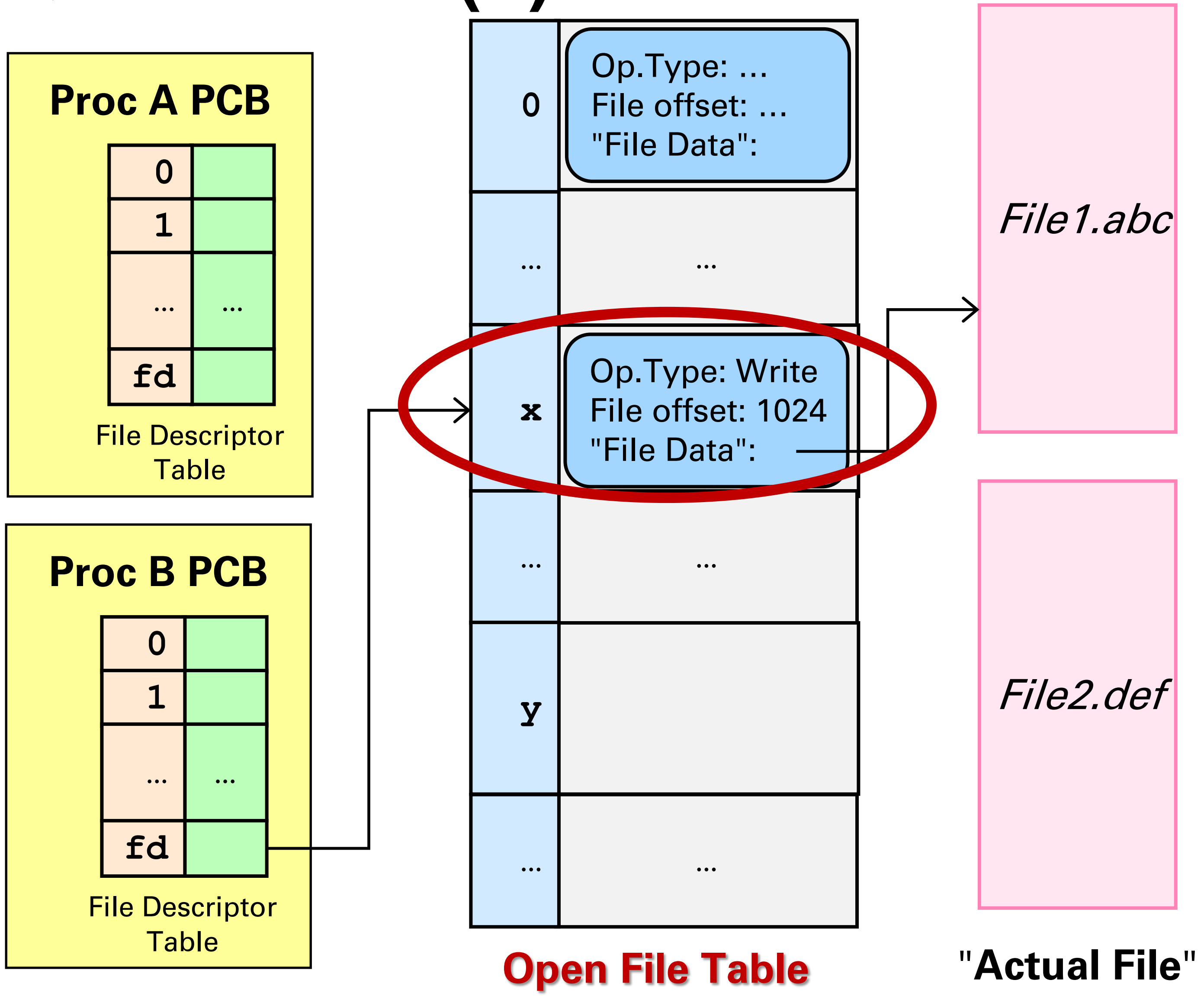
Question 3(a)

Process A tries to open a file that is currently being written by Process B.

- OS uses the Open File Table to check for existing opened file.
- Since the file is already opened by Process B for writing, it can reject the file open system call from process A.

Note: The answer here is a suggested solution to ensure there is no file corruption. by preventing process A from writing to a file that process B is currently writing on.

Question 3(a)



Process A tries to open a file that is currently being written by Process B.

