國立臺灣大學 National Taiwan University

EE 5173 Operating System
lab 03: Virtual Memory

Chen-Yin, Lee (李臻茵)
r13921090@ntu.edu.tw

2025/04/30

Contents

Goal

Implement **virtual memory** on Pintos

- Page table management
- Swap in/out page
- Stack growth

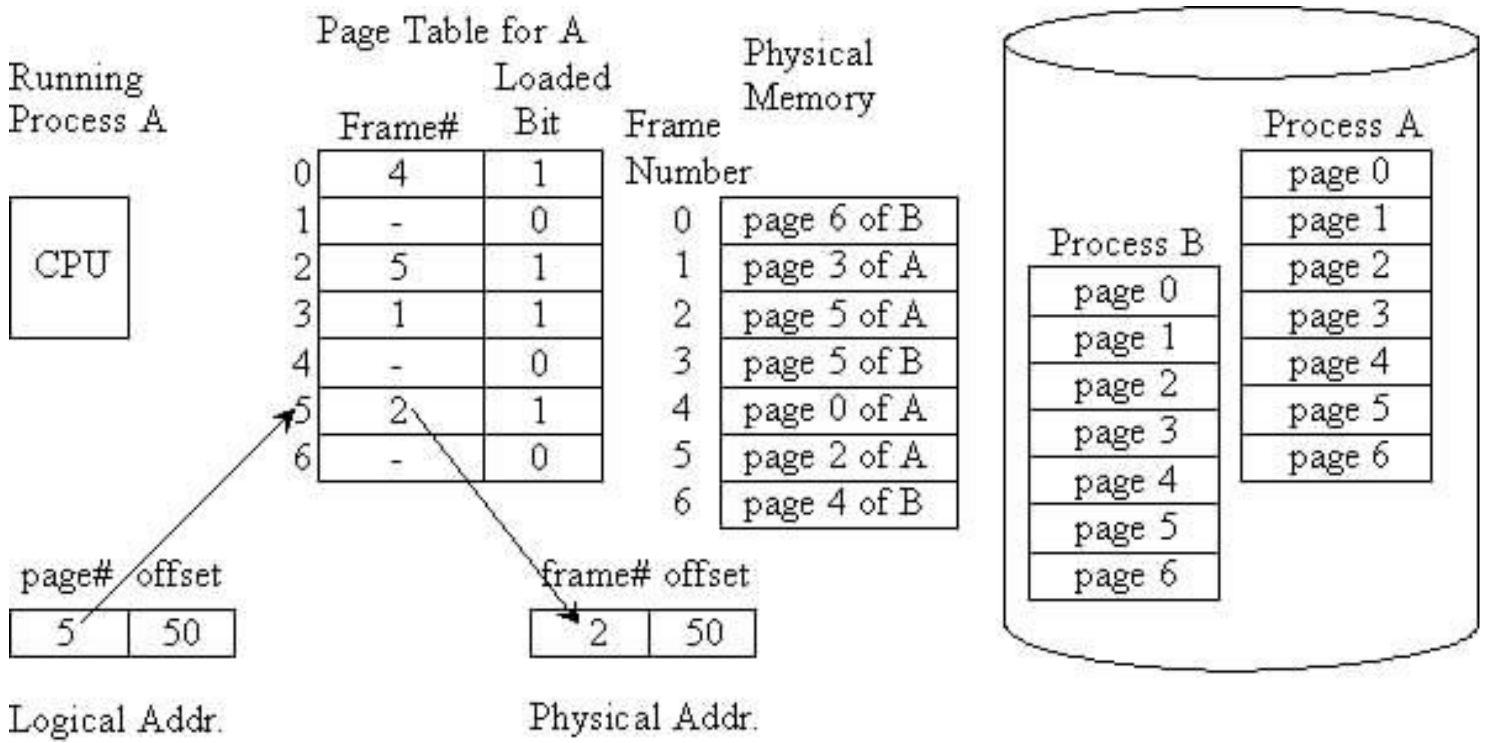
Why should we use VM?

Programmer’s view: one large, continuous address space

□ no need to track physical RAM or manual memory management.

- To simulate larger memory than physical’s.
- To isolate user program’s memory area.
- To make user programming easy (abstraction)

Virtual Memory (Ch.9)



Page Table Management - Quick Recap

- **Demand Paging & Page-Fault Flow**（lazy page-in）
- **Page Fault Rate & Locality**：

  Determine the min-num of frames with working set that should be configured
- **Page Table & TLB**：

  TLB hit can avoid an extra memory reference

  Two-level / multi-level or inverted for large-size page table
- **Replacement / Allocation**：

  Tradeoff between increasing multitasking and reducing page faults

Supplemental Page Table (SPT)

software-based "Extended Page Table" to supplement the hardware PTE

Hardware PT can't do	**What to supplement in**	**Typical scenarios**
can't remember where to load	Source information: file offset, swap slot, anonymous page	Lazy load、swap-in/out
without "not-present but valid"	Legal page description: frame is only used when fault occurs	Demand paging、stack growth
Don't know "Whose I/O is currently occupying this page”	Pin / In-flight logo	Preventing eviction during I/O
Paging algorithm needs to list "all pages in use"	Process-local page hash / index	Clock/LRU victim search

Resource Management Overview

You will need to design the following data structures:

- **Supplemental page table**

  Enables page fault handling by supplementing the hardware page table with additional data about each page, because of the limitations imposed by the page table's format.

