CS 3013 - Operating Systems

Project 4 (100 points)
Assigned: Thursday, February 15, 2018
Checkpoint: Tuesday, February 20, 2018
Due: Friday, February 23, 2018

Project 4: Implementing Virtual Memory

Virtual memory is a powerful construction that allows programs to believe they have access to a larger amount of memory resources than is present in the physical RAM on a computer. In this project you will simulate a virtual memory system with paging and swapping.

Part 1

Your objective is to create a **memory manager** that takes instructions from simulated processes and modifies physical memory accordingly. In part 1 of this project, you must implement the basics of paging, including a per-process page table, address translation, and support for multiple processes residing in memory concurrently.

Memory Implementation. To keep the implementation simple, you will simulate **memory** as an array of bytes, e.g., `unsigned char memory[SIZE])`. It is the responsibility of your memory manager to logically partition those bytes into pages, perform translations from virtual to physical addresses, load and store values in memory, and update the page tables. *Hint:* given our simple representation of memory, a physical address is simply an index into the `memory` array.

Page Table Implementation. You are free to use whatever format you would like for the page table and its entries (*hint:* the OSTEP book has plenty of good examples). However, **each process's page table must be stored in the `memory` array.** That is, each page table will require a page of physical memory. Further, we've set the virtual address space size such that the page table should require at most a single page of memory.

Input. Instructions will be supplied to your memory manager via `stdin` and will always a 4-tuple in the format described below:

```
process_id,instruction_type,virtual_address,value
```

The `process_id` is an integer value in the range $[0, 3]$ used to simulate instructions from different processes. The `instruction_type` specifies the desired memory operation—see the instruction list below. The `virtual_address` is an integer value in the range $[0, 63]$ specifying the virtual memory location for given process. The meaning of `value` depends on the instruction, but must be an integer value in the range $[0, 255]$.

Instructions. Your memory manager must support the following instructions:

- `map` tells the memory manager to allocate a physical page, i.e., it creates a mapping in the page table between a virtual and physical address. The manager must determine the appropriate virtual page number using the `virtual_address`. For example, `virtual_address` of 3 corresponds to virtual page 0. `value` argument represents the write permission for the page. If `value=1` then the page is writeable and readable. If `value=0`, then the page is only readable, i.e., all mapped pages are readable. These permissions can be modified by using a second `map` instruction for the target page.

- `store` instructs the memory manager to write the supplied `value` into the physical memory location associated with the provided `virtual_address`, performing translation and page swapping as necessary. Note, page swapping is a requirement for part 2 only.
- `load` instructs the memory manager to return the byte stored at the memory location specified by `virtual_address`. Like the `store` instruction, it is the memory manager's responsibility to translate and swap pages as needed. Note, the `value` parameter is not used for this instruction, but a dummy value (e.g., 0) should always be provided.

Output. The memory manager should be output the results of its actions by printing messages to `stdout`. See below for examples of printed output. If the process attempts an illegal instruction (e.g., writing to a read-only page) or provides invalid arguments, the manager should print a warning and ignore the instruction.

Simulation parameters. You may make the following simplifying assumptions.

- Physical memory consists of 64 bytes divided into pages of 16 bytes. Likewise, the virtual address space is also 64 bytes.
- For the purposes of this simulation, a process is simply a PID provided in the input command. Each process is given its own, isolated, virtual address space.
- Your memory manager should be single-threaded.
- All pages are readable, but only some are writeable—as specified by the `value` argument for the `map` instruction. We do not consider other protection bits for this project (e.g., executable).
- The there can be at most 4 different processes issuing instructions. These processes will have PIDs in the range $[0, 3]$.
- The manager will create page tables *on-demand*. That is, the manager will create the page table for a process when it receives the first command for that process. Each page table will be placed in its own page in the simulated `memory` array.
- The manager will simply print an error if it cannot allocate a physical page to satisfy a request. In part 2 of the project, you will relax this assumption using swapping.
- Each process will have a simulated hardware register pointing to the start of their respective page tables. You can simulate these registers with an array indexed by `process_id`. This array does not need to be stored in your simulated physical memory array.
- You may store limited information outside of your simulated physical memory array, e.g., local variables, a free list, and the page table registers. Yes, technically, these objects could also be put in the simulated memory, but it would complicate the implementation. However, as mentioned above, **all page tables must be placed in the simulated physical memory as well as all `loads` and `stores` from the simulated processes**.

Example output. An example run of the memory manager might look like the following. Note, `Instruction?` is the prompt printed by the manager.

```
Instruction? 0,map,0,1
Put page table for PID 0 into physical frame 0
Mapped virtual address 0 (page 0) into physical frame 1
Instruction? 0,store,12,24
Stored value 24 at virtual address 12 (physical address 28)
Instruction? 0,load,12,0
The value 24 is virtual address 12  (physical address 28)
```

Part 2

In part 2, you will extend your memory manager to support swapping memory pages to disk. The manager must ensure that the address spaces remain isolated. That is, memory from one process should not be readable from another process. The manager is expected to swap out page tables as necessary to free up memory. Hint: things get interesting when you swap out page table frames, especially if it is the page table for the process making the request.

Specifically, when the manager cannot satisfy a request because it does not have a free page it will do the following.

