CS2106: Operating Systems
Lab 4 – Virtual Memory Management

Important:

- The deadline of submission through LumiNUS is **Sat, 2 Nov, 2pm**
- The total weightage is **6% + [Bonus 1%]:**
  - Exercise 1: 2% **[Lab Demo Exercise]**
  - Exercise 2: 2%
  - Exercise 3: 2%
  - Exercise 4: (Bonus) 1%

Section 1. Overview

There are **three exercises (+ one bonus)** in this lab. The purpose of this lab is to learn about the page table, and how the operating system populates the page table when a page fault occurs. In exercises 1 and 2, you are going to implement a page fault handler using the second-chance page replacement algorithm described in the lecture. In exercises 3 and 4, you will augment your existing code to handle dynamic page allocation and deallocation. Each exercise in this lab builds on the previous exercise and is designed to guide you along.

General outline of the exercises:

- Exercise 1: Manage a fixed amount of read-only memory
- Exercise 2: Ex1 + the user process can write to the memory
- Exercise 3: Ex2 + the user process can allocate/deallocate memory (simulates mmap()/munmap())
- Exercise 4: Ex3 + mmap()'ed memory should commit-on-write

As each exercise builds on the previous one, it is strongly recommended to read the whole document before you start working on this lab assignment. Appropriate design decisions taken in earlier exercises may significantly reduce the work in later exercises.

1.1 Grading

You should submit three or four files (**ex1.c**, **ex2.c**, **ex3.c**, and **ex4.c**), placed inside a .zip folder as described at the end of this document. For each exercise, we will compile your code with our version of **runner.c** and header files, and run the resulting executable against our test cases for that exercise. You will obtain the marks for an exercise if it passes our test cases for that exercise (partial credit may be given if your code has bugs that only manifest on corner cases). Test cases are cumulative – our set of test case cases for each exercise is a superset of the set of test cases for the exercise

before it. One of the two marks allocated to exercise 1 will be awarded for the demonstration to your lab TA.

A file (ex1-4.c) for each exercise is required to avoid cases where you break your code for ex2 in the process of implementing ex3. If you successfully implement ex1 to ex4, you can submit copies of the same code in **ex1-4.c** (as each exercise is a superset of the previous). Similarly, if you complete ex1 and ex2, you can submit copies of the same code ex1.c and ex2.c.
(Note: In this document, when we say "ex1-4.c", we refer to the four files: ex1.c, ex2.c, ex3.c, and ex4.c.)

Only changes made in **ex1-4.c** will be used for grading. You may make changes to the contents of **ex1-4.c** in any way if your code still compiles and works correctly. You may change the other files during your own testing, but we will ignore any changes you make to those files. **runner.c** might be replaced with a different version during testing, but the contracts specified in **api.h** will not be violated.

The expected behaviour of each of the functions you need to use is specified in **api.h**. The assignment can be completed without reading and understanding the code in **runner.c**. We provide a few samples of inputs and output along with **runner.c, api.h**, and some skeleton code for all exercises (in **ex.c**).

To compile the program, you may use gcc as follows:

```
gcc ex.c runner.c -o ex
```

If your installation of gcc does not automatically compile your code in multi-threaded mode, you can additionally pass `-pthread` to the compilation command. On Solaris platforms (such as Sunfire) semaphores are in librt, so you will need to pass `-lrt` on those platforms.
It might be helpful to ask gcc to produce warnings, which might indicate potential bugs in your code. You can pass `-Wall -Wextra` to enable all warnings. However, marks will not be deducted if your code produces warnings when it is compiled.

To run an exercise with `ex_sample.in` as input, use the following command:

```
./ex < ex_sample.in
```

Section 2. Simulating the Memory Management System

2.0 Overview of Memory Management System

There are four components in a typical virtual memory management system, namely:

- the user process,
- the memory management unit (MMU),
- the operating system (OS),
- the disk.

The user process is any process that runs on an operating system. The user process reads and writes to memory using virtual addresses and it has no knowledge of the physical address of the memory that it uses (i.e. where the memory is located in physical RAM).

The MMU is a component that translates virtual addresses into physical addresses using a page table. Every memory access requested by the user process is passed through the MMU to resolve the virtual address (specified by the user process) into the physical address. To determine the physical address from a virtual address, a page table is used. The page table is populated by the operating system, and it provides a mapping between virtual pages and physical frames. The MMU is typically implemented in hardware for efficiency. We ignore the effects of the translation lookaside buffer (TLB) and we assume that the TLB is automagically updated whenever necessary.

