

## CS2106: Operating Systems

### Lab 5 – Simple File Operations and Virtual Memory Simulation

#### Important:

- The deadline of submission on IVLE is **6<sup>th</sup> November 5pm**
- The total weightage is 6%:
  - o Exercise 1: 1% [Lab Demo Exercise]
  - o Exercise 2: 1%
  - o Exercise 2: 4%
- System/OS restriction: Use **any Unix based** OS for this lab. You can either use **Sunfire** or any **Linux** installation. Be reminded to test your code on a Linux before submission.

## Section 1. Overview

To celebrate the end of lab session for CS2106, this lab should be "a bit lighter" on your time (theoretically). Exercise 1 and 2 are two simple tasks on file operations. The lab demo exercise 1 probably takes less than 5 lines of code ☺.

Exercise 3 is a simulator of virtual memory management. In an attempt to let you understand this topic better, a **working simulator** is provided. Your task is just to improve some parts of the simulator.

## Section 2. Exercises in Lab 5

### 2.1 Exercise 1 [Lab Demo Exercise]

Take some time to familiarize yourself with the following Unix file related system calls:

- **read()** : Reading data from an opened file descriptor.
- **lseek()** : Move to a specified location in the file You can read through the lecture slide and/or the relevant man pages on Unix.

Hint: You do not need any other file-related system calls to solve exercise 1 and 2.

Take a look at the sample execution session below. User input in **bold** font.

| Sample Run 1   |  |
|--|--|
| File Name: <b>10int.dat</b><br>Size = 40 bytes<br>123<br>124 | One of the provided input data file.<br>File size is printed<br>Data in file are read and printed (a total of 10 integers in this case). |

|     |  |
|-----|--|
| 125 |  |
| 126 |  |
| 127 |  |
| 128 |  |
| 129 |  |
| 130 |  |
| 131 |  |
| 132 |  |

|  |  |
|--|--|
| Sample Run 2                               |  |
| File Name: <b>wrong.dat</b><br>Cannot Open | This is a non-existent file.<br>Complain and exit. |

The given skeleton file **ex1.c** has quite a large chunk of logic implemented. Your tasks are essentially:

1. Check that the file can be opened.
2. Find out the file size (hint: use **lseek()**).
3. Read all data until end of the file.

### **A note on 32-bit vs 64-bit**

The exercises in this lab are sensitive to the word size of the underlying execution environment. As some of you may have a 64-bit processor and OS, it is a little tricky to ensure the lab work across both 32-bit and 64-bit environments. For maximum compatibility, we have decided to stick to 32-bit for this lab.

To ensure the correct execution environment, the first part of the skeleton source will do a simple check on the integer size and warn you if your machine + compiler operates in 64-bit. It will terminate the program if 64-bit environment is detected.

The remedy is quite simple, you just need to compile your source code with an additional flag "-m32", e.g.

```
gcc ex1.c -m32 //compiles as 32-bit application.
```

Please make sure you compile your code correctly.

### **For your own exploration:**

1. If you open up the **10int.dat** in a normal editor, what do you see? Can you explain?

## 2.2 Exercise 2

### Program Specification:

This is a follow up from exercise 1. You probably need to copy part of the code over. In this exercise, your task is to write a program to read data from a file according to user requests.

We will be dealing with two types of file:

1. **Character file:** Each data item is one byte. You can just treat it as the **char** type in C for simplicity.
2. **Integer file:** Each data item is one word (4 bytes). You can just treat it as the **int** type in C for simplicity.

After a file is successfully opened, a user can issue the following commands:

| 1. Read Command:   |
|--|
| Input format: <b>1 N S</b> <ul style="list-style-type: none"> <li>• 1 = indicates read command</li> <li>• N = Number of items to read</li> <li>• S = Item size (either 1 and 4 only) (Note: <math>N*S \leq 100</math>)</li> </ul> To read N items of size S each. Notes: <ul style="list-style-type: none"> <li>• Lesser than N items may be read near the end of file.</li> <li>• If no item can be read (i.e. already at the end of file), this operation can fail.</li> </ul> |
| Output format: <ul style="list-style-type: none"> <li>• <b>Successful:</b> All read items are printed on a single line with one space after each item.</li> <li>• <b>Not successful:</b> Print “failed”.</li> </ul>  |

