Lecture 12: Demand Paging

CSE 120: Principles of Operating Systems
Alex C. Snoeren

HW 3 Due 11/9
Complete Address Translation

- 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
  - Translation lookaside buffers (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
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: Exceptional Cases

- 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?
Page-Level Sharing

- 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?
Read Sharing

Virtual Address

Physical Address

Page X Offset

Frame A Offset

Frame A Offset

P0 Page Table

P1 Page Table

CSE 120 – Lecture 12: Demand Paging
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())
CoW: Read Sharing to Start
CoW: Exclusive Page on Write

Virtual Address

Page X

Offset

Frame B

Offset

Physical Address

Frame B

P0 Page Table

Physical Memory

Frame A

Frame B

Frame A

Frame A

Page Y

Offset

Virtual Address

Physical Address

P1 Page Table

Frame A

1 0
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
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.)
Paging Summary

- Optimizations
  - Managing page tables (space)
  - Efficient translations (TLBs) (time)
  - Demand paged virtual memory (space)

- Advanced Functionality
  - Sharing memory
  - Copy on Write
  - Mapped files
Locality

- All paging schemes depend on locality
  - Processes reference pages in localized patterns
- Temporal locality
  - Locations referenced recently likely to be referenced again
- Spatial locality
  - Locations near recently referenced locations are likely to be referenced soon
- Although the cost of paging is high, if it is infrequent enough it is acceptable
  - Processes usually exhibit both kinds of locality during their execution, making paging practical
Demand Paging (OS)

- Recall demand paging from the OS perspective:
  - Pages are evicted to disk when memory is full
  - Pages loaded from disk when referenced again
  - References to evicted pages cause a TLB miss
    - PTE was invalid, causes fault
  - OS allocates a page frame, reads page from disk
  - When I/O completes, the OS fills in PTE, marks it valid, and restarts faulting process

- Dirty vs. clean pages
  - Actually, only dirty pages (modified) need to be written to disk
  - Clean pages do not – but you need to know where on disk to read them from again
Demand Paging (Process)

- Demand paging is also used when a process first starts up
- When a process is created, it has
  - A brand new page table with all valid bits off
  - No pages in memory
- When the process starts executing
  - Instructions fault on code and data pages
  - Faulting stops when all necessary code and data pages are in memory
  - Only code and data needed by a process needs to be loaded
  - This, of course, changes over time…
Page Replacement

- When a page fault occurs, the OS loads the faulted page from disk into a page frame of memory.
- At some point, the process has used all of the page frames it is allowed to use.
  - This is likely less than all of available memory.
- When this happens, the OS must replace a page for each page faulted in.
  - It must evict a page to free up a page frame.
- The page replacement algorithm determines how this is done.
  - And they come in all shapes and sizes.
Evicting the Best Page

- The goal of the replacement algorithm is to reduce the fault rate by selecting the best victim page to remove.
- The best page to evict is the one never touched again:
  - Will never fault on it.
- Never is a long time, so picking the page closest to “never” is the next best thing:
  - Evicting the page that won’t be used for the longest period of time minimizes the number of page faults.
  - Proved by Belady.
- We’re now going to survey various replacement algorithms, starting with Belady’s.
Belady’s Algorithm

- Belady’s algorithm is known as the optimal page replacement algorithm because it has the lowest fault rate for any page reference stream
  - Idea: Replace the page that will not be used for the longest time in the future
  - Problem: Have to predict the future

- Why is Belady’s useful then? Use it as a yardstick
  - Compare implementations of page replacement algorithms with the optimal to gauge room for improvement
  - If optimal is not much better, then algorithm is pretty good
  - If optimal is much better, then algorithm could use some work
    » Random replacement is often the lower bound
First-In First-Out (FIFO)

- FIFO is an obvious algorithm and simple to implement
  - Maintain a list of pages in order in which they were paged in
  - On replacement, evict the one brought in longest time ago

- Why might this be good?
  - Maybe the one brought in the longest ago is not being used

- Why might this be bad?
  - Then again, maybe it’s not
  - We don’t have any info to say one way or the other

- FIFO suffers from “Belady’s Anomaly”
  - The fault rate might actually increase when the algorithm is given more memory (very bad)
Belady’s Anomaly w/FIFO

Page References

|   | 1 | 1 | 3 | 3 | 1 | 1 | 5 | 5 | 2 | 2 | 4 | 4 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 4 | 4 | 5 |   | 5 |   |   |   |   | 9  |
| 2 | 2 | 2 | 2 | 1 | 1 |   | 2 |   |   | 2 | 4 |   |
| 3 | 3 | 3 | 3 | 2 |   |   | 2 |   |   | 4 | 4 | 10 |
| 4 | 1 | 1 | 1 | 5 | 5 | 5 | 5 | 4 | 4 | 4 | 3 |   |
| 5 | 2 | 2 | 2 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 3 |   |

CSE 120 – Lecture 12: Demand Paging
Least Recently Used (LRU)

- LRU uses reference information to make a more informed replacement decision
  - Idea: We can’t predict the future, but we can make a guess based upon past experience
  - On replacement, evict the page that has not been used for the longest time in the past (Belady’s: future)
  - When does LRU do well? When does LRU do poorly?