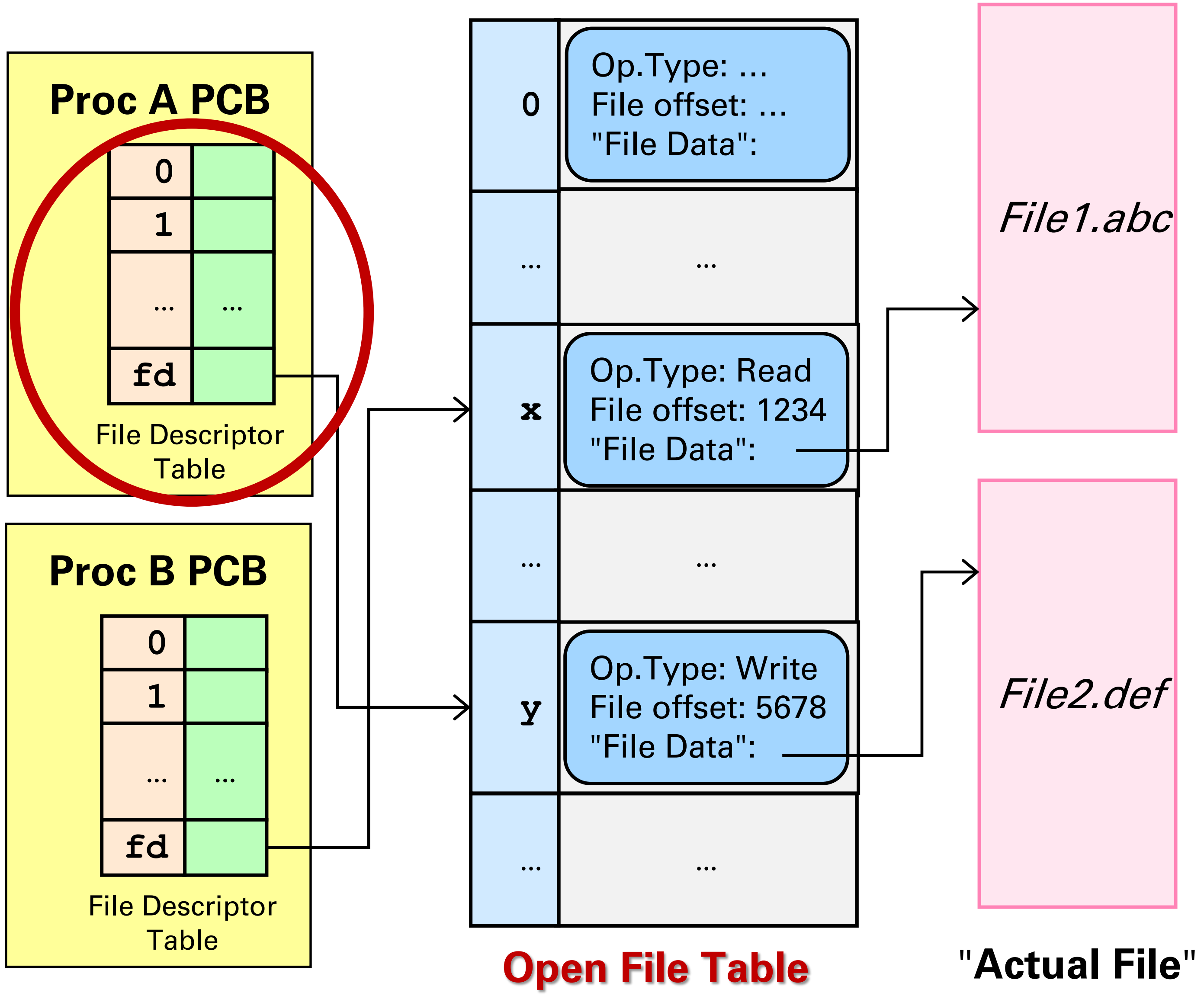
1. OS uses the Open File Table to check for existing opened file.
2. Since the file is already opened by Process B for writing, it can reject the file open system call from process A.

Question 3(b)

Process A tries to use a bogus file descriptor in a file-related system call.

- 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.

Question 3(b)



Process A tries to use a bogus file descriptor in a file-related system call.

