Programming Assignment 4 (Memory) [**25 pts + 10 bonus**]

COMP 4270: Operating Systems

Fall 2018

Due: 12/10/2018

No late submission will be accepted for this assignment.

Objectives

By completing this programming assignment, you will be familiarized with the paging-based memory management scheme. You will learn how to implement memory allocation, memory deallocation, and memory access. **For this assignment, you are free to use ANY programming language of your choice**.

Overview

In this assignment, you will write a program that simulates the paging-based memory management scheme. More specifically, you will write (1) a function that creates a simulated physical memory space, (2) a function that allocates a chunk of memory for a process, (3) a function that deallocates memory from a process, (4) a function that writes a value to a memory location of a process, (5) a function that reads a value from a memory location of a process, (6) a function that prints out memory contents.

Task 1 [10 base pts]

Write a function:

void memoryManager(int memory_size, int frame_size)

This function is used to create a simulated physical memory space. The physical memory is represented as an array of bytes (*e.g.*, unsigned char in C/C++). Thus, creating physical memory space is basically creating an array of bytes. This function takes two integer parameters *memory_size* and *frame_size*. The former is the size of physical memory in bytes, and the latter is the size of a frame in bytes. Recall that **physical memory** is divided into blocks of the same size called **frames**.

After creating physical memory (array of bytes), **initialize** the value of each byte to **0**. For example, a function call *memoryManager(100, 20)* will create a 100 bytes physical memory space which is divided into a number of frames. More specifically, since the frame size is 20 bytes, 5 frames will be created, say, frame #1, frame #2, …, frame #5 (100/20 = 5).

In this function, a **free frame list** should also be created. Recall that the free frame list is simply a vector consisting of the frame numbers of free (available) frames; it is used to keep track of free frames in memory. For example, if a function *memoryManager(100,20)* is invoked, 5 frames are created. Since these 5 frames are not mapped to **pages**, they are all free frames. The free frame list in this example consists of the frame numbers of these 5 free frames, *i.e.*, 1, 2, 3, 4, and 5.

Testing this function:

You program should take user input in the following format:

<command><space><parameter 1><space><parameter 2> …

In particular, command '**M**' is used to create a simulated physical memory space. The command '**M**' takes two parameters. The first parameter is the size of physical memory to be created, and the second parameter is the frame size. For example,

```
mwon>./prog4

Input: M 100 20

100 Bytes physical memory (5 frames) has been created.

Input: P (See Task 6)

f1: 00000000000000000000

f1: 00000000000000000000

f3: 00000000000000000000

f4: 00000000000000000000

f5: 00000000000000000000
```

Here command '**P**' is used to print out the contents of physical memory (See Task 6 for more details).

Task 2 [5 pts]

Write a function:

int allocate(int allocate_size, int pid)

This function is used to allocate a chunk of memory for a process. This function takes two integer parameters *allocate_size* and *pid*. The former is the size of allocated memory in bytes. The pid is the id of a process for which memory is allocated. The following lists what you should implement in this function:

(1) Find how many **pages** should be allocated for the process with pid. For example, let's assume that the **frame size** (page size) is 20 bytes. Now if *allocate_size=30*, then we need to allocate two pages because we need at least two pages (total of 40 bytes) to cover the requested 30 bytes of memory.

(2) Refer to the **free frame list** and find if there is enough free space (*i.e.*, "enough" number of free frames) to load requested pages. If there is not enough space, print out "*Not Enough Memory*", and return *-1*.

(3) If there is enough free space (*i.e.,* # of free frames >= # of requested pages), requested pages are then mapped to free frames. If you do not understand what this 'mapping' means here, please review the course slides and come back. Use the "sequential mapping" scheme – more precisely, in this scheme, the first page is mapped to the first available free frame, and then the next page is mapped to the next available free frame, and so on.

(4) Create a **page table** if it does not exist (Note that if we have allocated memory space to a process, a page table for the process would already exist). The page table specifies **mapping** between the pages and frames. More specifically, implement the page table using a vector in which each element represents the frame number. For example, if the first element of the vector is 3, this means that page number #1 is mapped to frame #3. If the third element of the vector is 4, this means that page number #3 is mapped to frame #4, *etc.* Note that you should create a page table for each process, *i.e.*, each process has its own page table. Also note that for simplicity, let's assume that page tables are not stored in the simulated physical memory space.

(5) Modify the **free frame list** if requested pages have been successfully mapped to free frames, since these mapped frames are no longer free frames.

(6) Return 1 to indicate successful memory allocation.

