CS202(-002): Lecture 12 Virtual Memory III

Plan
1. Weensy OS
2. Page faults

Where we are

1. Virtual address —(Page table)→ Physical address
2. Page table ⟶ Use a tree for efficiency
   - ↳ How to translate
3. TLB ↔ Cache translations to reduce translation cost.

WEENSY OS

- Not a substitute for recitation

That was yesterday

Rather some additional thoughts

- General Suggestions

a. Lab 4 and Weensy OS is a really cool learning tool
   → OS Kernel + Code covering everything we have covered so far

Can see everything you want

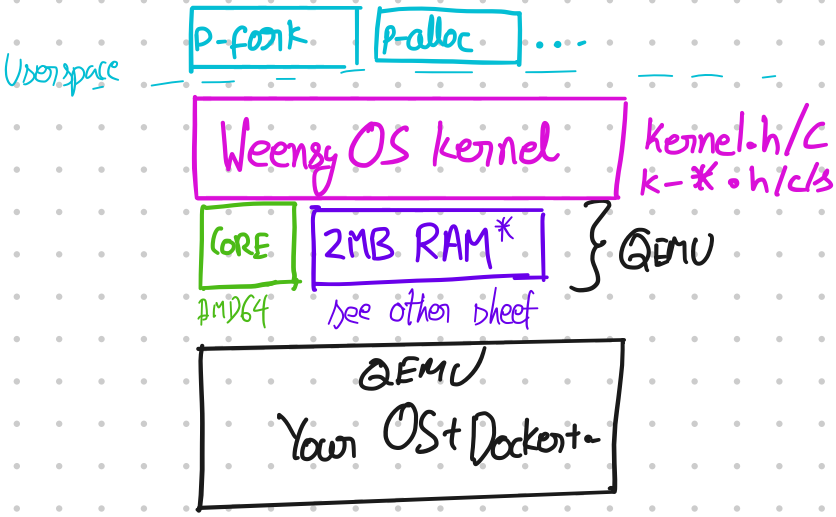
b. Lab 4 can take a bunch of time

→ Bugs can be unusually hard

→ Debugging can take a while
  ↳ Bug might mean no print output
  → Might require reading & reasoning about code

START EARLY, BUDGET TIME!

STRUCTURE



Your goal

CREATE & MAINTAIN ADDRESS SPACES FOR EACH PROCESS

Tracking virtual & physical memory
  ↳ Virtual

kernel.h

```
struct proc {
    ...
    x86_64_pagetable *p_pagetable;   <-> Pagetable
};
```

k-hardware.c - Search using virtual_memory_lookup

↳ Manipulate using virtual_memory_map

→ Physical

Kernel.c

struct physical_pageinfo pageinfo [# of physical pages]

- → owner → Process ID / FREE / RESERVED / KERNEL
- → refcount → # of page tables where page is mapped

Update using assign_physical_page

Page Fault - Demand Paging

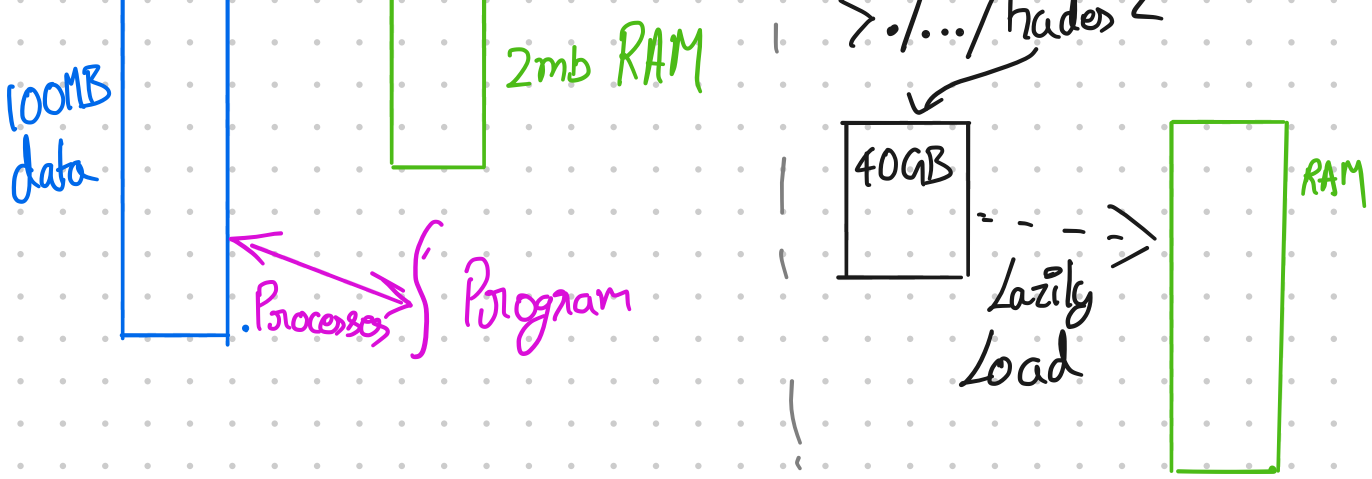
→ What?