| 2. Move Forward Command:  |
|---|
| Input format: <b>2 N S</b> <ul style="list-style-type: none"> <li>• 2 = indicates move forward command</li> <li>• N = Number of items</li> <li>• S = Item size (either 1 and 4 only)</li> </ul> To skip over N items of size S each. Notes: <ul style="list-style-type: none"> <li>• If the move does not exceed the file size, then the current file position will be advanced accordingly.</li> </ul> |
| Output format: <ul style="list-style-type: none"> <li>• <b>Successful:</b> The current file offset (in bytes) will be printed.</li> <li>• <b>Not successful:</b> Print “not allowed”.</li> </ul>  |

| 3. Move Backward Command:  |
|--|
| Input format: <b>3 N S</b> <ul style="list-style-type: none"> <li>• 3 = indicates move backward command</li> <li>• N = Number of items</li> <li>• S = Item size (either 1 and 4 only)</li> </ul> |

To go backward N items of size S each in file. Notes:

- If the move does not exceed the file starting point, then the current file position will be rewound accordingly.


Output format:







- **Successful:** The current file offset (in bytes) will be printed.
- **Not successful:** Print “not allowed”.

The program will continue to accept user commands until the user ends the input by [Ctrl-D]. The main loop is already coded for you in the ex2.c.

### Sample Program Run

Two sample data files: **character.dat** and **integer.dat** are provided for you. The first file, **character.dat** contains 62 characters, ‘0’ to ‘9’, then ‘a’ to ‘z’, followed by ‘A’ to ‘Z’. The second file, **integer.dat** contains 1000 integers from 0 to 999.

User input is in **bold font** in the following sample sessions. For sample run 1, the spaces and newline characters are explicit shown as “\_” and “

| <b>Sample Run 1</b>   |  |
|---|--|
| File Name: <b>character.dat</b>   | //Prompt user for file name                |
| Size = 62 bytes   | //Show the file size if file can be opened |
| <b>1 5 1</b>  | //Read 5 items of 1 byte each              |
| 0_1_2_3_4_   | //Items read and printed                   |
| <b>2 3 1</b>  | //Skip 3 1-byte items                      |
| 8            | //Move forward ok, now at offset 8         |
| <b>3 10 1</b>   | //Move back 10 items of 1 byte each        |
| not allowed  | //Move failed: file start overshoot        |
| <b>2 50 1</b>   | //Skip 50 1-byte items                     |
| 58           | //Move ok, now near the end of file        |
| <b>1 10 1</b>   | //Attempt to read 10 items of 1-byte       |
| W_X_Y_Z_     | //Only 4 items can be read and printed     |
| <b>2 1 1</b>  | //Try to move forward                      |
| not allowed  | //Failed as already at the end of file     |
| <b>[Ctrl-D]</b>   | //End the program                          |

| <b>Sample Run 2</b>           |                                       |
|-------------------------------|---------------------------------------|
| File Name: <b>integer.dat</b> |                                       |
| Size = 4000 bytes             |                                       |
| <b>1 5 4</b>                  | //Read 5 items of 4 bytes each        |
| 0 1 2 3 4                     | //Items read and printed              |
| <b>2 990 4</b>                | //Skip 990 4-bytes items              |
| 3980                          | //Now at 3980 offset                  |
| <b>3 1 4</b>                  | //Move back 1 item of 4 bytes         |
| 3976                          | //Move ok                             |
| <b>1 10 4</b>                 | //Attempt to read 10 items of 4-bytes |
| 994 995 996 997 998 999       | //6 items read and printed            |
| <b>3 1000 4</b>               | //Move all the way back to file start |

|          |                               |
|----------|-------------------------------|
| 0        | //Success. At file start now. |
| 1 1 4    | //Read 1 item                 |
| 0        | //0 is the first item         |
| [Ctrl-D] | //End the program             |

| <b>Sample Run 3</b>           |                                   |
|-------------------------------|-----------------------------------|
| File Name: <b>notHere.dat</b> | //Try to open a non-existent file |
| Cannot Open                   | //Report error and exit program   |

