CSCI4730/6730 – Operating Systems
Project #3

Due date: 11:59pm, 03/31/2022

Description

In this project, you will implement a virtual memory simulator in order to understand the behavior of a page table and the page replacement algorithms. Virtual memory allows the execution of processes that are not completely in memory. This is achieved by page table and page replacement techniques. A page table stores the mapping information between virtual addresses and physical addresses. A page fault mechanism by which the memory management unit (MMU) can ask the operating system (OS) to bring in a page from the swap disk. The system we are simulating is 16-bit machine (8 bits for index and 8 bits for offset).

Part 1: Page Table – (50%)

The page table structure that we will implement is a linear page table. Each page table entry contains a physical frame number (PFN), a valid bit, and a dirty bit. The size of main memory (# of physical memory frames) is configurable via command-line arguments. The size of virtual memory (# of virtual pages), and the number of process are specified in the input file. The machine we are simulating supports up to 256 pages, 256 frames, and 8 processes.

Initially, all physical frames are free. Our page allocation policy simply hands out the free physical frame until the free-frame list becomes empty. When the free-frame list is empty, you should find a victim page using page replacement policy.

We also simulate the placement of virtual pages on the swap disk. We keep track of the number of swap disk read and write operations that are needed to handle the page faults. We maintain “dirty” bit on the page table and swap out the dirty page when it is replaced.

In this project, you will need to handle two types of requests. The request format is the following:

*[**pid**] [**R** or **W**] [**Virtual Address**] [**Byte** (if W)]*

1. **Read req**: It reads 1 byte of the memory.
2. **Write req**: It writes 1 byte to the memory. For simplicity, we use a single character to fill-out the entire page. For example, if the request is "1 W 0x03D2 H", it will fill out the entire page from 0x0300 to 0x03FF (256 bytes) with "H". We assume that the entire process memory is initialized with "A".

In this project, you **do not** need to directly access data structure for “page table”, “inverted page table”, or “swap disk”. Instead, you can simply use APIs to access them. All APIs are defined in “API.h” header file.

1. To access the page table (convert virtual page number (VPN) to physical frame number (PFN))
   A. **PTE read_PTE (int pid, int VPN);**
   B. **void write_PTE(int pid, int VPN, PTE pte);**

2. To access the inverted page table (convert physical frame number (PFN) to virtual page number (VPN) and process id (pid)):
   A. **IPTE read_IPTE (int PFN);**
   B. **void write_IPTE (int PFN, IPTE ipte);**

3. To access the swap disk:
   A. **void swap_in(int pid, int VPN, int PFN);** // read swap disk: swap space -> physical memory
   B. **void swap_out(int pid, int VPN, int PFN);** // write swap disk: physical memory -> swap space

The following is details of data structure:

- Data structure for the page table is defined in vm.h
  - PTE is a page table entry. It contains virtual to physical mapping information for each virtual pages including [physical frame number], [valid bit], and [dirty bit].

    ```
    typedef struct {
      int PFN;
      bool valid;
      bool dirty;
    } PTE;
    ```

  - Each process has own page table.

- Invert page table is necessary to identify process id and virtual page number for each physical frame. You need this information during swapping procedure.

  ```
  typedef struct {
    int pid;
    int VPN;
  } IPTE;
  ```

Implementation guidelines:

1. You will need to implement the following two functions in “pagetable.c” file.
    - A. **int is_page_hit(int pid, int VPN, char type)**
        - i. This function checks the page table. If the requested page is available and valid, return PFN. If the page is not valid, return -1.
    - B. **int pagefault_handler(int pid, int VPN, char type)**
        - i. This function handles the page fault. If a free frame is available, simply use it to load a request page to physical memory. Note that, “get_freeframe()” function to get free frame is provided and you don’t need to implement it. If no free frame is available, call “find_replacement” function to find a victim frame from the physical memory.
        - ii. To read a page from swap disk, use “swap_in” function. To write a page from the memory to swap disk (when it is dirty), use “swap_out” function.
2. Currently, ZERO replacement algorithm is available that always replaces the frame 0 for the page fault. You will be implementing FIFO, LRU, and CLOCK in the next part.

Part 2: Page Replacement Algorithm– (50%, FIFO: 15%, LRU: 15%, CLOCK: 20%)

The current simulator only supports ZERO page replacement algorithm that always replace the frame 0.

In this part, you will implement three page replacement algorithms, First-in-first-out (FIFO) and Least-recently-used (LRU) algorithms, and second chance algorithm (CLOCK).

