CSC 369

Week 8: Paging Design & Implementation

University of Toronto, Department of Computer Science

Agenda

- Recap of full address translation
- Other implementation concerns
- Advanced Functionality

Thrashing

- Page replacement algorithms should avoid thrashing
  - When more time is spent by the OS in paging data back and forth from disk than executing user programs
  - No time spent doing useful work (making progress)
  - In this situation, the system is overcommitted/oversubscribed
    - 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 Windows10 with 4MB of memory…
  - Possible solutions
    - Swapping – write out all pages of a process and suspend it
    - OOM Killer daemon
    - Buy more memory

CPU utilization

- What percentage of time is the CPU busy
  - Mostly I/O heavy processes – CPU utilization is low
  - Compute-intensive processes – CPU utilization is high
- Say CPU utilization is low. Is it a good idea to increase the degree of multiprogramming, to better utilize the idle CPU cycles?
  - Nope! In fact typically decreases CPU utilization
  - Less memory is available to each program => higher page fault likelyhood
- What about decreasing?

Page Buffering

- Previously, we assumed the replacement algorithm is run and a victim page selected *when a new page needs to be brought in*
- Most of these algorithms are too costly to run on every page fault
  - Maintain a pool of free pages (free page list)
  - Run replacement algorithm when pool becomes too small (“low water mark”), free enough pages to at once replenish pool (“high water mark”)
    - Uses dedicated kernel thread, *the paging daemon*
  - On page fault, grab a frame from the free list
  - Frames on free list still hold previous contents, can be “rescued” if virtual page is referenced before reallocation

Addressing Page Tables

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

- Physical memory
  - Easy to address, no translation required (or very simple translation, like linear offset)
  - allocated page tables consume memory for lifetime of Virt. AS
- Virtual memory (OS virtual address space, KSEG2)
  - 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)

Managing Swap Space

- Option 1: Use raw disk partition
- Option 2: Use ordinary large file in file system
- Tradeoffs?

- When should swap be allocated / freed?
  - On process startup / shutdown?
  - On pageout / pagein?

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 does this address typically contain?
  - What kind of address is it? Virtual or physical?

Read Access (Load)

The read address 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
4. PTE specifies which physical frame holds the page
5. MMU combines physical frame & offset into a physical address
6. MMU reads from that physical addr, returns value to CPU

- Note: This is all done by the hardware

TLB Misses

- At steps 2 or 3, two other things can happen
  1. TLB does not have a PTE mapping for this virtual address
  2. PTE exists, but memory access violates PTE protection bits
- We'll consider each in turn

1. 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 OS
     - Software managed TLB
     - OS does lookup in page table, loads PTE into TLB
     - Return from exception, retry memory access
  - Note: Most machines will only support one method or the other
- Now there is a PTE for the address in the TLB
  - Known as a *minor page fault* (no I/O needed)

2. Access not permitted by PTE

- 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 may send fault back up to process, or might be using protection for other purposes (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 (known as *major page fault*)
      - OS allocates frame and reads it in

Check page faults

- Example, on CDF:
  - Number of major and minor page faults on a Linux system (all processes!)
    - `ps -eo min_flt,maj_flt,cmd`
  - Check the page faults generated by a specific program during its execution, e.g.:
    - `/usr/bin/time -v firefox`

    when program terminates, see the stats (Major/Minor page faults)

Next up

- Other implementation concerns
  - How much memory should each process get?
- Advanced Functionality
  - Shared memory
  - Copy-on-write
  - Memory mapped files

Fixed vs. Variable Space Allocation

- 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
    - Local replacement – replacement, set size are separate for each process

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
  - WS(t,Δ) = {pages P such that P was referenced in the time interval (t, t-Δ)}
  - t = time, Δ = working set window (measured in page refs)
- A page is in the working set (WS) only if it was referenced in the last Δ references

…2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3

Δ Δ

WS(t1) = {1,2,5,6,7} t1

WS(t2) = {1,3,4} t2

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-Δ)
- 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 Δ 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 Δ?
  - 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 Firefox need?”, they are in effect asking for the size of Firefox’s working set
- Approximations may be useful in practice

Advanced Functionality

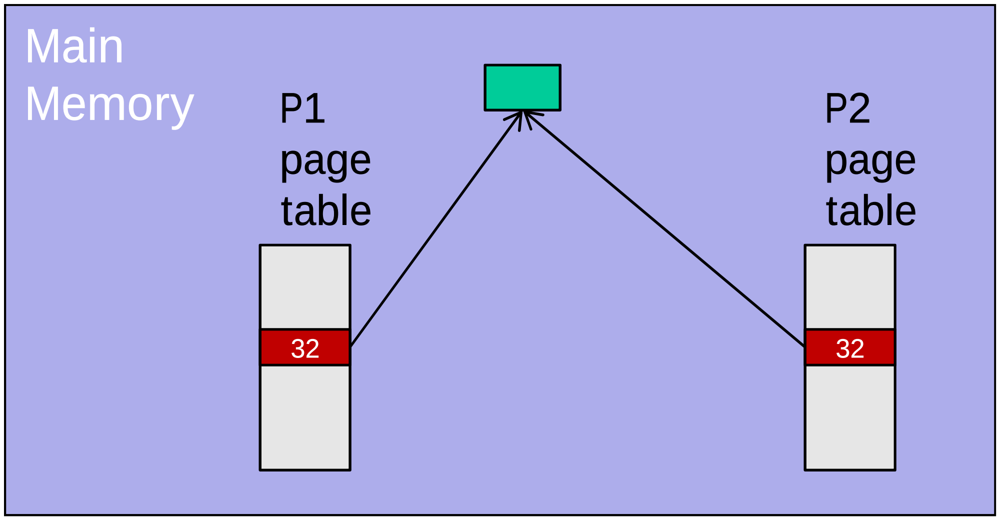
- Some advanced functionality that the OS can provide applications using virtual memory tricks
  - Shared memory
  - Copy on Write