### For your own exploration:

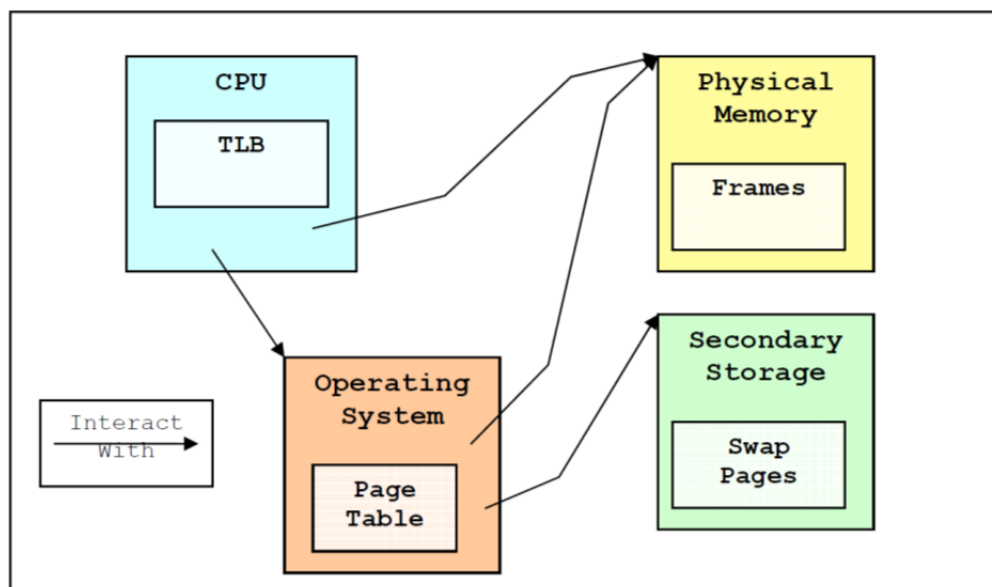
1. If you use the program on a character file (items of 1-byte), then try to read them as item of 4-bytes, what do you get? Why?
2. If you use the program on an integer file (4-byte items), then move forward 1 byte. Can you read any meaningful integer afterwards? Why?

## 2.3 Exercise 3

A simple virtual memory management simulator is provided for you in ex3.c. The simulator implemented the following mechanisms:

- Page Table and TLB.
- Page Replacement for physical memory frames.
- Swap a memory resident page to secondary storage.
- Load a swap page from secondary storage to physical memory.

The major components simulated are shown in the following diagram:



The simulator assumes the following:

- Memory is word addressable (i.e. each address stores a **word**).
  - Words is represented as an integer for simplicity
- Logical Memory space is 256 words

- Page Size = Frame Size = 8 words
- Total logical pages =  $256/8 = 32$  pages
- Total physical memory frames = 8
  
- Page table has 32 PTE
- TLB can hold 4 entries of PTE
  
- Secondary storage stores the logical pages in consecutive order. i.e. Logical Page 0 will be stored as swap page 0, etc...

### **Brief Description of Components**

#### **Secondary Storage:**

- Stores swap-out pages. For simplicity, we assume the total number of swap pages == total number of logical pages. So, there is enough space on the swap file to hold 32 pages.
- Provide functions to read/write a single page in one go.

#### **Physical Memory:**

- Contains 8 frames, each frame contains 8 words.
- Provide functions to:
  - Read/write a single frame in one go.
  - Read/write a single word.
- Memory address is specified by frame number and offset.

#### **Operating System:**

- Only functionalities and information relevant for virtual memory management are simulated.
- We assume there is only one process running, hence only 1 page table is maintained.
- Provides functions to:
  - Load a program for “execution” (more later).
  - Search for a PTE given page number (used by CPU).
  - Handle page fault.

#### **CPU:**

- Maintain the TLB
- Provide function to:
  - Simulate memory read/write access.

### **General flow of “program execution” in the simulator**

An “executable” in this case is simulated as a file containing a number of words (integer). The first value  $N$  specifies the number of words in the file, followed by  $N$  integers.

This executable is loaded by the OS into the swap file initially (i.e. no page are loaded into the physical memory). When memory access causes a page fault, the relevant swap file will then be loaded into physical memory. This is a scheme used in real OS, commonly known as **demand paging**. Under this scheme, all logical pages are non-

memory resident initially and brought into memory only when required ("demanded" by needs).

Memory accesses are then simulated from user input. Users of the simulator can generate read/write memory accesses to observe the changes to the major components in the simulator.

**Important note: You will learn about the general steps to handle memory access with virtual memory in tutorial 8 question 1a. Alternatively, you can dive right into the code and figure out the essential steps.**

### Sample Session with the Simulator

The easiest way to understand the working of simulator is to play with it 😊. Compile the simulator by “gcc -Wall ex3.c”.

You are provided with two sample “executables”:

- p1.exe, which contains 54 words (1 to 54).
- p2.exe, which contains 245 words (100 to 344).