- Exception because a virtual address cannot be translated into a physical address or access is disallowed by translation
- Why?

→ So far treated exceptions as error

↳ TODAY: LEVERAGE EXCEPTIONS FOR GOOD

Two goals



How? (1) OS tracks

- What pages are in (resident in) physical memory (RAM)
  - Marked P in Page table
- Mapping for pages not in RAM (swap)
  - Where to find their data.

Note: Some pages may be in neither category. Why?

(2) On page fault for page that is not currently in RAM

- (I) OS finds or makes (EVICTION) space in RAM — allocates a page

Paging In

{ (II) Loads Data into this page
(III) Changes page table to mark page as present

(IV) Resumes execution
↳ Which retries access.

Things we need for step 2

↳ Address that caused the fault

In CR2 ↔ Full address that caused the fault.

Also pushes "cause" on the stack

P	0 The fault was caused by a non-present page. 1 The fault was caused by a page-level protection violation.
W/R	0 The access causing the fault was a read. 1 The access causing the fault was a write.
U/S	0 A supervisor-mode access caused the fault. 1 A user-mode access caused the fault.
RSVD	0 The fault was not caused by reserved bit violation. 1 The fault was caused by a reserved bit set to 1 in some paging-structure entry.
I/D	0 The fault was not caused by an instruction fetch. 1 The fault was caused by an instruction fetch.
PK	0 The fault was not caused by protection keys. 1 There was a protection-key violation.
SGX	0 The fault is not related to SGX. 1 The fault resulted from violation of SGX-specific access-control requirements.

I) OS finds out

Lab4

makes (EVICTION) ← Now!

space in RAM — allocates a page

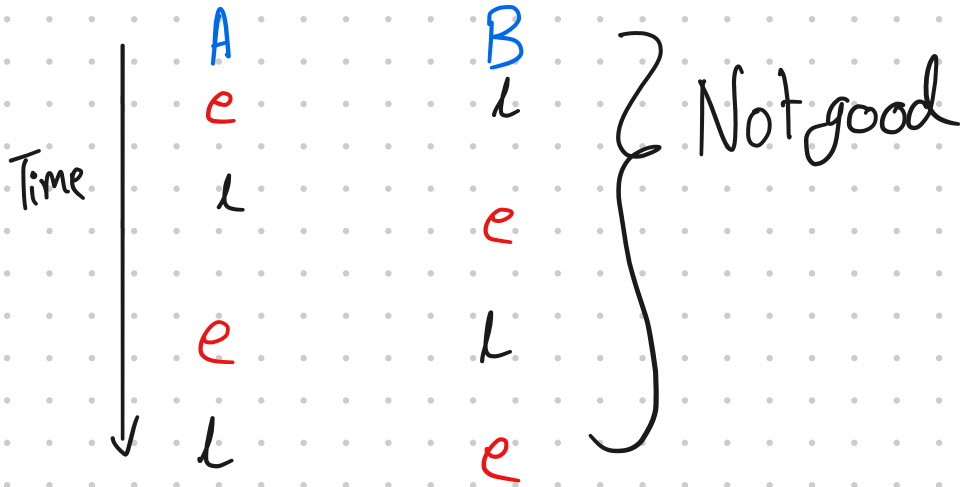
Disks & FS later ← II) LOADS DATA into this page

You know how! (Lab4) ← III) Changes page table to mark page as present

IRET ← IV) Resumes execution

EVICTION

- Goal
  - Minimize # of times data is paged in



Why? Cost!