- **Frame table**

  When none is free, a frame must be made free by evicting some page from its frame.

- **Swap table**

  When swapping, picking an unused swap slot for evicting a page from its frame to the swap partition, and allow freeing a swap slot when its page is read back or the process whose page was swapped is terminated.

Task 1: Paging

- All of these pages should be loaded lazily, that is, only as the kernel intercepts page faults for them.
- Upon eviction:
  - Pages modified since load should be written to swap.
  - Unmodified pages, including read-only pages, should never be written to swap because they can always be read back from the executable.
- Implement a global page replacement algorithm that approximates LRU.
  - Your algorithm should perform at least as well as the simple variant of the "second chance" or "clock" algorithm.

SPT @ Page Fault Point

page_fault() in userprog/exception.c

- Locate the page that faulted in the supplemental page table.
    - Invalid Accesses: Any invalid access terminates the process and thereby frees all of its resources.
- Obtain a frame to store the page.
- Fetch the data into the frame, by reading it from the file system or swap…
- Point the page table entry for the faulting virtual address to the physical page. (can use functions in userprog/pagedir.c)

[Tips] Highly recommended: use printf() for debugging.

Task 2: Accessing User Memory

- Adjust user memory access code in system call handling to deal with potential page faults.
- While accessing user memory, your kernel must either be prepared to handle such page faults, or it must prevent them from occurring.

Hint

- Don't let page faults happen while holding locks (e.g., during read()).
- Pin user pages before access in kernel.
- Unpin quickly to avoid blocking eviction.

Task 3: Stack Growth

- Purpose: Allows automatic stack growth during execution
- Allocate additional pages only if they "appear" to be stack accesses. Devise a heuristic that attempts to distinguish stack accesses from other accesses.

Hint

- When to trigger? Page fault near f->esp - 32 in user space
- Access ESP: Use f->esp from struct intr_frame *f (only in user mode)
- Stack Limit: Enforce a maximum size, e.g., 8 MB
- The first stack page can be allocated at load time; others are demand-paged

About project source code

- To avoid handling with any errors not related to this project, TA provided the source code for you to start with, which has already passed all the tests mentioned in lab01, and vm/* you need to implement can also be compiled correctly.
  - However, you can still use the code you completed in lab01 to continue, but please note that lab03 requires that all lab01 tests can still pass after adding the virtual memory mechanism.
- The scope you will need to implement/modify:
  - vm/*
  - userprog/exception.c
  - userprog/process.c
  - (threads/thread.ch)

Test Cases List

All tests in tests/userprog & tests/filesys/base

Part of the tests in tests/vm:

Frame Allocation & Merge	**Invalid Access Handling**	**Stack Growth**
page-linear	pt-bad-addr	page-merge-stk
page-parallel	pt-bad-read	pt-grow-stack
page-shuffle	pt-write-code	pt-grow-stk-sc
page-merge-seq	pt-write-code2	pt-big-stk-obj
page-merge-par	pt-grow-bad	pt-grow-pusha

- For more details, please check source file under tests/

Requirement in tech report

1. **Page Table Management**

   1.1 Data Structures: list & state purpose for your new or modified item.

   1.2 Algorithms:

   SPT Lookup / Update – how your code accesses and updates SPT entries for a given page.

   Accessed & Dirty-Bit – how you keep A/D bits consistent for aliased kernel/user mappings.

   1.3 Synchronization: how races are prevented when two user processes simultaneously request

2. **Paging between Disk**

   2.1 Data Structures: As above, but only items related to paging and swap.

   2.2 Algorithms

   Frame Eviction Policy – how you select a victim frame.

   Frame Reuse Adjustment – when process P reuses a frame that belonged to process Q.

Requirement in tech report

2.3 Synchronization

VM-Wide Locking Strategy – basic design and how it avoids deadlock.

Cross-Process Eviction Safety – protecting Q while P evicts Q’s page.

I/O In-Flight Protection – shielding a frame that is currently being read in.

Syscall-Time Page Access – handling page faults or “locked” frames during kernel system-call code.

3. Stack Growth

3.1 Stack-Fault Heuristic: the conditions under which a page fault on an invalid address triggers automatic stack growth.

Grading Policy

- Test score - 30%
- Report - 70%
  - Task 1 & 2 - 40%
  - Task 3 - 30%

Deadline 2025/5/21 23:59

- Filename (-3%)
- Late submission (-10%), over 1 weeks (-20%) over 2 weeks (Immediate 0 points)
- Plagiarism or All/Most LLM (Immediate 0 points)
- **Bonus (pass all lab3 vm test: Max 15)**

Report Format

- Content format: 12pt front,16pt row height, and align to the left.
- Caption format: 18pt and **Bold font**.
- Figure: center with single line row height.
- Upload with the file structure format :

```
G[team number]_3.zip (e.g. G01_3.pdf)
│── G[team number]_3.pdf
│── pintos
│     ├ devices
│     ├ …
│
```

Be sure to unzip and double check your .zip before uploading!!!

Reminder

- The test score is considered passed only if the TA job passes.
- 0 will given to cheaters. Do not copy & paste!
  - TA will check your repository
- Feel free to ask TA questions, but TA won’t help you debug your code.
- Have fun ☺

Related stuff for this lab

- GitHub repository:

  ```
  git clone https://github.com/teresasa0731/pintosLab.git
  ```

- Explanation video from TA [link]