Below is a sample session with explanation. User input is in **bold font**. Additional newlines are sometimes added to the session output to make it more readable.

|  |   |
|--|---|
| Enter swap file name: <b>swap.dat</b>    | An actual file will be created to store the swap pages. |
| Enter program file to run: <b>p1.exe</b> | This is one of the "executable" provided.               |
| Enter number of memory access: <b>5</b>  | Simulate 5 memory accesses.                             |

The simulator accepts the following six commands at this point:

|   |   |
|---|---|
| <b>1. Read Access:</b><br>Input Format: <b>1 P O</b><br><b>1</b> = read access<br><b>P</b> = Page Number<br><b>O</b> = Offset | <b>2. Write Access:</b><br>Input Format: <b>2 P O V</b><br><b>2</b> = write access<br><b>P</b> = Page Number<br><b>O</b> = Offset<br><b>V</b> = Value to be written |
| <b>3. Show current TLB entries.</b>   | <b>4. Show current Page Table Entries.</b>  |
| <b>5. Show current Physical Memory Frames.</b>  | <b>6. Show current swap pages.</b>  |

Note that only (1) and (2) counts as memory accesses. So you can execute (3) – (6) as many times as you want.

|     |                               |
|-----|-------------------------------|
| ... | Session continues from above. |
|-----|-------------------------------|

|   |   |
|---|---|
| <p><b>6</b></p> <p>Swap File with 32 Swap Pages</p> <pre> ----- Swap Page 0 ----- 1 2 3 4 5 6 7 8 ----- Swap Page 1 ----- 9 10 11 12 13 14 15 16 ----- Swap Page 2 ----- 17 18 19 20 21 22 23 24 ----- Swap Page 3 ----- 25 26 27 28 29 30 31 32 ----- Swap Page 4 ----- 33 34 35 36 37 38 39 40 ----- Swap Page 5 ----- 41 42 43 44 45 46 47 48 ----- Swap Page 6 ----- 49 50 51 52 53 54 0 0 ----- Swap Page 7 ----- [ not shown ] </pre> | <p>Let's take a look at the swap pages first.</p> <p>"p1.exe" contains 54 words stored in 6 swap pages.</p> <p>Each logical memory word is represented as a integer number. In this case, they happen to contain "1" to "54".</p> <p>This is the end of the logical memory space of "p1.exe".</p> |
| <p>...</p> <p><b>4</b></p> <pre> ----- Page Table ----- Page 0  DISK[0] Page 1  DISK[1] Page 2  DISK[2] Page 3  DISK[3] Page 4  DISK[4] Page 5  DISK[5] Page 6  DISK[6] Page 7  INVALID [ not shown ] </pre>  | <p>Session continues.</p> <p>Let's take a look at the page table entries.</p> <p>All pages are now in disk. Disk[X] = the page is in disk swap page X.</p> <p>Page 7 onwards is out of range for p1.exe.</p>  |
| <p>...</p> <p><b>3</b></p> <pre> --- Translation Look-Aside Buffer --- Entry 0  INVALID Entry 1  INVALID Entry 2  INVALID Entry 3  INVALID </pre>   | <p>Session continues.</p> <p>Now, it's time to inspect TLB entries.</p> <p>All entries are invalid initially.</p>   |



|   |   |
|---|---|
| <pre> ...  <b>5</b> Physical Memory with 8 frames ----- Frame 0 ----- 0 0 0 0 0 0 0 0 [ not shown ] </pre>  | <p>Session continues.</p> <p>What's in physical memory?</p> <p>All frames unused at this point. Recall the idea about "demand paging".</p>  |
| <pre> ...  <b>1 3 4</b> Read 3.4: 29  <b>3</b> --- Translation Look-Aside Buffer --- Entry 0  =&gt; PTE[ 3]: RAM[0] Entry 1  INVALID Entry 2  INVALID Entry 3  INVALID <b>4</b> ----- Page Table ----- Page 0  DISK[0] Page 1  DISK[1] Page 2  DISK[2] Page 3  RAM[0] Page 4  DISK[4] [ the rest not shown ] <b>5</b> Physical Memory with 8 frames ----- Frame 0 ----- 25 26 27 28 29 30 31 32 [other frames not shown] </pre> | <p>Session continues.</p> <p>Simulate a CPU read from page <b>3</b> offset <b>4</b>.<br/>Value read = 29</p> <p>Let's find out the effect of the read access above.</p> <p>Page 3 is now in frame 0.</p> <p>Check page table.</p> <p>Page 3 is in RAM</p> <p>Content of page 3 is now in physical memory frame 0.</p> |
| <pre> ... <b>2 2 5 321</b> Write 321 -&gt; 2.5 : ok <b>3</b> --- Translation Look-Aside Buffer --- Entry 0  =&gt; PTE[ 2]: RAM[0] Entry 1  INVALID Entry 2  INVALID Entry 3  INVALID <b>5</b> </pre>  | <p>Session continues.</p> <p>Write 321 to page 2 offset 5.<br/>Write successful.</p> <p>Page 2 is in frame 0</p>  |