As the number of physical frames allocated to the user process is limited (and typically lower than the actual number of pages required), pages might have to be swapped between the **disk** and the RAM. A page is marked as 'invalid' in the page table when it is not in RAM and must be fetched from the disk. When the MMU finds that a required page is not in RAM, it pauses the user process, raises a page fault, and invokes the OS to load that page into RAM.

The OS finds the page on the disk, loads it into RAM, and resumes the user process once the page has been loaded. If there are no free frames in RAM, the OS picks a victim page to write to disk before loading the required page into RAM. If the request is for a page that does not exist even on the disk, the OS typically sends SIGSEGV (segmentation fault) to the user process, which will kill the user process by default (but it is possible for the user process to block SIGSEGV and take alternative actions instead). Once the OS resumes the user process, the MMU retries the page lookup and succeeds (assuming the OS does its job correctly).

Your Task

Your task for this assignment is to implement the functionality of the OS. The MMU and the OS are *simulated* on separate processes. The MMU has been written for you, and it communicates with your OS via POSIX signals. A simple API for writing to and reading from the disk is also implemented and provided to you (in **api.h**). You only need to implement function **os_run()** in **ex.c**.

Note: The provided API implementation (in **runner.c**) only *simulates* the page manipulation in RAM and disk (it does not really reserve pages in RAM or read/write to the disk). However, it should make no difference to your OS code if we rewrote our API implementation to really manipulate pages.

The simulation environment

The simulation environment provided in **runner.c** implements the MMU and keeps track of the contents of both the disk and the RAM. This simulation environment runs on a separate process, from which we send POSIX signals to your OS when a page fault occurs. OS reads and writes to the disk using the provided API calls provided in **api.h**. The API calls are forwarded to the simulation environment, and they allow for the

environment to perform some checks. During these checks, the simulation environment is able to detect some common errors (e.g. overwriting a dirty page without first writing to disk, writing the incorrect page to disk), and checks that the memory of the user process is not corrupted after a page fault is handled by the OS. (The memory of the user process is corrupted if there is a page that doesn't generate a page fault when the process attempts to read it, but the content of the associated frame in RAM is not what the process expects.)

2.0.1 API calls

OS reads from and writes to disk using the API calls provided in file **api.h**. You can make use of the following API calls:

- `void disk_create(int page_num)` - Prepare and initialize `page_num` on the disk.
- `void disk_read(int frame_num, int page_num)` - Asks the disk manager to read the page `page_num` from disk into frame `frame_num`.
- `void disk_write(int frame_num, int page_num)` - Asks the disk manager to write page `page_num` from frame `frame_num` to the disk. If the page does not already exist on disk, calling this command is a fatal error for your OS.
- `void disk_delete(int page_num)` - Delete page `page_num` from the disk.

2.0.2 `page_table` struct

We implemented and provided a `page_table` struct in **page_table.h**. This struct contains the page table that the MMU uses to translate virtual addresses into physical addresses. As the MMU is typically implemented in hardware, the layout of the page table (and page table entries) is determined by the hardware. **Hence you are not allowed to modify the layout of the page table (and page table entries).**

The `page_table` contains an array of `page_table_entry`. Each `page_table_entry` contains four fields:

- `frame_index`: the frame that this page is located at
- `valid`: 1 if this page is valid, 0 otherwise
- `referenced`: set to 1 by the MMU whenever this page is accessed
- `dirty`: set to 1 by the MMU whenever this page is written to

2.0.3 MMU behaviour

The simulated MMU in **runner.c** adheres to the following behaviour:

When the user process wants to read from some page X:

1. If `entries[X].valid` is 0, raise a page fault to your OS (further explained in the description for exercise 1)
2. Otherwise, set `entries[X].referenced` to 1 and read data from the frame located at `entries[X].frame_index`

When the user process wants to write from some page X:

1. If `entries[X].valid` is 0, raise a page fault to your OS (further explained in the description for exercise 1)
2. Otherwise, set `entries[X].referenced` to 1 and `entries[X].dirty` to 1 and write data to the frame located at `entries[X].frame_index`

In particular, this means that if `entries[X].valid` is 0, `entries[X].frame_index` is not used by the MMU. Furthermore, `entries[X].referenced` and `entries[X].dirty` are never read by the MMU. **You may use these assumptions when designing your OS page fault handler.**

When a page fault is raised to your OS, your OS should make the necessary amendments to the page table and then signal the MMU to continue. The check for memory corruption of the user process happens immediately after the MMU receives the signal to continue; failing the check results in an error message and in the immediate termination of the simulation environment. The MMU will also check that `entries[X].valid` is now 1. If both checks pass, the MMU will re-run the above algorithm on the instruction that produced the page fault, and then continue with subsequent instructions.

