CS 212 Project 3: Virtual Memory

Spring 2026

Motivation

What does Virtual Memory (VM) solve?

User processes and user virtual addresses were severely restricted up until now.

- When a process loads, all bytes of a process are allocated up front.
  - User stack is limited to one page.
  - No way to release pages that are not in use until a process exits.
- Physical memory is not large enough to hold pages of all running processes.

VM allows you to **give pages on demand and pretend you have access to much more physical memory than you actually do.**

- Achieve the speed of memory with the size of disk (80 / 20 rule).

How does VM solve?

Virtual memory operates through

Two principles:

- Divide physical memory into fixed-size units (**pages**).
- Ensure pages are in physical memory when they are accessed (**page faults**).

Two mechanisms:

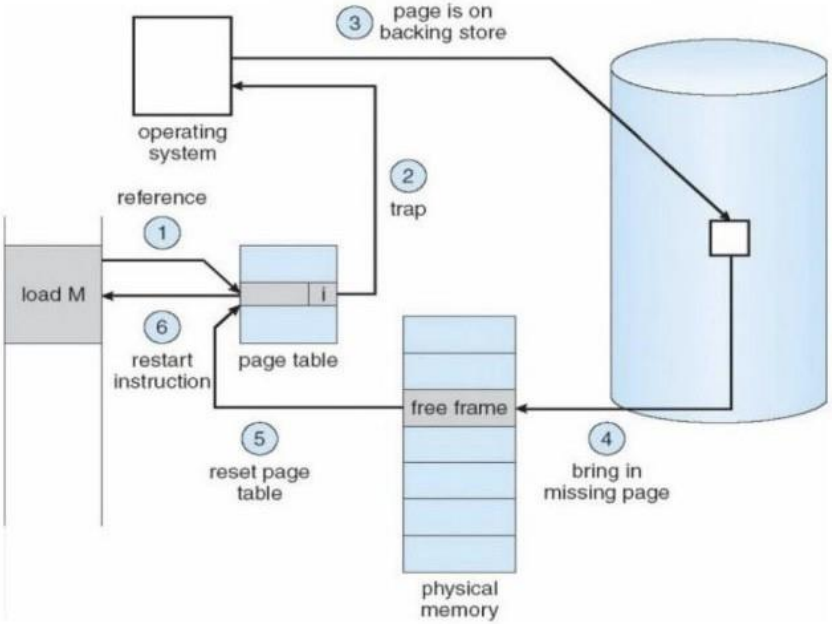
1. **Paging**: Load pages into physical memory when accessed, if not already resident.
2. **Eviction**: Make room for a newly needed page when physical memory is full.

VM Terminology

- **Page (virtual page):** A contiguous, fixed-size region of virtual memory.
- **Frame (physical page):** A contiguous, fixed-size region of physical memory.
  - A resident page is backed by a frame.
- **Swap Slot**: An on-disk slot containing an evicted page.
- **Page Table:** A data structure for mapping pages to frames.

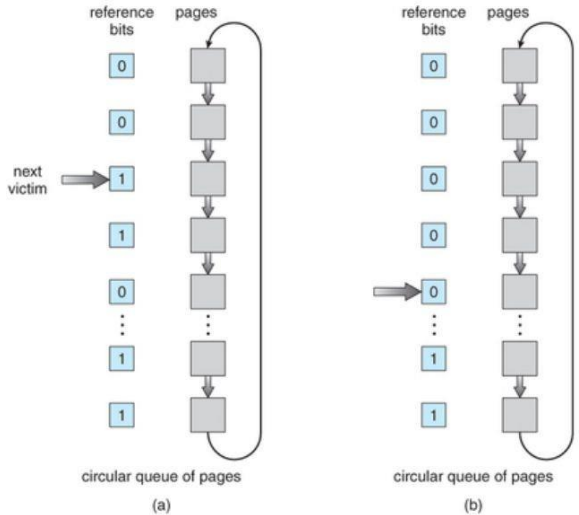
Paging

- Bring in a page from disk if necessary.



Eviction

- Paging requires a free frame - how do we choose which page to evict?
- One strategy: evict the **Least Recently Used (LRU) page.**
  - Approximate using the **clock algorithm.**



Project Design and Details

Memory

- Physical memory is divided into **2 pools**.
  - User pool - palloc_get_page(PAL_USER)
  - Kernel pool - palloc_get_page(0)
- Each process has its own set of **user pages**. The kernel has **global pages** that are active no matter which thread or process is running.
- Access a physical memory address via PHYS_BASE + phys_address.
  - Reason: in Pintos, **frames are mapped directly to kernel pages.**
- CPU sets accessed bit = 1on page read, and dirty bit = 1on page write.
  - OS can set bits back to 0, CPU cannot.