|  |  |
|--|--|
| Physical Memory with 8 frames<br>----- Frame 0 -----<br>17 18 19 20<br>21 321 23 24<br><br>..... <i>Sample Session Ended</i> ..... | Check memory for the value written.<br><br>Note the new value 321. |
|--|--|

### Your task in exercise 3

You can see that the simulator is working “ok” at this point. However, you should have noticed there is something strange about the second write request. The request writes to a non-memory resident page, which should be brought into memory (we assumes no **write-around**). Even though the page is correctly brought into the physical memory, it should not replace page 3 (loaded by the first read request) as there are many other free frames! Your task is to fix a few minor flaws like these.

The simulator, though simplistic, is still quite big (700+ lines). So, it is hard to understand all the code at once. A good way to get started is:

1. Browse through the structure definitions at the top of the program.
  - Find out what components are defined and the information they contain.
2. Each structure definition has a set of associated functions:
  - Read through the function declarations.
  - Most are quite self-explanatory.
3. Zoom into the function **readMemAccess()** function (line 646), and the **findFrameNumber()** (line 594). These are the starting point of the whole memory access mechanism. I have written (too) many comments, which hopefully can help you to get the idea easily.

### **You need to add/fix three things:**

1. Print out the following statistics at the end of the simulation:
  - a. Total number of **successful** memory access.
  - b. Percentage of TLB-Miss (2 place of precision).
    - $\text{Number of TLB-Miss} / \text{Total Successful Access} * 100\%$
  - c. Percentage of Page-Fault (2 place of precision)
    - $\text{Number of Page-Fault} / \text{Total Successful Access} * 100\%$
2. TLB Replacement (in **findFrameNumber()** at line 594)
  - a. Currently the 0<sup>th</sup> TLB entry is always replaced when there is a new PTE coming in. Fix this.

- b. Details already in code.
3. Page Replacement (in `handlePageFault()` at line 503)
- a. Currently the first physical frame always gets replaced. Fix this using FIFO page replacement algorithm.

The tasks are marked by a comment with “**//TODO**” in the source code so you can easily locate them. Further details are also included in the comments.

### **Endianness – You still remember them?**

In addition to 32-bit/64-bit environment, this exercise is also sensitive to the **endianness** of the processor. In a nutshell, endianness determines the order in which processor store **multi-byte** data item in the memory. Do not worry, the only thing you need to do is to comment/uncomment the “`#define LITTLEENDIAN`” at the beginning of `ex3.c`. If you use any intel based machine (including most laptops and macbooks), then you should leave this macro alone. If you compile the program on Solaris (Sparc), you need to comment off this macro as Sparc is a Big Endian processor. **Please remember to uncomment this macro before submission.**

### **Reality Check**

Other than being a scaled down version of the real thing, the simulator tries to mimic the steps as closely as possible. You can also see that the connections between components are restricted to give a better understanding, e.g. CPU only access physical memory (i.e. CPU has link to memory but not to secondary storage).

However, due to complexity, there are points that are simplified. For example:

- Most hardware nowadays uses a MMU (Memory Management Unit) that performs the mapping of logical address to physical address (i.e. checks of TLB etc).
- The swap pages may not be in consecutive order. Also, a logical page may not be stored at the same place every time. OS can choose to manage the swap pages like the use of memory frame (i.e. keep track of free swap pages and select a suitable page when needed).
- The page table is in stored in physical memory, though the simulator shows it as “outside” of the physical memory.
- The OS can be involved in TLB-entry replacement (known as **software TLB management**), where the choice of TLB entry is decided by OS.

### Section 3. Submission

Zip the following files as A0123456.zip (**use your student id!**):

**a. ex2.c**

**b. ex3.c**

Upload the zip file to the "Student Submission→Lab 4" workbin folder on IVLE. Note the deadline for the submission is **6<sup>th</sup> November, 5pm**.

Again, please ensure you have followed the instructions carefully (output format, how to zip the files etc). Deviations will cause unnecessary delays in the marking of your submission.