Testing this function: Use command '**A**' to allocate a chunk of memory for a process. The first parameter of this command is the size of allocated memory. The second parameter is the pid. For example,

```
mwon>./prog4

Input: M 100 20

100 Bytes physical memory (5 frames) has been created.

Input: A 20 1

20 Bytes memory has been allocated for process 1.

Input: P (See Task 6)

f1->p1 (proc1): 00000000000000000000

f1: 00000000000000000000

f3: 00000000000000000000

f4: 00000000000000000000

f5: 00000000000000000000
```

Task 3 [5 pts]

Write a function:

int deallocate(int pid)

This function is used to deallocate memory from a process with pid. The following lists what you should implement in this function.

(1) Remove the page table of the process with pid.
(2) Modify the **free frame list**.

Testing this function:

Use command '**D**' to deallocate memory from a process with pid. The first parameter of this command is the pid. For example,

```
mwon>./prog4

Input: M 100 20

100 Bytes physical memory (5 frames) has been created.

Input: A 20 1

20 Bytes memory has been allocated for process 1.

Input: D 1

20 Bytes memory has been deallocated from process 1.

Input: P (See Task 6)

f1: 00000000000000000000

f1: 00000000000000000000

f3: 00000000000000000000

f4: 00000000000000000000

f5: 00000000000000000000
```

Task 4 [5 pts]

Write a function:

int write(int page_number, int offset, int pid)

This function is used to write a value of '1' to a memory location of a process with pid. The memory location is specified by a **logical address** consisting of a **page number** and an **offset** (Review the course slides on paging if you do not understand this). It is important to note that this logical address must be converted to a physical address in order to access the simulated physical memory space. For this conversion, use the **page table** of a process with pid. This function returns 1 if a value of '1' has been written to the memory location. If not, this function returns -1.

Print out '**Illegal memory access!**' if the process with the pid does not exist, or if it attempts to write to memory location of other process.

Testing this function:

Use command '**W**' to write a value of '1' to a memory location of a process with pid. The first parameter of this command is the page number, the second is the offset, and the third is the pid. For example,

```
mwon>./prog4

Input: M 100 20

100 Bytes physical memory (5 frames) has been created.

Input: A 20 1

20 Bytes memory has been allocated for process 1.

Input: W 1 20 1

Input: P (See Task 6)

f1->p1 (process1): 00000000000000000001

f1: 00000000000000000000

f3: 00000000000000000000

f4: 00000000000000000000

f5: 00000000000000000000
```

Task 5 [5 pts]

Write a function:

int read(int page_number, int offset, int pid)

This function is used to read a byte from a memory location for a process with pid (third parameter). The memory location is specified by a logical address consisting of a page number (first parameter), offset (second parameter). Similar to the write() function, address conversion from logical address to physical address must be done to access the simulated physical memory space. This function should return the read value. Print out 'Illegal memory access!' if the process with the pid does not exist, or if it attempts to read from memory location of other process.

Testing this function: Use command '**R**' to read a byte from a process with pid. The first parameter of this command is the page number, the second is the offset, and the third is the pid. For example,

```
mwon>./prog4

Input: M 100 20

100 Bytes physical memory (5 frames) has been created.

Input: A 20 1

20 Bytes memory has been allocated for process 1.

Input: W 1 20 1

Input: P

f1->p1 (process1): 00000000000000000001

f1: 00000000000000000000

f3: 00000000000000000000

f4: 00000000000000000000

f5: 00000000000000000000

Input: R 1 20 1

1

Input: R 1 19 1

0

Input R 1 21 1

Illegal memory access!
```

Task 6 [5 pts]

Write a function:

void printMemory(void)

This function is used to print out the memory contents of physical memory. Each line of the output should display the memory contents of a frame. Each line starts with a frame number (*e.g.,* f1). If a frame is mapped to a page, each line should start with the frame number of the frame, the page number of the page, and the pid in the following format: f#->p# (process#). For example, f1->p1 (process1) indicates that frame #1 is mapped to page #1 of process #1 (pid).

For example,

Use command '**P**' to print out the memory contents frame by frame.

```
Input: P

f1->p1 (process1): 00000000000000000000

f1->p1 (process2): 00000000000000000000

f3->p2 (process1): 00000000000000000000

f4: 00000000000000000000

f5: 00000000000000000000
```

Evaluation criteria

- Your assignment will be evaluated based on the following:

  Documentation 10% - your code should be easy to read and well commented. For each function used in your program, the use of function, its parameters, and return values should be well described.

  Compilation 20% - your program should compile with no errors and/or warnings (base points)

  Correctness 70% - To grade your work, we will run your program with some test cases. You will get full credits for correctness if your program prints out correct output for our input test cases.

References

[1] Operating Systems Concepts 9th Edition by Silberscharz, Yale, and Gagne, Wiley