Complete virtual memory systems

CS 537: Introduction to Operating Systems

Louis Oliphant & Tej Chajed

University of Wisconsin - Madison

Spring 2024

Administrivia

- Project 3 out – Due Feb 20th @ 11:59pm (**tonight**)
- Code reviews: Signup for 15min slot. TA will give feedback on your P3 code. Grading is based on completion.
- Midterm 1 (more details on next slide)

Administrivia: midterm 1

- Regular Time: Feb 23rd, 5:45-7:15, Humanities 2650 (Lec 001), Humanities 3650 (Lec 002)
- Unable to attend? Fill out this form: https://forms.gle/7wPNekXjamkam8Q86
- Alternate Time: Feb 23rd, 7:30-9pm, CS 1325
- McBurney Time: Feb 23rd, 5:45-8pm, CS 1221
- Bring **#2 Pencil** and **UW Student ID**
- Review Material in Canvas → Files → Shared Old Exams

Review: Beyond Physical Memory

- Idea: store unreferenced pages on disk (*swap space*)
- **Mechanisms:** Add present bit to PTE to track if page is in memory or disk, restore them during *page fault handler*
- Replacement **policy**: which *victim page* to swap to disk? (algorithms like LRU, Clock)

Agenda: what do real virtual memory systems look like?

- Kernel virtual memory layout
- Lazy optimizations (eg, copy on write)
- Huge pages
- Security: ASLR and KASLR

Motivation

- Understand virtual memory features beyond the basics
  - Copy-on-write, larger pages, ASLR
- Talk about performance and security

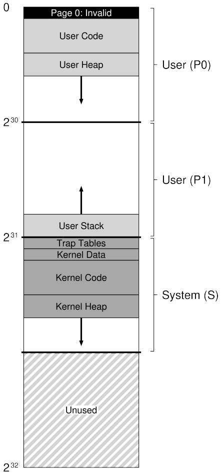
Kernel virtual memory layout

So far: virtual memory = code + heap + stack

Real layout:

- Make page 0 invalid (so NULL dereferences fail)
- Map kernel into each process's virtual memory
- Linux: "kernel logical memory" is mapped linearly to physical memory
- Need to protect kernel from user code: privilege bits in PTEs

VAX/VMS virtual memory



Lazy optimizations: demand zeroing

Need to zero a page to clear sensitive data, wasteful if process doesn't use the page

Demand zeroing:

- On allocation: map page but mark PTE invalid, remember that it is "to-be-zero'd"
- On page fault: zero page and map into process
- No work if page is never used

Lazy optimization: copy-on-write

- Copying a page from one process to another is expensive, wasteful if not written to
- Share physical page until one is written, then copy
- Add a **reference count** (refcount or rc) to each physical page
  - read-only if rc > 1
  - writable if rc $= 1$
  - unused if rc $= 0$

Summary: copy-on-write

- Useful for shared libraries
- Critical to make `fork()` and `exec()` work
- Technique is more broadly useful with dynamic sharing

Larger pages

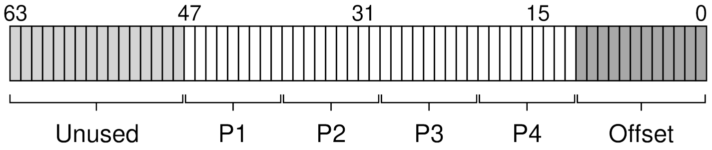
x86-64 supports 2MB and 1GB pages as well

- Main motivation: better use of TLB

  A 64-entry TLB with 4K pages can hold mappings for only 256KB of memory

- Secondary benefit: makes address translation on TLB miss faster

4-level paging



2MB page mapping

From Intel SDM chapter 4.5

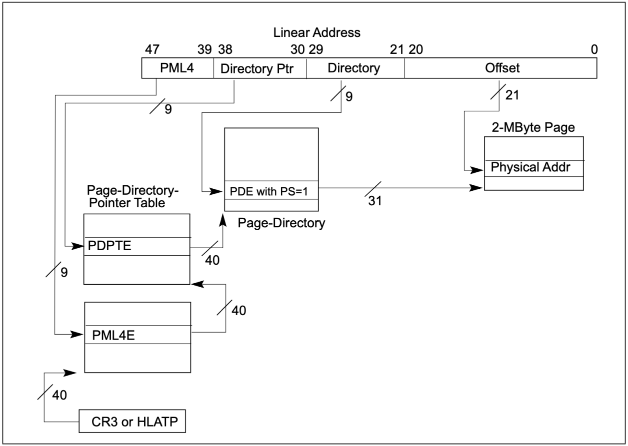


Figure 4-9. Linear-Address Translation to a 2-MByte Page using 4-Level Paging

Huge pages in Linux

- First version: request explicitly in `mmap()`
- **Transparent huge pages** more recently
- Costs: internal fragmentation, slower allocation, defragmentation costs

Summary: larger pages

- Main idea: better TLB hit rate
- Larger memory sizes make this more important
- Linux added support incrementally

Security considerations: buffer overflows

```
int some_function(char *input) {
    char *dest_buffer[100];
    strcpy(dest_buffer, input); // buffer overflow
}
```

What can attacker do with this?

Return-oriented programming (ROP) means essentially anything

Address space layout randomization (ASLR)

Instead of putting code at predictable locations, **randomize virtual addresses**

Should still avoid buffer overflows, but ASLR reduces their impact

Some attacks are still possible

Summary

Real virtual memory systems have more features for performance and security

- Lazy optimizations (demand zeroing, copy-on-write)
- Larger page sizes improve TLB performance
- ASLR improves security