- What do we want:

Ideally, only evict pages that will not be accessed for a 'long' time

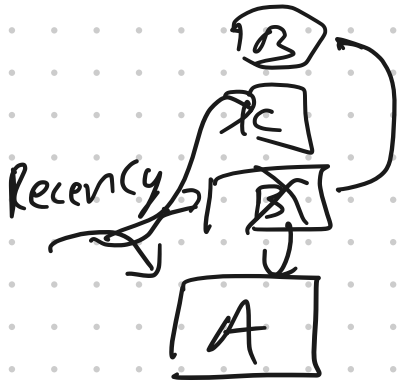
Beladi's MIN

- Problem → Ideal policy REQUIRES knowing the future

But we do not

- Solution: Policies that try to predict based on past behavior

Least Recently Used



A A A A A A A A A B B C B B C C B | ✗

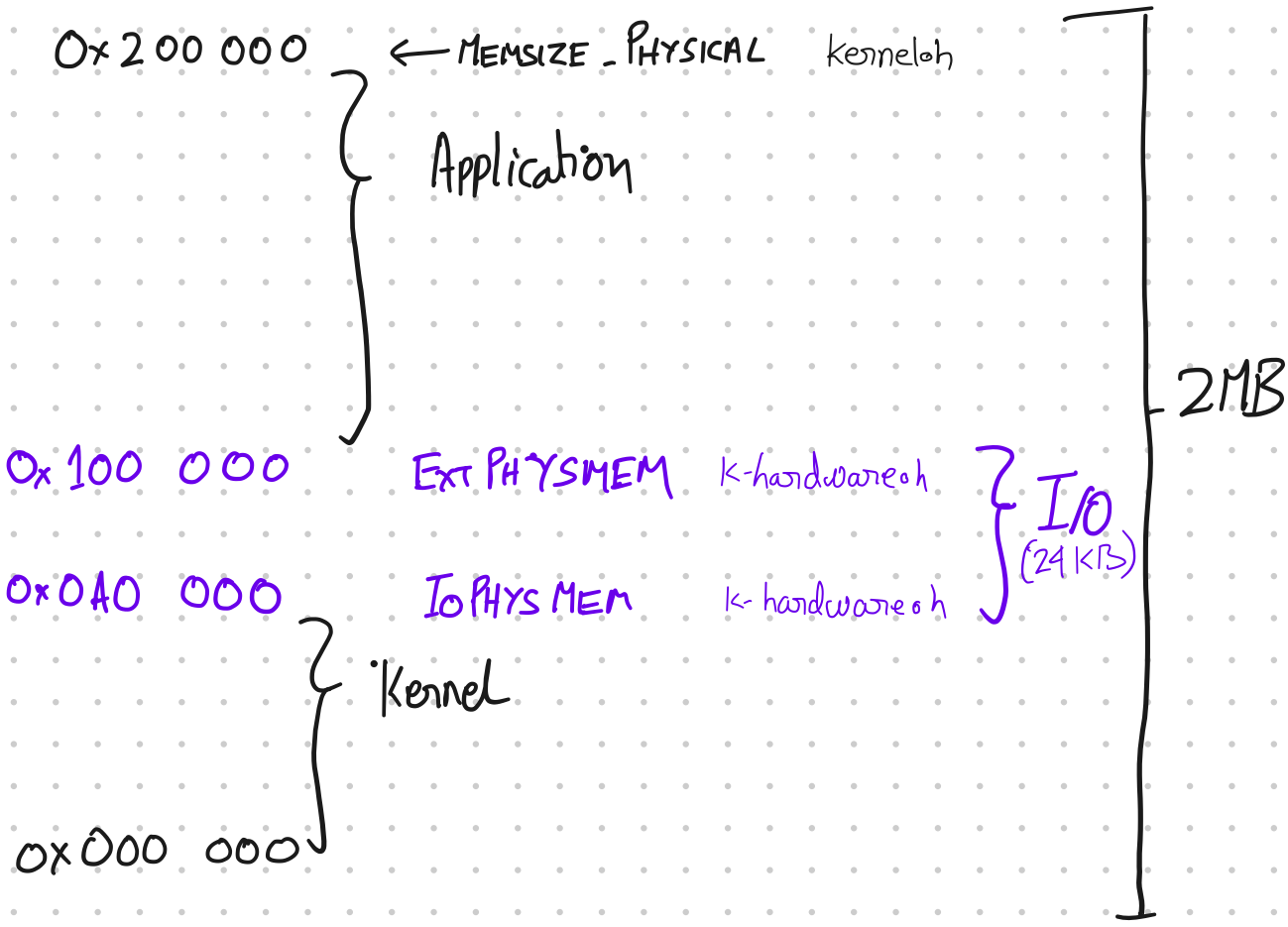
Evict

Least FREQUENTLY Used

A A A A A A A A A B B C B B C C B | &

Evict

Ag



call $\xrightarrow[RIP]{push}$ Stack

syscall $\xrightarrow[rIP]{push}$

movq (...), %rax
→ addq %rax, %rbx

Syscall
→ movq %-rx

= *((uint64_t*)(PTE >> 1))