

CSE 120

Principles of Operating Systems

Fall 2001

Lecture 9: Paging

Geoffrey M. Voelker

Lecture Overview

Today we'll cover more paging mechanisms:

- Optimizations
 - ◆ Managing page tables (space)
 - ◆ Efficient translations (TLBs) (time)
 - ◆ Demand paged virtual memory (space)
- Recap address translation
- Advanced Functionality
 - ◆ Sharing memory
 - ◆ Copy on Write
 - ◆ Mapped files

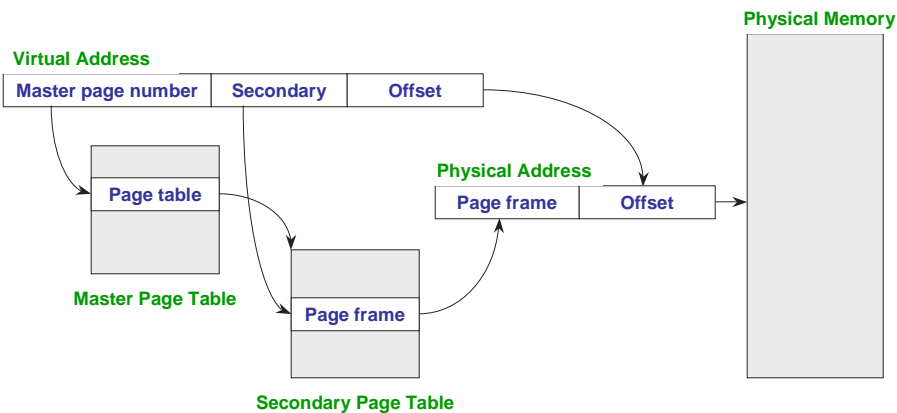
Managing Page Tables

- Last lecture we computed the size of the page table for a 32-bit address space w/ 4K pages to be 4MB
 - This is far far too much overhead for each process
- How can we reduce this overhead?
 - Observation: Only need to map the portion of the address space actually being used (tiny fraction of entire addr space)
- How do we only map what is being used?
 - Can dynamically extend page table...
 - Does not work if addr space is sparce (internal fragmentation)
- Use another level of indirection: two-level page tables

Two-Level Page Tables

- Two-level page tables
 - Virtual addresses (VAs) have three parts:
 - » Master page number, secondary page number, and offset
 - Master page table maps VAs to secondary page table
 - Secondary page table maps page number to physical page
 - Offset indicates where in physical page address is located
- Example
 - 4K pages, 4 bytes/PTE
 - How many bits in offset? $4K = 12$ bits
 - Want master page table in one page: $4K/4$ bytes = 1K entries
 - Hence, 1K secondary page tables. How many bits?
 - Master (1K) = 10, offset = 12, inner = $32 - 10 - 12 = 20$ bits

Two-Level Page Tables



October 31, 2001

CSE 120 – Lecture 9 – Paging

5

Addressing Page Tables

Where do we store page tables (which address space)?

- Physical memory
 - ◆ Easy to address, no translation required
 - ◆ But, allocated page tables consume memory for lifetime of VAS
- Virtual memory (OS virtual address space)
 - ◆ Cold (unused) page table pages can be paged out to disk
 - ◆ But, addressing page tables requires translation
 - ◆ How do we stop recursion?
 - ◆ Do not page the outer page table (called **wiring**)
- If we're going to page the page tables, might as well page the entire OS address space, too
 - ◆ Need to wire special code and data (fault, interrupt handlers)

October 31, 2001

CSE 120 – Lecture 9 – Paging

6

Efficient Translations

- Our original page table scheme already doubled the cost of doing memory lookups
 - ♦ One lookup into the page table, another to fetch the data
- Now two-level page tables triple the cost!
 - ♦ Two lookups into the page tables, a third to fetch the data
 - ♦ And this assumes the page table is in memory
- How can we use paging but also have lookups cost about the same as fetching from memory?
 - ♦ Cache translations in hardware
 - ♦ Translation Lookaside Buffer (TLB)
 - ♦ TLB managed by Memory Management Unit (MMU)

October 31, 2001

CSE 120 – Lecture 9 – Paging

7

TLBs

- Translation Lookaside Buffers
 - ♦ Translate **virtual page #s** into **PTEs** (not physical addrs)
 - ♦ Can be done in a single machine cycle
- TLBs implemented in hardware
 - ♦ Fully associative cache (all entries looked up in parallel)
 - ♦ Cache tags are virtual page numbers
 - ♦ Cache values are PTEs (entries from page tables)
 - ♦ With PTE + offset, can directly calculate physical address
- TLBs exploit locality
 - ♦ Processes only use a handful of pages at a time
 - » 16-48 entries/pages (64-192K)
 - » Only need those pages to be “mapped”
 - ♦ Hit rates are therefore very important