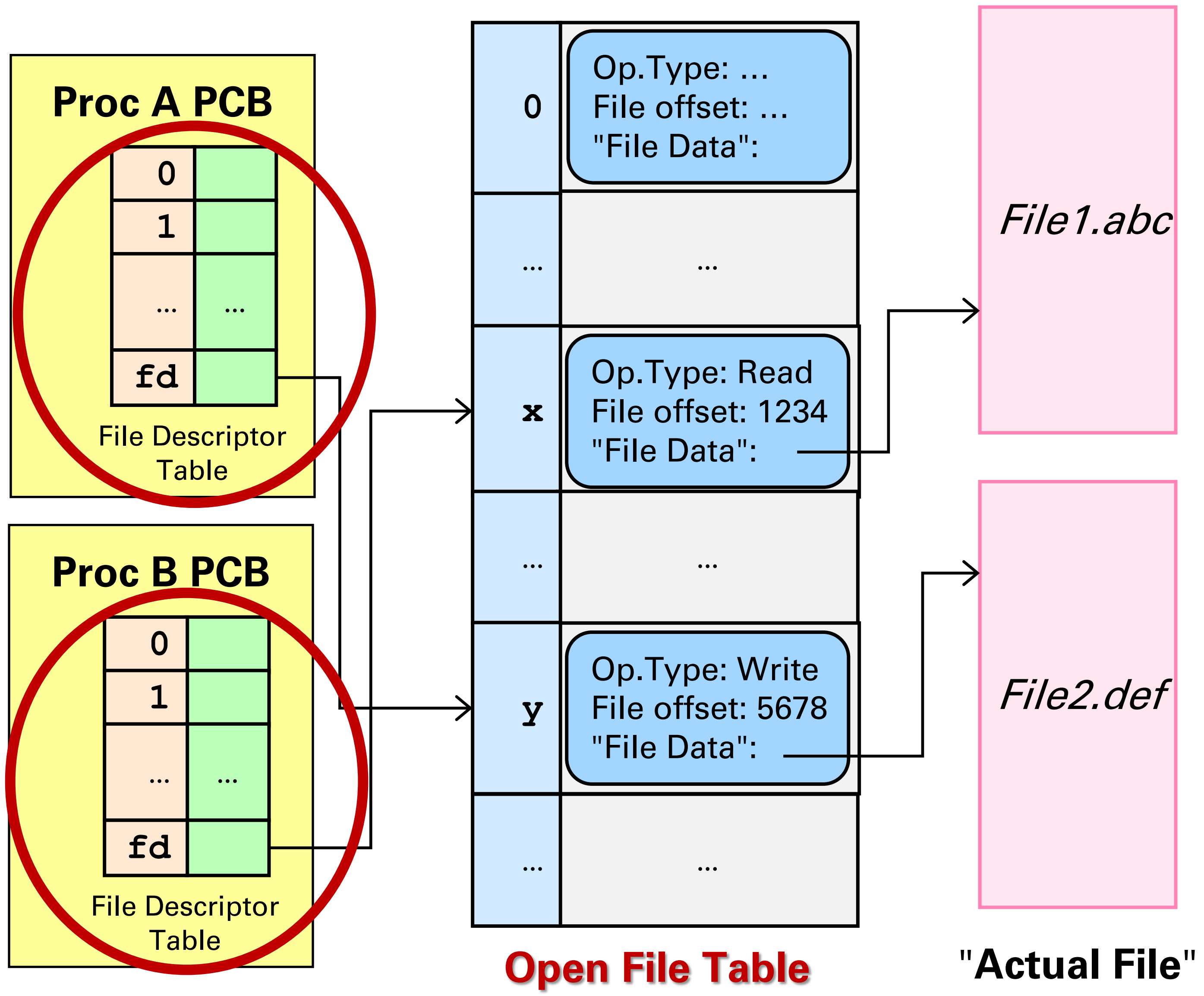
- A file descriptor is a non-negative integer that a process uses to refer to an open file.
- So, if Process A tries to use a bogus file descriptor (i.e. a non valid integer) that does not exist within Process A's file descriptor table, then the OS will reject the file related system call made by Process A.

Question 3(c)

Process A can never "accidentally" access files opened by Process B.

- Since the fd index is in process specific PCB, there is no way Process A can access Process B's file descriptor table.

Question 3(c)



Process A can never "accidentally" access files opened by Process B.