- First, the manager will pick a page to evict. You get to pick the eviction strategy, but a simple round-robin should be sufficient. Note, we do not consider "always evict page 1" to be an acceptable eviction strategy. The manager must be able to swap out all pages, even those containing a page table.
- Second, the manager will write the evicted page to disk. The swap space will be modeled by reading from and writing to a file. You can assume your swap space is very large.
- Third, the manager will update the appropriate page table to record the swap-file location for the evicted page. Hint: you need a way to specify in the page table whether the page is in memory or swapped to disk.
- Finally, the manager will use the newly-freed page to satisfy the request.

Example output. An example run of the memory manager might look like the following. Note, we manually added annotations to help clarify the example.

```
Instruction? 0,map,0,0
Put page table for PID 0 into physical frame 0
Mapped virtual address 0 (page 0) into physical frame 1
#this should error
Instruction? 0,store,7,255
Error: writes are not allowed to this page
#this should update the permissions
Instruction? 0,map,0,1
Updating permissions for virtual page 0 (frame 1)
Instruction? 0,store,7,255
Stored value 255 at virtual address 7 (physical address 23)
Instruction? 0,load,7,0
The value 255 is virtual address 7  (physical address 23)
#this should print an error
Instruction? 0,map,10,1
Error: virtual page 0 is already mapped with rw_bit=1
#let's map a couple other pages
Instruction? 0,map,16,1
Mapped virtual address 16 (page 1) into physical frame 2
Instruction? 0,map,32,1
Mapped virtual address 32 (page 2) into physical frame 3
#Our physical memory should be full at this point, now we need to swap
Instruction? 1,map,0,0
Swapped frame 2 to disk at swap slot 0
Put page table for PID 1 into physical frame 2
Swapped frame 1 to disk at swap slot 1
Mapped virtual address 0 (page 0) into physical frame 1
Instruction? 0,load,7,0
```

```
Swapped frame 3 to disk at swap slot 2
Swapped disk slot 1 into frame 3
The value 10 is virtual address 7  (physical address 55)
Instruction? End of file
```

Checkpoint Contributions

Students must submit work that demonstrates substantial progress towards completing the project on the checkpoint date. Substantial progress is judged at the discretion of the grader to allow students flexibility in prioritizing their efforts. For this project, substantial progress might be defined as supporting the `map, load, store` commands for a single simulated process. **Projects that fail to submit a checkpoint demonstrating significant progress will incur a 10% penalty during final project grading.**

Deliverables and Grading

When submitting your project, please include the following:

- The source code for the memory manager,
- a set of testing files demonstrating the correct operation of your manager,
- output from your tests,
- a Makefile that compiles your code, and
- a document called README.txt explaining your project and anything that you feel the instructor should know when grading the project. Only plaintext write-ups are accepted.

Please compress all the files together as a single .zip archive for submission. As with all projects, please only standard zip files for compression; **.rar, .7z, and other custom file formats will not be accepted**.

The project programming is only a portion of the project. Students should use the following checklist in turning in their projects to avoid forgetting any deliverables:

1. Sign up for a project partner or have one assigned (URL: `https://ia.wpi.edu/cs3013/request_teammate.php`),
2. Submit the project code and documentation via InstructAssist (URL: `https://ia.wpi.edu/cs3013/files.php`),
3. Complete your Partner Evaluation (URL: `https://ia.wpi.edu/cs3013/evals.php`), and
4. Schedule your Project Demonstration (URL: `https://ia.wpi.edu/cs3013/demos.php`), which may be posted slightly after the submission deadline.

A grading rubric has been provided at the end of this specification to give you a guide for how the project will be graded. No points can be earned for a task that has a prerequisite unless that prerequisite is working well enough to support the dependent task. Students will receive a scanned markup of this rubric as part of their project grading feedback.

Groups **must** schedule an appointment to demonstrate their project to the teaching assistants. Groups that fail to demonstrate their project will not receive credit for the project. If a group member fails to attend his or her scheduled demonstration time slot, he or she will receive a 10 point reduction on his or her project grade.

During the demonstrations, the TAs will be evaluating the contributions of group members. We will use this evaluation, along with partner evaluations, to determine contributions. If contributions are not equal, under-contributing students may be penalized.

Project 4 – Virtual Memory – Grading Sheet/Rubric

Evaluation?

Grader: _____

Date/Time: _____

Team ID: _____

Late?: _____

Checkpoint?: _____

Student Name: _____ _____

Student Name: _____ _____

Student Name: _____ _____

Project Score: _____

Earned	Weight	Task ID	Description
_____	5%	0	Correct parsing of input instructions and parameters.
_____	20%	1	Correct implementation of the map, load, and store commands.
_____	25%	2	Correct implementation of the page table data structure, and supporting functions, which stores all the needed information for paging.
_____	25%	3	Pages are correctly swapped to and from disk. Manager handles any cascading faults or evictions that may result.
_____	15%	4	Memory manager supports requests from multiple concurrent processes.
_____	5%	5	Appropriate user testing methodology with example input files. Should include scenarios that result in page faults and scenarios with illegal instructions.
_____	5%	6	Appropriately verbose output (via stdout) that demonstrates the operation of the implemented memory manager.

Grader Notes: