CS 111 Assignment 6: Virtual Memory

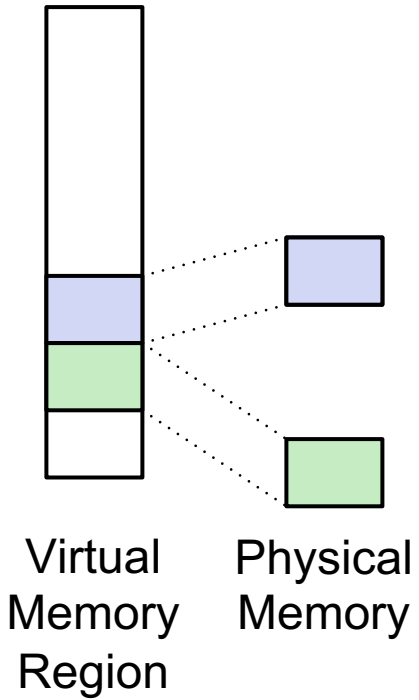
Overview

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

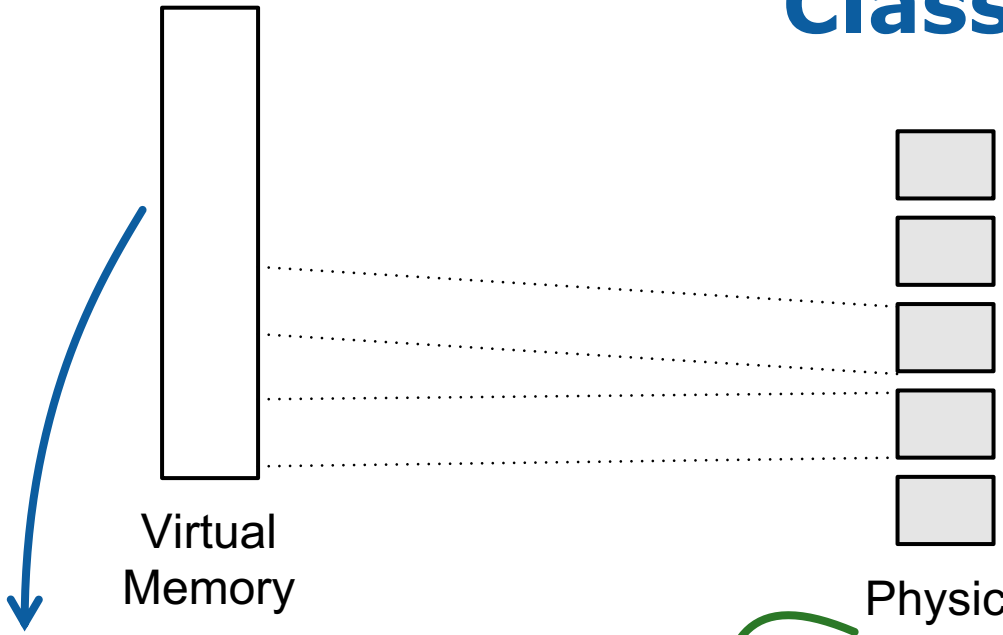
- Write code in **VirtualMemoryRegion** to manage a virtual address space and a page map, map virtual pages to physical pages.

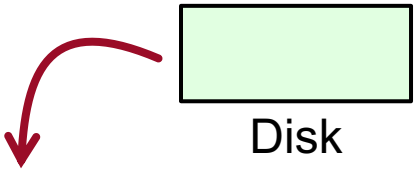
Part 2: add demand paging with the clock algorithm (physical memory might fill up, and pages must be swapped to disk)

- Add code to **VirtualMemoryRegion** to run the clock algorithm if we need a new physical page and there are no more pages.



Classes





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

PhysMem:
- Manages pool of physical pages

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
- We aren't intercepting every memory access – instead, we only intercept *new* memory accesses – these trigger a page fault and cause **handle_fault** to run.
  - ”new access” = accessing previously unaccessed page, or writing to a read-only page
  - Accessing again in the same way doesn't run your code – you just handle new accesses.