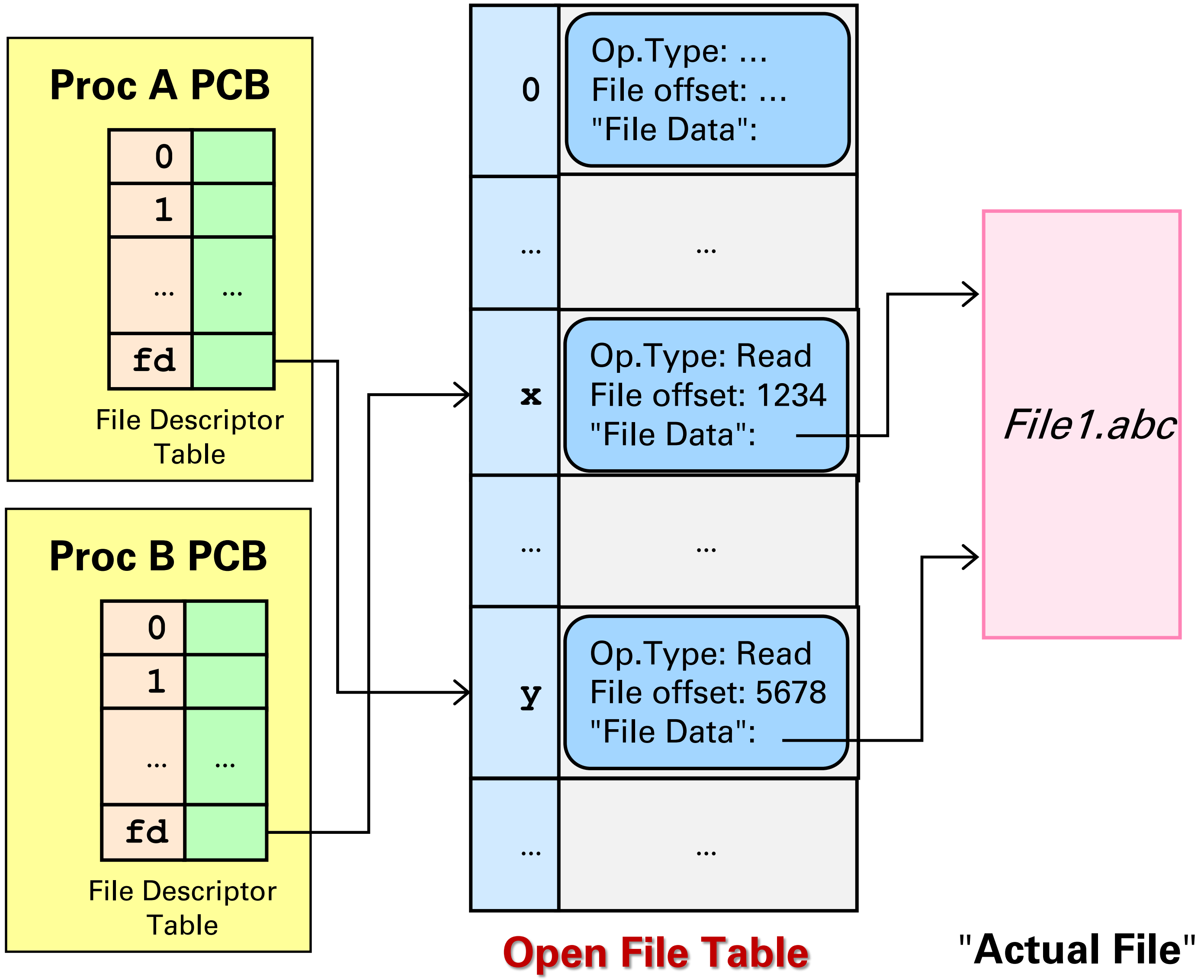
- A file descriptor is a non-negative integer that a process uses to refer to an open file.
- Each process has its own file descriptor table.
- For example, file descriptor index 3 in process A and process B can point to different entries in the Open File Table.
- Process A cannot access Process B’s file descriptors.

Question 3(d)

Process A and Process B read from the same file. However, their reading should not affect each other.

- As discussed in lecture, Process A and Process B can have their own fds, 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.

Question 3(d)



Process A and Process B read from the same file. However, their reading should not affect each other.