2.0.4 Second chance page replacement algorithm (CLOCK)

The algorithm is defined below to ensure that your OS places pages at identical frames as our implementation. During grading, the location of each page in RAM is used to check that your implementation of the algorithm is correct.

A circular queue of frames is maintained, starting with frame 0, then frame 1, and so on until the last frame. The queue then wraps back to frame 0 (hence its "circular" property). The algorithm also maintains a *next victim* index into this queue. The frame that the *next victim* points to is called the next victim frame. Initially, all frames do not contain valid pages, and the next victim is frame 0.

When a page fault occurs, the algorithm should choose the frame for the replacement page as follows:

1. If the next victim frame does not contain a valid page, or the page in the next victim frame does not have its 'reference' bit set, this frame is chosen. Then go to step 3.
2. Otherwise, clear the 'reference' bit, and set *next victim* to the next element in the circular queue (wrapping around if necessary), and go back to step 1.
3. Set *next victim* to the next element in the circular queue (wrapping around if necessary).

Remember that since the MMU will re-run the instruction that caused the page fault, the 'referenced' bit is set on the replacement page once the MMU is signalled to continue.

2.1 Exercise 1 [Lab Demo Exercise]

In this exercise, your OS needs to handle page faults for memory. Assume that the user program never writes to its memory (memory is read-only). Hence, the 'dirty' bit from the page table entry is never set by the MMU, and the OS does not need to write a page from RAM to the disk.

When a page fault occurs, you need to implement the second chance page replacement algorithm as described in Section 2.0.4 and the lecture notes. The MMU sets the 'referenced' bit in the relevant page table entry whenever the user process reads from (or writes to) a page.

You need to implement the function specified below in your ex1.c file:

```
void os_run(int initial_num_pages, page_table *pg_table)
```

This is the main function of your OS, and it should not return until the user program has terminated. `initial_num_pages` is the number of pages that the user program starts with; those pages are conveniently labelled from 0 to `(initial_num_pages - 1)`. `pg_table` is a pointer to the page table of the user process. The total number of physical frames in RAM allocated to your process is $2^{\text{FRAME\_BITS}}$ (i.e. `(1<<FRAME_BITS)` in C code), and the maximum number of pages that the user process is allowed to have at any time during its execution is $2^{\text{PAGE\_BITS}}$ (i.e. `(1<<PAGE_BITS)` in C code). FRAME_BITS and PAGE_BITS are compile-time constants specified in `page_table.h`. This means that the total number of frames and the maximum number of pages are guaranteed to be powers of two. We may modify the values of FRAME_BITS and PAGE_BITS during grading, but it is guaranteed that they will be integers between 1 and 20 inclusive. `initial_num_pages` is guaranteed to be between 1 and `(1<<PAGE_BITS)` inclusive.

The sample input and output for all exercises in this document assume that FRAME_BITS is 2 and PAGE_BITS is at least 5.

In your `os_run()` function, you should do the following:

1. For each page from `0` to `(initial_num_pages - 1)`, create the page on the disk. For simplicity, the disk stores every page of the user process, regardless of whether it is also in RAM. The pages on disk and pages in the virtual memory of the user process are identified using the same index.
2. Wait on SIGUSR1 using `sigwaitinfo()`. SIGUSR1 is sent by the MMU to the OS to indicate a page fault. The page required by the user process is stored in the integer value of the `sigval union` associated with the received signal (see the skeleton code in **ex.c** for details).
    a. If the page required is '-1', it is a sentinel value that indicates that the user process has exited. Your OS should do any necessary clean-up (e.g. free memory that you allocated) and then return from the `os_run()` function.
    b. Otherwise, a page fault has occurred. Your OS should load the required page from disk to a frame (as per the second chance page replacement

algorithm described above, using the `disk_read()` API call), modify the page table as necessary, then send SIGCONT back to the MMU (the pid of the MMU can be obtained from the `siginfo_t structure` populated by `sigwaitinfo()`). Then go back to step 2.