October 31, 2001

CSE 120 – Lecture 9 – Paging

8

Managing TLBs

- Address translations for most instructions are handled using the TLB
 - ♦ >99% of translations, but there are misses (TLB miss)...
- Who places translations into the TLB (loads the TLB)?
 - ♦ Hardware (Memory Management Unit)
 - » Knows where page tables are in main memory
 - » OS maintains tables, HW accesses them directly
 - » Tables have to be in HW-defined format (inflexible)
 - ♦ Software loaded TLB (OS)
 - » TLB faults to the OS, OS finds appropriate PTE, loads it in TLB
 - » Must be fast (but still 20-200 cycles)
 - » CPU ISA has instructions for manipulating TLB
 - » Tables can be in any format convenient for OS (flexible)

Managing TLBs (2)

- OS ensures that TLB and page tables are consistent
 - ♦ When it changes the protection bits of a PTE, it needs to invalidate the PTE if it is in the TLB
- Reload TLB on a process context switch
 - ♦ Invalidate all entries
 - ♦ Why? What is one way to fix it?
- When the TLB misses and a new PTE has to be loaded, a cached PTE must be evicted
 - ♦ Choosing PTE to evict is called the TLB replacement policy
 - ♦ Implemented in hardware, often simple (e.g., Last-Not-Used)

Paged Virtual Memory

- We've mentioned before that pages can be moved between memory and disk
 - This process is called **demand paging**
- OS uses main memory as a page cache of all the data allocated by processes in the system
 - Initially, pages are allocated from memory
 - When memory fills up, allocating a page in memory requires some other page to be evicted from memory
 - » Why physical memory pages are called "frames"
 - Evicted pages go to disk (where? the swap file)
 - The movement of pages between memory and disk is done by the OS, and is transparent to the application

October 31, 2001

CSE 120 – Lecture 9 – Paging

11

Page Faults

- What happens when a process accesses a page that has been evicted?
 1. When it evicts a page, the OS sets the PTE as invalid and stores the location of the page in the swap file in the PTE
 2. When a process accesses the page, the invalid PTE will cause a trap (**page fault**)
 3. The trap will run the OS page fault handler
 4. Handler uses the invalid PTE to locate page in swap file
 5. Reads page into a physical frame, updates PTE to point to it
 6. Restarts process
- But where does it put it? Have to evict something else
 - OS usually keeps a pool of free pages around so that allocations do not always cause evictions

October 31, 2001

CSE 120 – Lecture 9 – Paging

12

Address Translation Redux

- We started this topic with the high-level problem of translating virtual addresses into physical address
- We've covered all of the pieces
 - Virtual and physical addresses
 - Virtual pages and physical page frames
 - Page tables and page table entries (PTEs), protection
 - TLBs
 - Demand paging
- Now let's put it together, bottom to top

The Common Case

- Situation: Process is executing on the CPU, and it issues a read to an address
 - What kind of address is it? Virtual or physical?
- The read goes to the TLB in the MMU
 1. TLB does a lookup using the **page number** of the address
 2. Common case is that the page number matches, returning a **page table entry (PTE)** for the mapping for this address
 3. TLB validates that the **PTE protection** allows reads (in this example)
 4. PTE specifies which **physical frame** holds the page
 5. MMU combines the physical frame and offset into a **physical address**
 6. MMU then reads from that physical address, returns value to CPU
- Note: **This is all done by the hardware**

TLB Misses

- At this point, two other things can happen
 1. TLB does not have a PTE mapping this virtual address
 2. PTE exists, but memory access violates PTE protection bits
- We'll consider each in turn

Reloading the TLB

- If the TLB does not have mapping, two possibilities:
 - ♦ 1. MMU loads PTE from page table in memory
 - » Hardware managed TLB, OS not involved in this step
 - » OS has already set up the page tables so that the hardware can access it directly
 - ♦ 2. Trap to the OS
 - » Software managed TLB, OS intervenes at this point
 - » OS does lookup in page table, loads PTE into TLB
 - » OS returns from exception, TLB continues
- A machine will only support one method or the other
- At this point, there is a PTE for the address in the TLB

TLB Misses (2)

Note that:

- Page table lookup (by HW or OS) can cause a recursive fault if page table is paged out
 - ♦ Assuming page tables are in OS virtual address space
 - ♦ Not a problem if tables are in physical memory
 - ♦ Yes, this is a complicated situation
- When TLB has PTE, it restarts translation
 - ♦ Common case is that the PTE refers to a valid page in memory
 - » These faults are handled quickly, just read PTE from the page table in memory and load into TLB
 - ♦ Uncommon case is that TLB faults again on PTE because of PTE protection bits (e.g., page is invalid)
 - » Becomes a page fault...