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 VirtualMemoryRegion with 1 page, and read it
1
INIT 1 1
READ 1 0
```

See spec for more details on script file format.

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)

Part 1: Paging

Milestone 1: Read-only pages -> get free ppage, map it to accessed vpage.

Milestone 2: Reading from disk -> does the mapped page have initial contents on disk?

Milestone 3: Read/Write pages -> Process might write to a page

Milestone 4: Destructor -> Remove mappings, free physical pages

VirtualMemoryRegion

void handle_fault(char *fault_addr);

Private - called when a page fault occurs – passed virtual address that was accessed

static void set_physical_memory_size(size_t nppages);

Public – called by client to specify how many physical pages there are

~VirtualMemoryRegion()

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

Implemented for you to use:

void map(VPage va, PPage pa, Prot prot);

Private – you must call when you want to add/update a mapping

void unmap(VPage va);

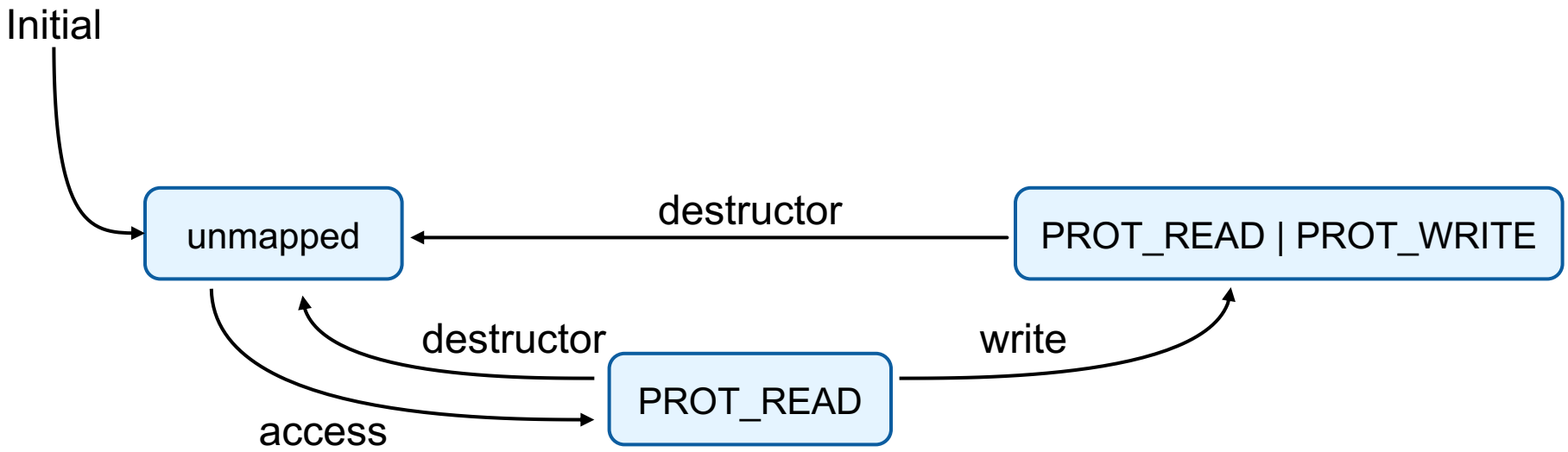
Private – you must call when you want to remove a mapping

Protections

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)

State of a VPage



Milestone 1+: Creating/Using PhysMem

PhysMem(size_t npages)

Initialize a PhysMem with the given number of physical pages

PPage page_alloc()

Call to get physical page

void page_free(PPage p)

Call to free physical page

size_t npages()

Returns number of physical pages in total

size_t nfree()

Returns number of free physical pages

PPage pool_base()

Returns the address of the lowest (first) physical page – all physical pages are contiguous

Note: PhysMem must be created *lazily* – in other words, don't initialize the PhysMem object itself until **set_physical_memory_size** is called.

Milestone 2+: DiskRegion

VirtualMemoryRegion already has a **DiskRegion** instance variable.

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

Page Map

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.

Part 2: Demand Paging

Milestone 1: clock_sweep

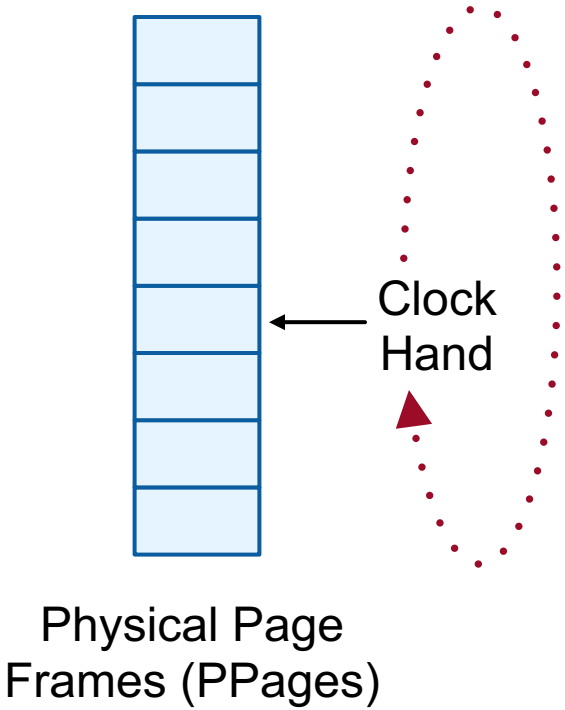
Milestone 2: clock_should_remove

Milestone 3: clock_remove

Milestone 4: Dirty Pages

Milestone 5: Clock Algorithm

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**

VirtualMemoryRegion Part 2

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

Strongly encouraged to add error checking to each to check whether passed-in page is in your page map (to alert you about any erroneous calls)!

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.

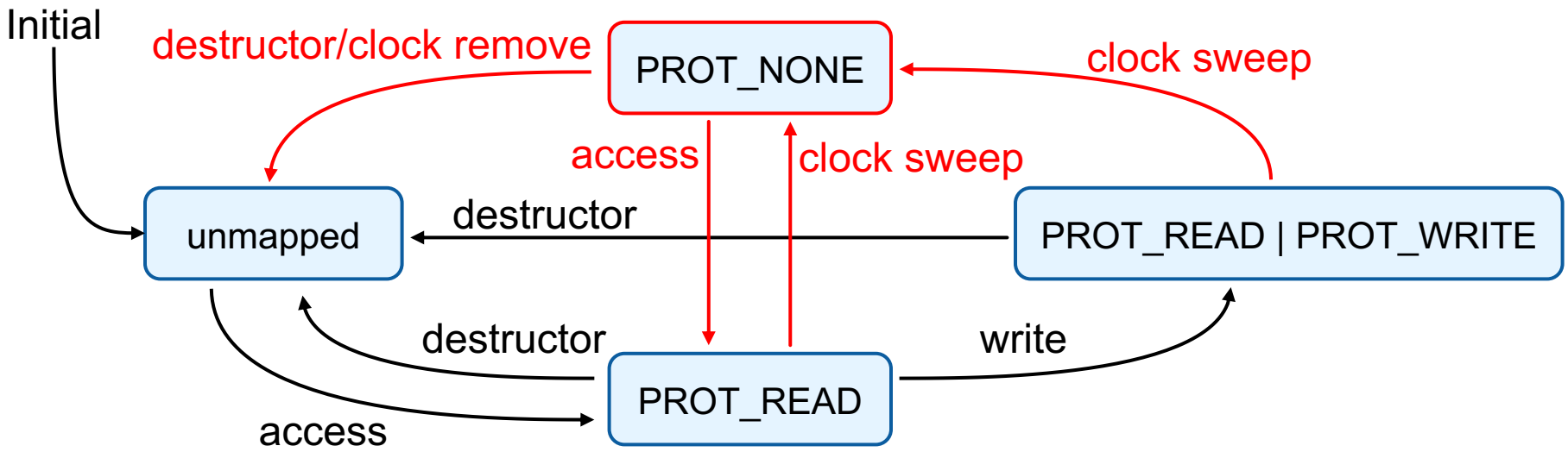
PROT_READ or **PROT_READ | PROT_WRITE** means referenced = 1

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



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.

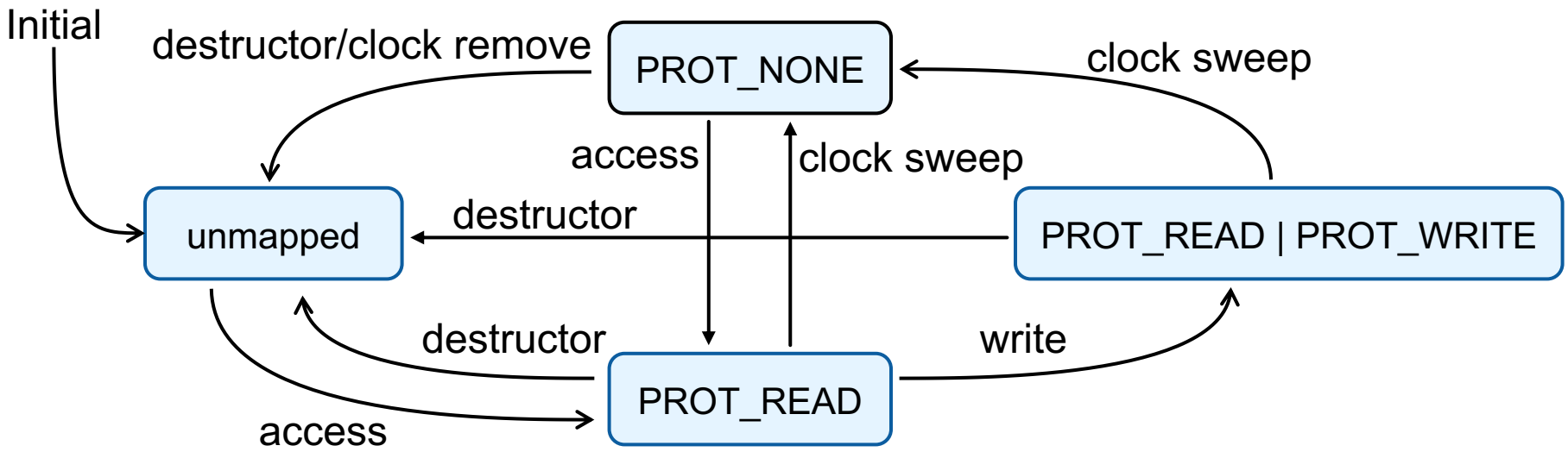
How do we track dirty state?

DiskRegion Part 2

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

Update your code wherever you call PhysMem::page_alloc() to check if there are more physical pages, and if not, run the clock algorithm to kick one out.

You should maintain a fixed-size vector 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?**
  - "Specifically, we need to know, for each physical page, which VirtualMemoryRegion is currently using it, and which virtual page they have it mapped to. This is because we will need to call the clock_ methods to tell them e.g. that one of their pages is being kicked out."

Clock Algorithm

Update your code wherever you call PhysMem::page_alloc() to check if there are more physical pages, and if not, run the clock algorithm to kick one out.

You can access "pool" of unallocated pages via PhysMem:

std::size_t nfree()

Returns number of pages in unallocated pool

PPage page_alloc()

Call to get a fresh physical page if available

void page_free(PPage p)

Returns page to 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
- write your own custom script files for testing (at least 2 required)

- There isn't much code, but what you do write takes some thinking to connect all the pieces together. Make sure to work incrementally!