Data Structures

- Required for project 3:
  - Frame Table
  - Supplemental page table
  - Swap table
  - File mapping table
- **Can wholly / partially merge these data structures as you see fit.**
- Make sure that these data structures cannot be evicted.

Frame Table

- Stores mapping between user pages and the frames they occupy.
  - And any other information that might be needed.
- **Goal:** Provide programs with free frames when needed.
  - If the frame table is not full, use palloc_get_page to get a free frame. Otherwise, use the frame table to evict a page.
- **Tip:** The number of frames is fixed because the size of physical memory is fixed. This should simplify the design of your data structure.

Supplemental Page Table

- Supplements the page table with additional information about every page.
- **Goal 1:** Useful when handling page faults.
- **Goal 2:** Useful when deciding which resources to free when a process terminates.
- For example, may keep track of what pages a process owns.

Project Requirements: Swap Table

- **Goal:** Efficiently store and retrieve dirty evicted pages.
  - A small partition of the filesystem where dirty pages go when they can’t live anywhere else.
  - For example, evicting dirty stack pages.
- **Tip:** Start simple. Evict a random page instead of doing something fancy to make sure swap works.

Project Requirements: File Mapping Table

- Track which pages are used by each memory mapped file.
  - Used by mmap() and munmap().
- Stores the pages in use by a memory mapped file.

Project Features

Paging

- A page fault may be caused by a **dereference of an address which is not in memory.**
- To page in the necessary page, modify the page fault handler to do the following:

1. Locate the page that faulted in the SPT.
2. If the reference is valid, use SPT to locate the page's data. It could be in the filesystem, in a swap slot, or be an all-zero page.
    a. If the reference is invalid, kill the process.
3. Obtain a free frame to store the page.
4. Fetch the data into the free frame.
5. Update the page table entry to point the virtual address to the new physical address.

Eviction

- If no frames are available for a page during paging in, **then a page must be evicted from its frame.**
- You must implement an algorithm for eviction at least as good as the clock algorithm.

1. Choose a page to evict using the algorithm.
2. Remove references to the frame from any PTE that refers to it..
3. If needed, write the evicted page to file system or swap.
    a. Other processes should not have to wait for this to complete.

Stack Growth

- In project 2: the stack was limited to a single page.
  - Now: allocate a new stack page if the stack grows beyond its current page (**page fault** from stack access).
- Only allocate a new stack page if an access appears to be a stack access.
  - Devise a heuristic to tell if an access is a stack access.
- Stack pages can also be evicted.
- Impose an absolute limit on stack size for a process.

Memory Mapped Files

- Allows you to use demand paging on the data of a file.
  - One-to-one correspondence between the contents of a file and memory.
- Implement the following functions.
  - mapid_t mmap (int fd, void *addr)
    - Maps file open at fd into consecutive virtual pages starting at addr.
    - Lazily load file data into pages when access occurs.
    - When page is evicted, load data back into file.
  - void munmap (mapid_t mapping)
    - Unamps mapping designated by mapid_t returned by prior call to mmap.
    - All mappings remain until process exits or munmap is called.

Accessing User Memory

- A page may be evicted from its frame while it is being accessed by kernel code.
  - For example, you might evict a buffer from a upage that a system call is midway through accessing.
  - **What if you acquired the file system lock before reading?**
- Make your kernel handle such page faults, or prevent them from occurring in the first place.
  - You can implement **pinning** i.e. marking certain pages as unevictable.
  - This restricts the amount of memory available, so only pin them when needed.