- All replacement functions are declared in pagetable.c file, but they are empty now.
- The simulator requires a command-line argument to specify a replacement policy.
    - 0: ZERO
    - 1 : FIFO
    - 2 : LRU
    - 3 : CLOCK
- You can use “double linked list” to maintain LRU list. “list.c” and “list.h” are double linked list implementation. If you want, feel free to use your own or favorite linked list implementations.
    - Define head node with “struct Node”. This should be initialized as NULL.
    - “list_insert_head()” to insert a new item into the head of the list.
    - “list_insert_tail()” to insert an item into the tail of the list.
    - “list_remove()” to remove an item.
    - “list_print()” to print the list.
    - list_test.c is a test implementation for the list.

Command-Line Arguments

Our virtual memory simulator requires three command line arguments.

1. Number of frames in the physical memory (between 1 and 256)
2. Page replacement policy (0 - ZERO, 1 - FIFO, 2 - LRU, 3 – CLOCK)
3. Input file that contains
    A. The number of processes and virtual pages
    B. reference string
4. ex) ./vm 4 0 ./example_inputs/belady_input.txt

Input Generator

The input format for your simulator will be in the following format:

*[**pid**] [**R** or **W**] [**Virtual Address**] [**Byte** (if W)]*

You can find example inputs, including the belady sequence and reference string, in "example_input" folder.

The first line of the input file indicates "# of pages" and "# of processes"

You may use a provided input generation tool "input_gen" to generate random inputs.

- $./input_gen [# of page] [# of process] [# of requests]

Output Format

The output of your simulation should be the result of each memory request: "[PID, R/W] original_request --> physical_address: byte [hit/miss]".

For the process termination, the output should be:

"[PID, T] # of freed physical frame"

At the end of a request sequence, the total number of requests, the total numbers of hit and miss in page table, and the total numbers of swap disk read and write should be printed. A code for the output is already implemented, and you will just need to use provided APIs to update "hit", "miss", "swap read", and "swap write" count. You do not need any additional implementation for that.

Example of output (belady input with 6 virtual pages, 3 physical frames, FIFO):

```
$ ./vm 3 1 ./example_inputs/belady_input.txt

# PAGES: 6, # FRAMES: 3, # PROCS: 1, Replacement Policy: 1 - FIFO

[pid 0, W] 0x0100 --> 0x0000: a [miss]

[pid 0, W] 0x0200 --> 0x0100: b [miss]

[pid 0, W] 0x0300 --> 0x0200: c [miss]

[pid 0, W] 0x0400 --> 0x0000: d [miss]

[pid 0, R] 0x0100 --> 0x0100: a [miss]

[pid 0, R] 0x0200 --> 0x0200: b [miss]

[pid 0, W] 0x0500 --> 0x0000: e [miss]

[pid 0, R] 0x0100 --> 0x0100: a [hit]

[pid 0, R] 0x0200 --> 0x0200: b [hit]

[pid 0, R] 0x0300 --> 0x0100: c [miss]

[pid 0, R] 0x0400 --> 0x0200: d [miss]

[pid 0, R] 0x0500 --> 0x0000: e [hit]

=====================================

Request: 12

Page Hit: 3 (25.00%)

Page Miss: 9 (75.00%)

Swap Read: 9

Swap Write: 4
```

Important! You should remove all debugging messages before the submission. TA will use "diff" to compare your output with the solution output. You could have a penalty if your code emits anything other than the defined output.

Tip! You can use "debug(..)" instead of "printf(..)" to print out your debugging messages. It will print out the messages in the screen if you comment out "#define NDEBUG" in vm.h file. You can simply enable "#define NDEBUG" before the submission to remove the debugging messages.

Testing

You may use “vm_reference” executable to generate reference output and compare it with yours. Note that it is not yet fully tested and could have bugs and can crash.

$./vm_reference 3 1 ./example_inputs/belady_input.txt

Submission

Submit a tarball file to include all source code and a README file:

1. README file with:
    a. Your name
    b. List what you have done and how you tested them. So that you can be sure to receive credit for the parts you’ve done.
    c. Explain your design of data structures.
    d. README should be **PDF format. Only PDF will get points.**
2. **Your code should be compiled in odin machine. Any compilation error on odin will lead to 0 point.** Make sure to use Makefile and command "make" to compile the project on Odin. If you have any special compilation needs, make sure to modify Makefile accordingly.
3. Submit a tarball through eLC.