The SIGCONT signal should have the integer value of its `sigval union` set to '0'. This is because a different value indicates an error for later exercises.

The usage of signals and the `sigval union` is demonstrated in the code provided in **ex.c**. You may study it and modify it as necessary.

The **runner.c** provided to you will wait indefinitely for the SIGCONT signal from your OS. However, during grading, our runner will only wait for a "reasonable" amount of time (of at least ten milliseconds per page fault, excluding the time spent in the disk API functions), after which the runner will assume that your OS is not functioning properly.

Your task for Exercise 1:

Currently, the implementation of **`os_run()`** implements a trivial memory management algorithm that only uses a single frame.

Your task is to amend the implementation of **`os_run()`** to use the second chance page replacement algorithm, and adhere to the signalling protocol specified above. For this exercise, you may assume that pages are never dirty, so you do not need to write them back to disk. (No marks will be deducted if you write pages back to disk, as long as you do not corrupt the memory of the user process or the disk storage.)

Full marks are awarded if your program satisfies all the following:

1. It produces the correct sequence of calls to `disk_read()` (comparing both the frame and page indices) (see Section 2.0.1 for details on `disk_read`)
2. It does not corrupt the memory of the user process
3. It does not make any API call with invalid parameters or raise invalid signals to the MMU

Note that calls to `disk_write()` are not considered for item 1 as long as it satisfies items 2 and 3. Furthermore, the constraints above imply that you do not need to make all calls to `disk_create()` before any call to `disk_read()`. For example, you may instead only call `disk_create()` for page X at the first time the MMU requests for page X.

Note that it is not necessary to use multi-threading in any exercise of this lab assignment. However, if you do so, take note that the disk API functions are not thread-safe – you will have to ensure that successive calls to disk API functions are synchronised with each other.

Sample input (read by the simulation environment) – ex1_sample1.in

```
10
r 1
```

```
r 2
r 0
r 1
r 6
r 7
r 0
r 2
r 8
r 9
```

The first line of the input contains a single positive integer, specifying the number of pages the user process needs on initialisation. This is passed to **`os_run()`** via the `initial_num_pages` parameter. Each subsequent line contains a character 'r' followed by a page number, indicating that the user process wants to read from the specified page.

Possible output

```
disk_create(): Creating disk page 0... OK.
disk_create(): Creating disk page 1... OK.
disk_create(): Creating disk page 2... OK.
disk_create(): Creating disk page 3... OK.
disk_create(): Creating disk page 4... OK.
disk_create(): Creating disk page 5... OK.
disk_create(): Creating disk page 6... OK.
disk_create(): Creating disk page 7... OK.
disk_create(): Creating disk page 8... OK.
disk_create(): Creating disk page 9... OK.
disk_read(): Loading frame 0 from disk page 1... OK.
disk_read(): Loading frame 1 from disk page 2... OK.
disk_read(): Loading frame 2 from disk page 0... OK.
disk_read(): Loading frame 3 from disk page 6... OK.
disk_read(): Loading frame 0 from disk page 7... OK.
disk_read(): Loading frame 3 from disk page 8... OK.
disk_read(): Loading frame 1 from disk page 9... OK.
Runner: Child terminated normally with status 0.
```

Explanation of output

`disk_read` is called only when there is a page fault. Hence you will not see a `disk_read` for every read in the input file.

Note that the calls to disk_create() need not be sequenced before any call to disk_read(), as long as:

- For every page X, every call to disk_read() that loads page X must happen after some call to disk_create() that creates page X
- For every page X, there must be at most one call to disk_create() that creates page X

The constraints above allow the for lazy creation of pages on disk, which will be required in exercise 4.

Getting a message that ends with an exclamation mark, like in the example below, usually indicates a bug in your OS implementation or that an incorrectly formatted input command is supplied to the runner.

```
disk_read(): OS tried to load a page that is not on the disk!
```

2.2 Exercise 2

In this exercise, you will build on your code for exercise 1 to implement **writing pages from RAM to the disk**. When the user process writes to a page, the MMU sets the 'dirty' bit on the relevant page table entry. You should use the dirty bit to decide whether a page must be written to disk when it is evicted from RAM. To write a page from RAM to the disk, use the **`disk_write()`** function. Take note that the `page_num` parameter of the `disk_write()` function must be the actual page index of the frame.