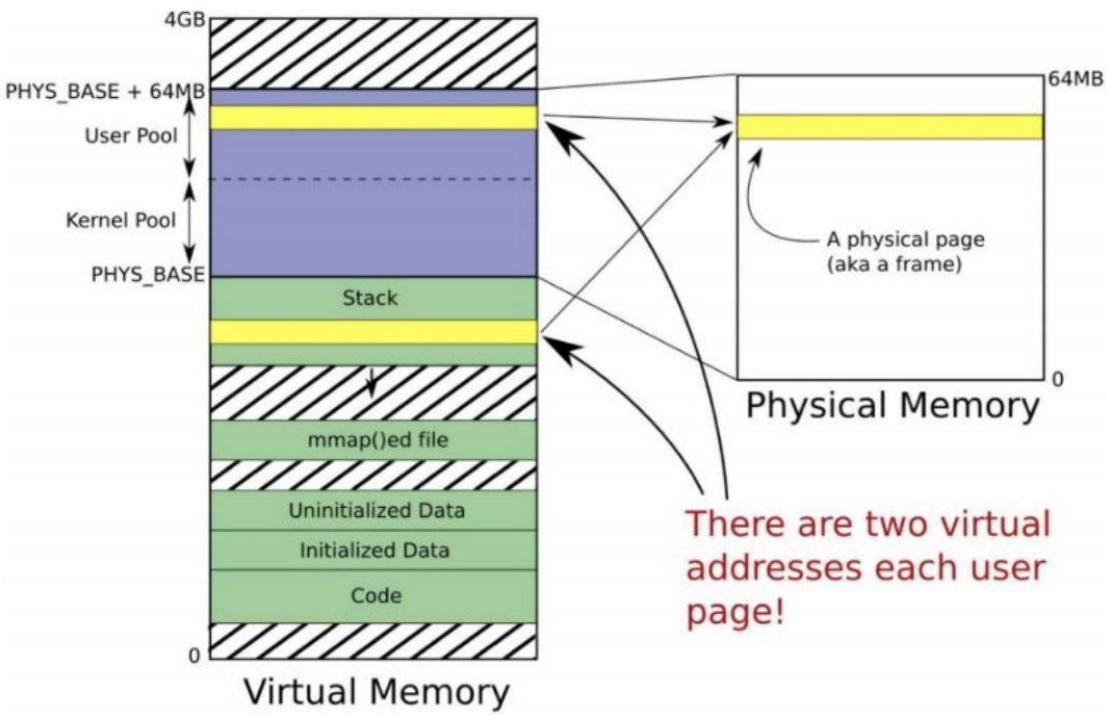
Considerations

A Note on Synchronization

"If one page fault requires I/O, in the meantime processes that do not fault should continue executing and other page faults that do not require I/O should be able to complete. ***This will require some synchronization effort.****"*

- Think about interactions with synchronization mechanisms that would not correctly satisfy this property.
- There are other interactions (evicting frames, freeing resources etc.) that all need synchronization.

Aliases


4GB
PHYS_BASE + 64MB
User Pool
Kernel Pool
PHYS_BASE
Stack
mmap()ed file
Uninitialized Data
Initialized Data
Code
0
Virtual Memory
64MB
A physical page
(aka a frame)
0
Physical Memory
There are two virtual addresses each user page!

Aliases

- In Pintos, every user virtual page (upage) is aliased with a kernel virtual page (kpage).
  - Every frame can be accessed from a user virtual address and a kernel virtual address.
- Accessing a frame **will only set the accessed and dirty bits for the PTE of the virtual page used to access the frame.**
  - In order to check if a frame has been accessed, you either need to check both PTE's or only access frames through the user virtual page.

Getting Started

Suggested Order of Implementation

1. Fix all remaining project 2 bugs.
2. Implement the Frame Table without eviction.
3. Implement the Supplemental Page Table and paging.
4. Implement stack growth, memory mapped files and freeing pages on process exit.
5. Implement eviction.

Tips: Data Structures

Important: Use existing data structure implementations (lists, hash tables, etc.) and don't waste your time implementing anything more advanced. It's not worth it.

Remember your basic tradeoffs:

- **Size:** fixed vs. dynamic. Bitmaps and arrays are easy when size is fixed.
- **Access**: random vs. linear. Do you need iteration or fast lookups?
- **Ownership**: private vs. global. Is the data owned by a single process or global?
- **Synchronization:** coarse vs. fine. Should you lock everything or only each element/bucket/bit?

Tips: General

- **Start early:** This is the hardest project yet, possibly all quarter.
- **Leverage Existing Code:** For example, vaddr.h contains methods like pg_ofs and pg_no.
- **Careful Synchronization:** This assignment typically has the most deadlocks.
- **Be Lazy:** All allocations for this assignment are done lazily. You only ever evict one page, read one page, etc.
- **Add New Files:** Keep your code clean.

Project 3: A Concrete Walkthrough

CS 212 Section

The machine

Hardware

- 16 frames of RAM (= 64 KB)
- 8 swap slots (= 32 KB on disk)
- Page size = 4 KB

Workload

- Process A: `grep`
  2 code + 1 data + 2 stack = 5 pages
- Process B: `cat data.txt`
  1 code + 1 data + 1 stack + 3 mmap = 6 pages

The numbers don't fit
11 virtual pages

8 user-pool frames

⇒ **at least 3 pages can't be resident.**

Some sit in swap.
Some are still "lazy" (file-backed, never faulted in).

Step 1: Pintos splits RAM into two pools at boot

- `palloc_get_page(0)` → kernel pool. Used for kernel data structures that are *never paged*: page directories, your VM bookkeeping, etc.
- `palloc_get_page(PAL_USER)` → user pool. The actual user data pages live here. **This is what we evict.**

The handout says: “frames used for user pages should be obtained from the user pool.”

