CS 111 Project 6: Virtual Memory
Overview

**Part 1:** implement paging (no demand paging - assuming sufficient physical pages)

**Part 2:** add demand paging with the clock algorithm (physical memory might fill up, and pages must be swapped to disk)

- Write code in `VirtualMemoryRegion` to manage a virtual address space and a page map.
- Write code in `PhysicalMemory` to give physical pages and run the clock algorithm to kick pages to disk.
**Classes**

- **VirtualMemoryRegion**:  
  - Allocates virtual memory  
  - Catch page faults  
  - Map/unmap pages  
  - Maintain a page map

- **PhysicalMemory**:  
  - Manages pool of physical pages  
  - Runs clock algorithm

- **DiskRegion**:  
  - Stores pages to disk (e.g. swap)  
  - Loads pages from disk (e.g. swap, code)
Assignment Structure

Slightly modified mechanism for implementing virtual memory (due to not writing OS code):

- **VirtualMemoryRegion** models a virtual address space of a specified size
- Processes don’t request pages – we assume entire region is ok to access, but not actually mapped until used
- *Page fault* if process accesses unmapped address – runs handle_fault, which should add mapping.
- Accessing again in the same way doesn’t run your code – you just handle new accesses.
Helpful Assignment Types/Functions

**VPage** – type that represents start of a virtual page (really just a pointer)

**PPage** – type that represents start of a physical page (really just a pointer)

**get_page_size()** – returns page size in bytes (guaranteed to be power of 2)
Test Harness

test_harness.cc is the provided testing program – it can run script .txt files in a special format to test your code. The script specifies what code of yours to run and how. Each sanity check test is a script file.

./test_harness somescript.txt

Example: samples/scripts/one_page_read.txt:
# Make a VMRegion with 1 page, and read it
1
INIT 1 1
READ 1 0

See spec for more details on script file format.
Part 1: Paging

Milestone 1: read-only pages
Milestone 2: reading from disk
Milestone 3: Read/Write pages
Milestone 4: Destructor

You will write code only in virtualmemoryregion.hh/cc.
VirtualMemoryRegion

void handle_fault(char *fault_addr);
   Private - called when a page fault occurs – passed virtual address that was accessed

~VirtualMemoryRegion()
   Destructor – called when a region goes away (must unmap / free pages)

void map(VPage va, PPage pa, Prot prot);
   Private - implemented for you – you must call when you want to add/update a mapping

void unmap(VPage va);
   Private - implemented for you – you must call when you want to remove a mapping
VirtualMemoryRegion

Two already-initialized private instance variables:

PhysicalMemory *physical_memory_;
   Use to get and return physical pages

DiskRegion *disk_;  
   Use to store and load saved data from disk
PhysicalMemory

PPage get_new_ppage(VPage mapped_page, VirtualMemoryRegion *owner);
   Call to get physical page

void page_free(PPage p)
   Call to free physical page
### DiskRegion

`bool is_page_stored_on_disk(const VPage vpage);`

Returns whether there is data for this virtual page stored on disk

`void load_page_from_disk(const VPage vpage, PPage dst);`

Reads data from disk for this virtual page into specified physical page
How do we know whether a page should be read-only or read/write?

- Set all new mappings to be read-only (PROT_READ).
- If process writes to that page, it will trigger another page fault; use that as an indicator that the page should be read-write, and update its protections to read-write (PROT_READ | PROT_WRITE).
You will need to maintain a page map instance variable starting in milestone 3.

- Tracks information about mappings across calls to `handle_fault`
- Model as a map data structure (`unordered_map`) that contains only present pages
- You will update the design of your page map over time as you implement more functionality; only add what you need at each milestone.
State of a VPage

Initial

unmapped

PROT_READ

PROT_READ | PROT_WRITE

destructor
destructor

write

access
Part 2: Demand Paging

Milestone 1: clock_sweep
Milestone 2: clock_should_remove
Milestone 3: clock_remove
Milestone 4: Dirty Pages
Milestone 5: Clock Algorithm

You will write more code in virtualmemoryregion.hh/cc and write code in physicalmemory.hh/cc for the clock algorithm.
The Clock Algorithm

- If need PPage but all in use:
  - Check if hand is pointing to removal candidate
  - Not candidate?
    - Indicate swept over
    - Advance hand, try next page
  - Candidate? Kick out:
    - Indicate kicked out
    - Advance hand, stop
- Then get new PPage from pool
void clock_sweep(VPage vp)
    For clock algorithm, called when clock hand sweeps over page and marks unreferenced

bool clock_should_remove(VPage vp)
    For clock algorithm, should return whether page is unreferenced

void clock_remove(VPage vp)
    For clock algorithm, should mark page as kicked to disk
Referenced Bit

From lecture: when clock algorithm sweeps over a page, if referenced = 1, set it to 0 and continue. If referenced = 0, pick it to swap to disk.

For this assignment, instead of referenced bit, we will use page protections.  
PROT_READ or PROT_READ | PROT_WRITE means referenced = 1  
PROT_NONE (new – means no read, no write) means referenced = 0

E.g. in clock_sweep, you must update the corresponding virtual page to have protection PROT_NONE.

If the virtual page is accessed again, we get a page fault and we should upgrade to PROT_READ.
State of a VPage

Initial

- unmapped
  - destructor/clock remove
  - access
  - destructor
- PROT_NONE
  - clock sweep
  - access
  - destructor/clock remove
- PROT_READ
  - access
  - destructor
  - write
- PROT_READ | PROT_WRITE
  - clock sweep
  - write
Dirty Pages

If a page is kicked out of memory, we need to swap it to disk only if it’s dirty (has been modified since being mapped).

We will assume any page that the process attempts to write to is dirty.

**NOTE**: a PROT_NONE page could be dirty! E.g. page written to, then clock hand sweeps over.
void store_page_to_disk(const VPage vpage, const PPage src);

Stores physical page contents to disk, labeled as for the given virtual page.
State of a VPage

- How do you keep track of whether a page is dirty?
- In which states can the page be dirty?
- Which arrows check/update the dirty state?
Clock Algorithm

Write code in PhysicalMemory::get_new_ppage() to check if there are more physical pages, and if not, run the clock algorithm to kick one out.

PhysicalMemory should maintain a vector instance variable with info about each physical page – needed to loop over pages in clock algorithm.

- The index of the vector represents physical page numbers
  - e.g. index 2 means physical page #2
  - how do you get from PPage to physical page number?
- What information do you need for each physical page?
Clock Algorithm

Write code in `PhysicalMemory::get_new_ppage()` to check if there are more physical pages, and if not, run the clock algorithm to kick one out.

`PhysicalMemory` can access “pool” of unallocated pages by:

- `std::size_t nfree()`
  
  Returns number of pages in unallocated pool

- `PPage page_alloc()`
  
  Call within `get_new_ppage` to get a fresh physical page if available

- `void page_free(PPage p)`
  
  Returns page to unallocated pool.
Clock Algorithm

PPage get_new_ppage(VPage mapped_page, VirtualMemoryRegion *owner) {

    // check if pages in unallocated pool
    // if not, run clock algorithm to free up page, you will call the
    clock_ methods that you implemented previously.

    // now get a page from the unallocated pool

}
Final Tips

- Make sure to keep your page map updated
- Make sure to call map() whenever you change protections
- See spec for how to run in GDB
- You can write your own custom script files for testing if you’d like