You also need to check if the user process is attempting to access an unmapped page (i.e. a page that it does not have access to). For exercise 2, this means the OS should give a segmentation fault if the user process is trying to access a page that is not in the range `0` to `(initial_num_pages - 1).` When this happens, the OS should raise the SIGCONT signal to the MMU like before, but the integer value of its `sigval union` should be set to '`1`'. Note that the user process may not necessarily be killed by the segmentation fault – it may decide to recover from the error instead (see sample input 2 for details).

Your task for Exercise 2:

You should build on your solution for exercise 1, implementing the ability (in the OS) to write evicted pages to disk, and to detect a segmentation fault of the user process.

Full marks will be given if your program satisfies all the following:

1. All requirements from exercise 1.
2. It produces the correct sequence of `disk_write()` and `disk_read()`, according to the second chance page replacement algorithm described above.
3. If the user process attempts to access a page that it does not have access to, the SIGCONT signal is raised with the proper sigval as described above.

Sample input 1 (read by the simulation environment) – ex2_sample1.in

```
10
r 0
r 1
r 2
r 3
```

```
w 0
w 2
w 4
r 1
r 5
r 0
w 3
r 1
r 2
w 4
r 3
```

Each line that contains a character 'w' followed by a page number, indicates that the user process wants to write to the specified page.

Possible output 1

```
disk_create(): Creating disk page 0... OK.
disk_create(): Creating disk page 1... OK.
disk_create(): Creating disk page 2... OK.
disk_create(): Creating disk page 3... OK.
disk_create(): Creating disk page 4... OK.
disk_create(): Creating disk page 5... OK.
disk_create(): Creating disk page 6... OK.
disk_create(): Creating disk page 7... OK.
disk_create(): Creating disk page 8... OK.
disk_create(): Creating disk page 9... OK.
disk_read(): Loading frame 0 from disk page 0... OK.
disk_read(): Loading frame 1 from disk page 1... OK.
disk_read(): Loading frame 2 from disk page 2... OK.
disk_read(): Loading frame 3 from disk page 3... OK.
disk_write(): Writing frame 0 to disk page 0... OK.
disk_read(): Loading frame 0 from disk page 4... OK.
disk_write(): Writing frame 2 to disk page 2... OK.
disk_read(): Loading frame 2 from disk page 5... OK.
disk_read(): Loading frame 3 from disk page 0... OK.
disk_read(): Loading frame 1 from disk page 3... OK.
disk_write(): Writing frame 0 to disk page 4... OK.
disk_read(): Loading frame 0 from disk page 1... OK.
disk_read(): Loading frame 2 from disk page 2... OK.
disk_read(): Loading frame 3 from disk page 4... OK.
Runner: Child terminated normally with status 0.
```

The runner might produce a warning message similar to the following one:

```
disk_write(): Warning, OS is writing non-dirty page 3 to disk.
```

The above message means that you are writing a page that is not actually dirty to disk, so this write was redundant. It is not a hard error because the user process is still able

to function, but it reduces the efficiency of your OS. Marks will be deducted if this message is produced.

Sample input/output 2 shows how segmentation fault is handled:

Sample input 2 (read by the simulation environment) – ex2_sample2.in

```
8
r 0
r 1
r 2
r 3
r * 8
r * 9
w * 10
r 4
```

The '*' (asterisk) character is included when reading and writing pages that are not mapped to the user process. These instructions are expected to cause a segmentation fault for the user process. While the input format would already have been semantically unambiguous without the use of the asterisk, mandating the asterisk for unmapped pages makes it easier to catch errors in the input file format. The runner checks that the page is indeed not mapped to the user process.

Possible output 2

```
disk_create(): Creating disk page 0... OK.
disk_create(): Creating disk page 1... OK.
disk_create(): Creating disk page 2... OK.
disk_create(): Creating disk page 3... OK.
disk_create(): Creating disk page 4... OK.
disk_create(): Creating disk page 5... OK.
disk_create(): Creating disk page 6... OK.
disk_create(): Creating disk page 7... OK.
disk_read(): Loading frame 0 from disk page 0... OK.
disk_read(): Loading frame 1 from disk page 1... OK.
disk_read(): Loading frame 2 from disk page 2... OK.
disk_read(): Loading frame 3 from disk page 3... OK.
Runner: OS detected user process segfault... OK.
Runner: OS detected user process segfault... OK.
Runner: OS detected user process segfault... OK.
disk_read(): Loading frame 0 from disk page 4... OK.
Runner: Child terminated normally with status 0.
```

Sample output 2 shows that OS detects a segmentation fault for each of the reads and writes on pages 8, 9, and 10. This is because only pages 0-7 are mapped and can be accessed.

2.3 Exercise 3

In this exercise, you will build on your code for exercise 2 to allow the user process to `mmap/munmap` memory. For simplicity, every `mmap` called by the user process asks for

memory that is exactly one page size, and the user process does not demand that the memory is mapped to a particular virtual address. Specifically, your OS code should determine a free page to map the new memory region. If the user process tries to `munmap` un-allocated memory, then `munmap` should be ignored (no operation is done by the OS).

The OS knows that the user process is calling `mmap/munmap` when SIGUSR2 is sent by the MMU:

- For `mmap`, the integer value of the `sigval union` associated with the received signal is '-1'.
- For `munmap`, the integer value of the `sigval union` associated with the received signal is the page number that needs to be unmapped.

You need to modify the signal handling on your own to listen to SIGUSR2.

To `mmap` new memory, the OS does the following steps:

1. Find a page that is not mapped. Before any page is mapped using `mmap`, the pages available for mapping are from `initial_num_pages` to `(1<<PAGE_BITS - 1)` inclusive, i.e. all pages that are not already used by the user process. It is guaranteed that there are pages available for mapping when a `mmap` call is done. The OS may choose any available page for mapping.
2. Create the new page on disk (map on disk), by using the `disk_create` API call.
3. Raise SIGCONT signal to MMU with an integer value of its `sigval union` set to the new mapped page.

To `munmap` memory, the OS does the following steps:

1. Determine the page that needs to be unmapped by reading the integer value of the `sigval union`.
2. If currently mapped to a frame, the page should be marked as invalid.
3. Delete the page from disk using `disk_delete` API call.
4. Raise SIGCONT signal to MMU (integer value of its `sigval union` is not important).

Note that the user process is allowed to `munmap` from the initial set of pages created at the start of execution.

Similar to ex2, OS should give a segmentation fault if the user process is trying to access unmapped memory. But you need to check for both initially allocated pages and mapped pages using `mmap`.

Your task for Exercise 3:

Full marks will be given if your program satisfies all the following:

1. All requirements from exercise 2.
2. It waits for `mmap/munmap` by listening to SIGUSR2.
3. It executes `mmap` requests properly by choosing an available page, which can then be used by read/write operations.

4. It executes `munmap` requests properly, and subsequent read/write operations to the unmapped page raise a segmentation fault correctly.
    - If the user process attempts to `munmap` a page that was not mapped, no operation is performed and execution of the user process resumes.

Remember that in all situations you have to raise SIGCONT signal to MMU.

Sample input (read by the simulation environment) – ex3_sample1.in

```
5
r 2
r 1
r 0
m 10
r 10
r 3
w 10
w 1
w 0
r 4
r 2
r 1
u 10
m 11
r 0
w 11
u 11
u * 16
r 2
r 3
```

Each line that contains a character 'm'/'u' followed by an **identifier**, indicates that the user process wants respectively to mmap/munmap the specified memory identifier. This identifier is solely for the purposes of parsing the input file, and is never sent to the OS. Note that the identifier is NOT equivalent to the page number that should be mapped/unmapped. In the sample input above, the use of '10' as an identifier means that the page mapped at "m 10" is the same page read at "r 10", the same page written to at "w 10", and the same page unmapped at "u 10". The OS has the freedom to decide where to `mmap` the new page to – in the output below, the OS decided to map the new page at index 5. The runner internally maintains a mapping between the identifiers and the page numbers in the OS.

The '*' (asterisk) character in an munmap instruction indicates that the page/identifier is not mapped to the user process. When this is encountered, the munmap operation is expected to make no visible changes to the mapped pages of the user process. Note that as the OS has the freedom to decide where to `mmap` the new page to, it may be possible that the identifier in the input file is already a mapped page. In cases where the identifier is currently a mapped page or was a mapped page at some point in time, the runner may use a different page index when communicating with the OS, so that the page index sent to the OS is guaranteed to not currently be mapped.

Possible output

```
disk_create(): Creating disk page 0... OK.
disk_create(): Creating disk page 1... OK.
disk_create(): Creating disk page 2... OK.
disk_create(): Creating disk page 3... OK.
disk_create(): Creating disk page 4... OK.
disk_read(): Loading frame 0 from disk page 2... OK.
disk_read(): Loading frame 1 from disk page 1... OK.
disk_read(): Loading frame 2 from disk page 0... OK.
disk_create(): Creating disk page 5... OK.
Runner: mmap operation OK, OS mapped page 5.
disk_read(): Loading frame 3 from disk page 5... OK.
disk_read(): Loading frame 0 from disk page 3... OK.
disk_write(): Writing frame 1 to disk page 1... OK.
disk_read(): Loading frame 1 from disk page 4... OK.
disk_write(): Writing frame 2 to disk page 0... OK.
disk_read(): Loading frame 2 from disk page 2... OK.
disk_write(): Writing frame 3 to disk page 5... OK.
disk_read(): Loading frame 3 from disk page 1... OK.
disk_delete(): Deleting disk page 5... OK.
Runner: munmap operation OK, OS unmapped page 5.
disk_create(): Creating disk page 5... OK.
Runner: mmap operation OK, OS mapped page 5.
disk_read(): Loading frame 0 from disk page 0... OK.
disk_read(): Loading frame 1 from disk page 5... OK.
disk_delete(): Deleting disk page 5... OK.
Runner: munmap operation OK, OS unmapped page 5.
Runner: munmap operation OK, OS did not unmap any page because
page 16 was already unmapped.
disk_read(): Loading frame 3 from disk page 3... OK.
Runner: Child terminated normally with status 0.
```

In the output above, the OS decided to allocate page 5 as a result of the `mmap` request. As the OS is allowed to choose any available page, the page chosen by your OS may differ.

2.4 Exercise 4

In this exercise, you will implement **commit-on-write behaviour** for memory, in addition to all requirements for ex3. Specifically, a page should be created on disk only when the first read or write is done on it (i.e. lazy creation of pages on disk). The output for this exercise should show `disk_create()` operations taking place just before the first `disk_read()` or `disk_write()` on that page (instead of in the beginning of the execution or during an `mmap` operation).

For the input in exercise 3, this is a possible output:

Possible output

```
disk_create(): Creating disk page 2... OK.
disk_read(): Loading frame 0 from disk page 2... OK.
disk_create(): Creating disk page 1... OK.
```

```
disk_read(): Loading frame 1 from disk page 1... OK.
disk_create(): Creating disk page 0... OK.
disk_read(): Loading frame 2 from disk page 0... OK.
Runner: mmap operation OK, OS mapped page 5.
disk_create(): Creating disk page 5... OK.
disk_read(): Loading frame 3 from disk page 5... OK.
disk_create(): Creating disk page 3... OK.
disk_read(): Loading frame 0 from disk page 3... OK.
disk_write(): Writing frame 1 to disk page 1... OK.
disk_create(): Creating disk page 4... OK.
disk_read(): Loading frame 1 from disk page 4... OK.
disk_write(): Writing frame 2 to disk page 0... OK.
disk_read(): Loading frame 2 from disk page 2... OK.
disk_write(): Writing frame 3 to disk page 5... OK.
disk_read(): Loading frame 3 from disk page 1... OK.
disk_delete(): Deleting disk page 5... OK.
Runner: munmap operation OK, OS unmapped page 5.
Runner: mmap operation OK, OS mapped page 5.
disk_read(): Loading frame 0 from disk page 0... OK.
disk_create(): Creating disk page 5... OK.
disk_read(): Loading frame 1 from disk page 5... OK.
disk_delete(): Deleting disk page 5... OK.
Runner: munmap operation OK, OS unmapped page 5.
Runner: munmap operation OK, OS did not unmap any page because
page 16 was already unmapped.
disk_read(): Loading frame 3 from disk page 3... OK.
Runner: Child terminated normally with status 0.
```

In the output above, each page is only created on disk just before the first read or write to that page.

Section 3. Submission

Zip the following files as E0123456.zip (**use your NUSNET id, NOT your student no A012…B, and use capital 'E' as prefix):**

a. **`ex1.c`**
b. **`ex2.c`**
c. **`ex3.c`**
d. **`ex4.c`**

Do **not** add additional folder structure during zipping, e.g. do not place the above in a "lab4\" subfolder etc.

Upload the zip file to the "Lab Assignment 4" folder on LumiNUS. Note the deadline for the submission is **Sat, 2 November, 2pm**.

Please ensure you follow the instructions carefully (output format, how to zip the files etc). **Deviations will be penalized.**