Page Faults

- PTE can indicate a protection fault
 - ♦ Read/write/execute – operation not permitted on page
 - ♦ Invalid – virtual page not allocated, or page not in physical memory
- TLB traps to the OS (software takes over)
 - ♦ R/W/E – OS usually will send fault back up to process, or might be playing games (e.g., copy on write, mapped files)
 - ♦ Invalid
 - » Virtual page not allocated in address space
 - OS sends fault to process (e.g., segmentation fault)
 - » Page not in physical memory
 - OS allocates frame, reads from disk, maps PTE to physical frame

Advanced Functionality

- Now we're going to look at some advanced functionality that the OS can provide applications using virtual memory tricks
 - ◆ Shared memory
 - ◆ Copy on Write
 - ◆ Mapped files

Sharing

- Private virtual address spaces protect applications from each other
 - ◆ Usually exactly what we want
- But this makes it difficult to share data (have to copy)
 - ◆ Parents and children in a forking Web server or proxy will want to share an in-memory cache without copying
- We can use **shared memory** to allow processes to share data using direct memory references
 - ◆ Both processes see updates to the shared memory segment
 - » Process B can immediately read an update by process A
 - ◆ How are we going to coordinate access to shared data?

Sharing (2)

- How can we implement sharing using page tables?
 - Have PTEs in both tables map to the same physical frame
 - Each PTE can have different protection values
 - Must update both PTEs when page becomes invalid
- Can map shared memory at same or different virtual addresses in each process' address space
 - Different: Flexible (no address space conflicts), but pointers inside the shared memory segment are invalid (Why?)
 - Same: Less flexible, but shared pointers are valid (Why?)
- What happens if a pointer inside the shared segment references an address outside the segment?

October 31, 2001

CSE 120 – Lecture 9 – Paging

21

Copy on Write

- OSES spend a lot of time copying data
 - System call arguments between user/kernel space
 - Entire address spaces to implement fork()
- Use Copy on Write (CoW) to defer large copies as long as possible, hoping to avoid them altogether
 - Instead of copying pages, create **shared mappings** of parent pages in child virtual address space
 - Shared pages are protected as read-only in child
 - » Reads happen as usual
 - » Writes generate a protection fault, trap to OS, copy page, change page mapping in client page table, restart write instruction
 - How does this help fork()? (Implemented as Unix vfork())

October 31, 2001

CSE 120 – Lecture 9 – Paging

22

Mapped Files

- Mapped files enable processes to do file I/O using loads and stores
 - ♦ Instead of “open, read into buffer, operate on buffer, ...”
- Bind a file to a virtual memory region (mmap() in Unix)
 - ♦ PTEs map virtual addresses to physical frames holding file data
 - ♦ Virtual address `base + N` refers to offset `N` in file
- Initially, all pages mapped to file are invalid
 - ♦ OS reads a page from file when invalid page is accessed
 - ♦ OS writes a page to file when evicted, or region unmapped
 - ♦ If page is not dirty (has not been written to), no write needed
 - » Another use of the dirty bit in PTE

October 31, 2001

CSE 120 – Lecture 9 – Paging

23

Mapped Files (2)

- File is essentially backing store for that region of the virtual address space (instead of using the swap file)
 - ♦ Virtual address space not backed by “real” files also called **Anonymous VM**
- Advantages
 - ♦ Uniform access for files and memory (just use pointers)
 - ♦ Less copying
- Drawbacks
 - ♦ Process has less control over data movement
 - » OS handles faults transparently
 - ♦ Does not generalize to streamed I/O (pipes, sockets, etc.)

October 31, 2001

CSE 120 – Lecture 9 – Paging

24

Summary

Paging mechanisms:

- Optimizations
 - ◆ Managing page tables (space)
 - ◆ Efficient translations (TLBs) (time)
 - ◆ Demand paged virtual memory (space)
- Recap address translation
- Advanced Functionality
 - ◆ Sharing memory
 - ◆ Copy on Write
 - ◆ Mapped files

Next time: Paging policies

Next time...

- Read Chapter 4.4-4.6