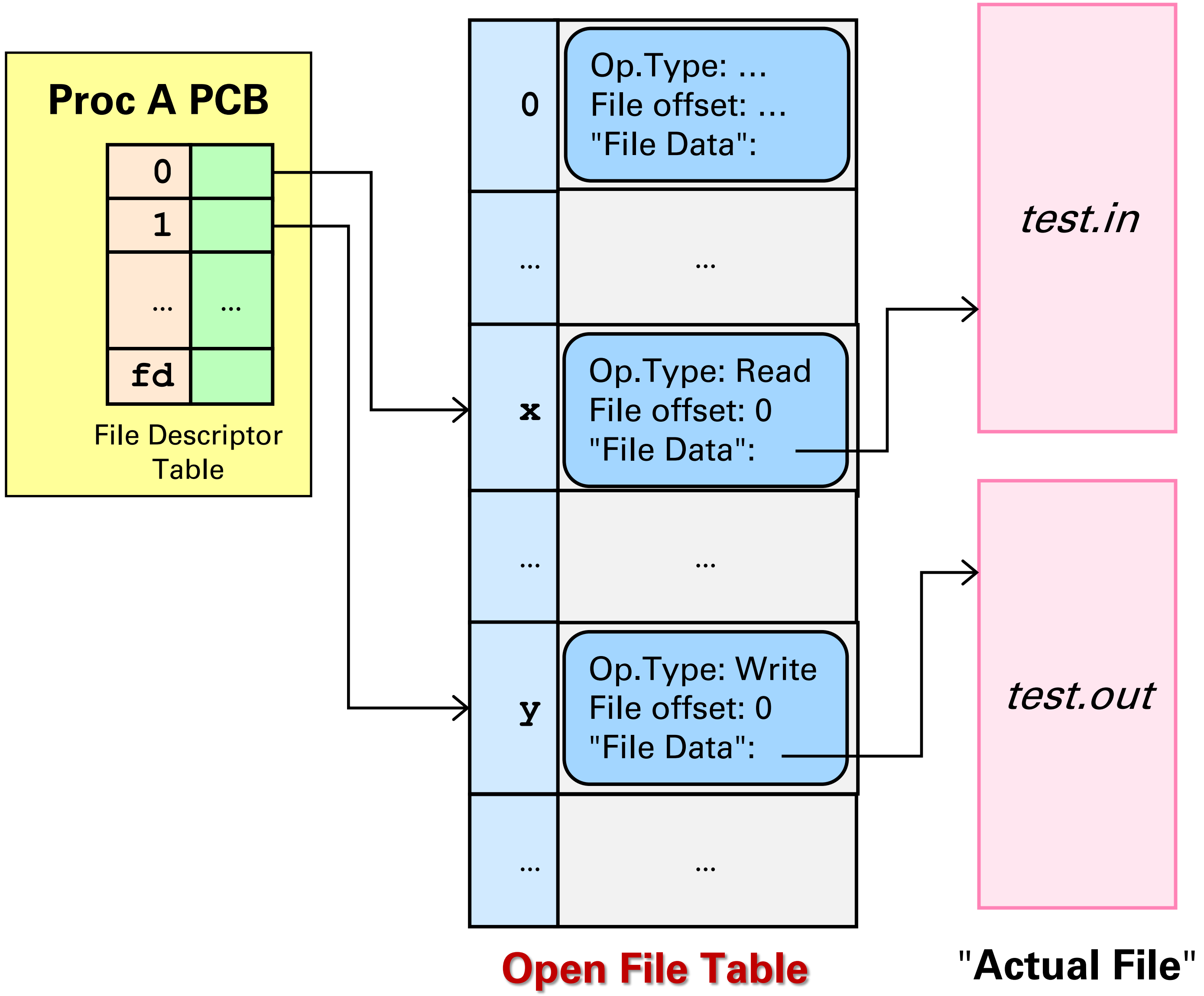
- The same fd number in Process A and Process B can refers to two distinct locations in the open file table.
- Each entry of the open file table keeps track of the current location separately. This enables Process A and Process B to read from the same file independently.

Question 3(e)

Redirect Process A's standard input / output.

- Example: "`a.out < test.in > test.out`".
- 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:
    - Opening and possibly creating the file
    - Replace the corresponding file descriptor to point to the entry from (1) in the open file table.

Question 3(e)



Redirect Process A's standard input / output.

Example: "`a.out < test.in > test.out`".

- For the above example, we are reading from "test.in" into a program called "a.out".
- The output from "a.out" is redirected by writing into a new file called "test.out".

Question 4

File Systems: Directory Permission

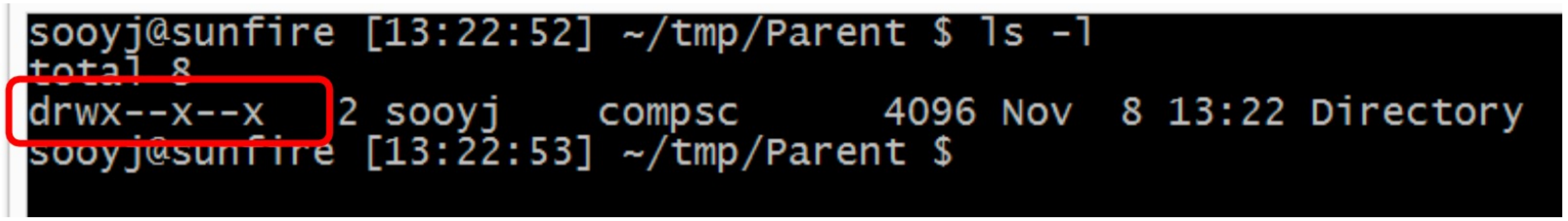
Context

Permission Bits for file and directories in *NIX are in the form of

Owner	Group	Others
RWX	RWX	RWX

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**.

Question 4

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 timestamp.
f) Perform "`touch newfile.txt`" to create a new file.

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

END OF TUTORIAL