- Implementation
  - To be perfect, need to time stamp every reference (or maintain a stack) – much too costly
  - So we need to approximate it
Approximating LRU

- LRU approximations use the PTE reference bit
  - Keep a counter for each page
  - At regular intervals, for every page do:
    » If ref bit = 0, increment counter
    » If ref bit = 1, zero the counter
    » Zero the reference bit
  - The counter will contain the number of intervals since the last reference to the page
  - The page with the largest counter is the least recently used

- Some architectures don’t have a reference bit
  - Can simulate reference bit using the valid bit to induce faults
  - What happens when we make a page invalid?
LRU Clock

- Not Recently Used (NRU) – Used by Unix
  - Replace page that is “old enough”
  - Arrange all of physical page frames in a big circle (clock)
  - A clock hand is used to select a good LRU candidate
    - Sweep through the pages in circular order like a clock
    - If the ref bit is off, it hasn’t been used recently
      - What is the minimum “age” if ref bit is off?
    - If the ref bit is on, turn it off and go to next page
  - Arm moves quickly when pages are needed
  - Low overhead when plenty of memory
  - If memory is large, “accuracy” of information degrades
    - Use additional hands
In a multiprogramming system, we need a way to allocate memory to competing processes.

Problem: How to determine how much memory to give to each process?

- Fixed space algorithms
  - Each process is given a limit of pages it can use
  - When it reaches the limit, it replaces from its own pages
  - Local replacement
    - Some processes may do well while others suffer

- Variable space algorithms
  - Process’ set of pages grows and shrinks dynamically
  - Global replacement
    - One process can ruin it for the rest
Working Set Model

- A working set of a process is used to model the dynamic locality of its memory usage
  - Defined by Peter Denning in 60s

- Definition
  - \( \text{WS}(t,w) = \{ \text{pages P such that P was referenced in the time interval (t, t-w)} \} \)
  - \( t \) – time, \( w \) – working set window (measured in page refs)

- A page is in the working set (WS) only if it was referenced in the last \( w \) references
Working Set Size

- The working set size is the number of pages in the working set
  - The number of pages referenced in the interval \((t, t-w)\)

- The working set size changes with program locality
  - During periods of poor locality, you reference more pages
  - Within that period of time, the working set size is larger

- Intuitively, want the working set to be the set of pages a process needs in memory to prevent heavy faulting
  - Each process has a parameter \(w\) that determines a working set with few faults
  - Denning: Don’t run a process unless working set is in memory
Working Set Problems

- Problems
  - How do we determine w?
  - How do we know when the working set changes?

- Too hard to answer
  - So, working set is not used in practice as a page replacement algorithm

- However, it is still used as an abstraction
  - The intuition is still valid
  - When people ask, “How much memory does Netscape need?”, they are in effect asking for the size of Netscape’s working set
Page Fault Frequency (PFF)

- Page Fault Frequency (PFF) is a variable space algorithm that uses a more ad-hoc approach
  - Monitor the fault rate for each process
  - If the fault rate is above a high threshold, give it more memory
    » So that it faults less
    » But not always (FIFO, Belady’s Anomaly)
  - If the fault rate is below a low threshold, take away memory
    » Should fault more
    » But not always

- Hard to use PFF to distinguish between changes in locality and changes in size of working set
Thrashing

- Page replacement algorithms avoid **thrashing**
  - When most of the time is spent by the OS in paging data back and forth from disk
  - No time spent doing useful work (making progress)
  - In this situation, the system is **overcommitted**
    » No idea which pages should be in memory to reduce faults
    » Could just be that there isn’t enough physical memory for all of the processes in the system
    » Ex: Running Windows XP with 64 MB of memory…

- Possible solutions
  » Swapping – write out all pages of a process
  » Buy more memory
CSE 120 – Lecture 12: Demand Paging

Summary

- Page replacement algorithms
  - Belady’s – optimal replacement (minimum # of faults)
  - FIFO – replace page loaded furthest in past
  - LRU – replace page referenced furthest in past
    » Approximate using PTE reference bit
  - LRU Clock – replace page that is “old enough”
  - Working Set – keep the set of pages in memory that has minimal fault rate (the “working set”)
  - Page Fault Frequency – grow/shrink page set as a function of fault rate

- Multiprogramming
  - Should a process replace its own page, or that of another?
Next time...

- New topic: Filesystems
- Read Chapters 9, 10