Frame Table: tracks user-pool frames

The handout's spec: "one entry for each frame that contains a user page."

What it needs to answer: which user page (if any) is currently resident in each user-pool frame?

In our example, all 8 user-pool frames are occupied:

frame	kernel addr	what's resident
0	0xC0008000	A's code page `0x08048000`
1	0xC0009000	A's code page `0x08049000`
2	0xC000A000	A's data page `0x0804A000`
3	0xC000B000	A's stack page `0xBFFFF000`
4	0xC000C000	B's code page `0x08048000`
5	0xC000D000	B's stack page `0xBFFFF000`
6	0xC000E000	B's mmap page `0x10000000`
7	0xC000F000	B's mmap page `0x10001000`

Supplemental Page Table: per-process bookkeeping

The handout's spec: supplements the hardware page table with extra data, so the kernel can handle a fault and so it can free the right resources at process exit.

For each virtual page that exists, the SPT has to answer:

- Where do I find this page's data right now?
- Is it writable?
- When the process exits, what do I free?

A's address space has 5 pages; B's has 6. So A's SPT has 5 entries and B's has 6 (one per virtual page that "exists" in the process).

Process A: 5 virtual pages, 4 different states

vaddr	kind of page	where the data is right now
`0x08048000`	code (read-only)	resident in frame 0; backed by `grep` executable
`0x08049000`	code (read-only)	resident in frame 1; backed by `grep` executable
`0x0804A000`	data (writable)	resident in frame 2; came from `grep` executable
`0xBFFFE000`	stack	**evicted to swap slot 3**
`0xBFFFF000`	stack	resident in frame 3

Notice the variety:

- Two pages are file-backed and read-only.
- One is file-backed but writable (data segment).
- One was paged in, modified, and later evicted to swap.
- One is currently resident as a stack page.

The SPT has to record enough to make sense of all of these.

Process B: 6 virtual pages, including lazy and mmap'd

vaddr	kind of page	where the data is right now
`0x08048000`	code (read-only)	resident in frame 4; backed by `cat` executable
`0x0804A000`	data (writable)	**not yet faulted in**; will come from `cat`
`0xBFFFF000`	stack	resident in frame 5
`0x10000000`	mmap of `data.txt`	resident in frame 6 (modified)
`0x10001000`	mmap of `data.txt`	resident in frame 7 (clean)
`0x10002000`	mmap of `data.txt`	**not yet faulted in**

- Two pages are *lazy*: they have an SPT entry saying "come from a file when accessed," but no frame yet.
- Three are mmap'd. The handout says these get written back *to the file* when modified, not to swap. The SPT has to know this.

The four states a virtual page can be in

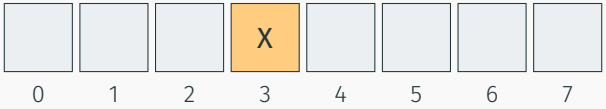
1. **Resident in a frame.**
   The hardware PTE points at a frame; everything else is fast.
2. **In swap.**
   Was resident, got evicted as a dirty private page (stack, data segment). Re-readable from the swap partition.
3. **File-backed (not resident).**
   Either never faulted in, or evicted clean. Re-readable from the file.
4. **All-zero.**
   BSS, or a stack page that grew but was never written. The kernel manufactures the zeros on demand.

The page-fault handler (handout, 4 steps): locate in the SPT, obtain a frame, fetch the data into the frame, install the PTE.

Swap Table: which slots are in use

The handout's spec: the swap table should pick a free slot when evicting, free a slot when its page is read back in, and free slots when the owning process dies.

8 swap slots, only one currently in use:



Slot 3 holds A's middle stack page (`0xBFFFE000`). The SPT entry for that page knows to look at slot 3.

Tip from the slides: a bitmap is a natural representation when the size is fixed and you just need "in use vs. free."

File Mapping Table: per-process active mmaps

A's mappings: *empty.* A doesn't use mmap.

B's mappings: one active mapping.

```
mapid = 2
file: data.txt
base address: 0x10000000
3 pages mapped
```

This table needs to answer: when `munmap(2)` is called, what range of pages do I tear down? When the process exits, what mappings do I clean up?

What lives where

structure	scope	notes
Frame table	global	one entry per user-pool frame, fixed size
SPT	per-process	one entry per virtual page that exists
Swap table	global	one entry per swap slot, fixed size
Mappings	per-process	one entry per active mmap region

Crucial invariant: all of this bookkeeping has to be in *non-pageable* memory (the kernel pool). If your SPT or frame table were itself pageable, a fault on the bookkeeping would need the bookkeeping to handle it.