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)

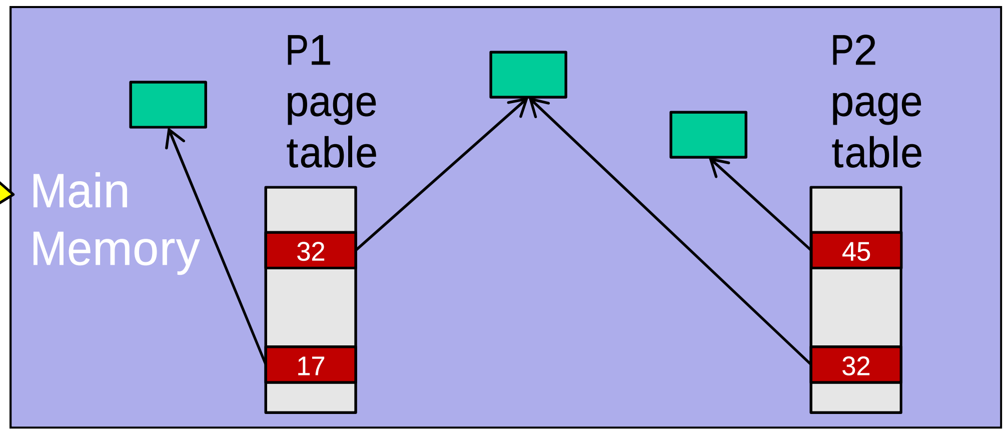
- How can we implement sharing using page tables?
  - Have PTEs in both tables map to the same physical frame
  - Each PTE can have different protection values
  - Must update both PTEs when page becomes invalid



Mapping Shared Regions

- 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?)

Shared memory mapped at different virtual addresses



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())

Memory mapped files

- Mapped files enable processes to do file I/O using loads and stores
  - Instead of “open file, 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
  - => Can read or write at various offsets in file, using memory operations
- 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

Memory mapped files

- 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.)

Summary

Paging mechanisms:

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

Paging policies

Advanced Functionality

- Sharing memory
- Copy on Write
- Memory mapped files

Example Paging Systems

- Next few slides cover Windows and Linux VM *very* briefly
- For fun/extra knowledge only – not on exam

Windows XP Virtual Memory

- 4KB page size on IA32 processors
  - 8 kB on the IA64
- 4GB virtual address space, upper 2 GB used by XP in kernel mode
- Multi-level page table
  - Page directory contains 1024 page directory entries (PDE) of size 4 bytes
  - PDEs point to page tables containing 1024 page table entries (PTEs) of size 4 bytes
- Page frames are tracked using a “page frame database” with one entry per page of physical memory; entry points to PTE which points to frame

Windows XP Paging Policy

- Local page replacement
  - Per-process FIFO
  - Pages are stolen from processes using more than their minimum working set
  - Processes start with a default of 50 pages
  - XP monitors page fault rate and adjusts working-set size accordingly
  - On page fault, *cluster* of pages around the missing page are brought into memory

Linux Paging

- Global replacement, like most Unix
- Modified second-chance clock algorithm
  - Pages *age* with each pass of the clock hand
  - Pages that are not used for a long time will eventually have a value of zero
- Continually under development

Some A3 tips

- Main goal: implement 2-level page tables and demand paging
- We use a simulator to simulate the physical memory
  - 1. We generate memory access traces of some programs (like in Exercise 8)
  - 2. We go through the trace of virtual addresses, translate each of them into a physical address, using 2-level page table and demand paging (swapping)
  - 3. Implement different page replacement policies
  - 4. Analyze memory access patterns

More A3 tips

- 32 vs 64-bit traces (TRACE_64 macro)

64-bit

28 bits unused	12 bits PGDIR_INDEX (4K PDEs)	12 bits PGTBL_INDEX (4K PTEs per tbl)	12 bits OFFSET (4K page size)

PGDIR_SHIFT = 24 bit =>
PGDIR_INDEX = VADDR >> 24

32-bit

10 bits PGDIR_INDEX (1K PDEs)	10 bits PGTBL_INDEX (1K PTEs per tbl)	12 bits OFFSET (4K page size)

PGDIR_SHIFT = 22 bit =>
PGDIR_INDEX = VADDR >> 22

Your code should be the same regardless of 32/64-bit. Do not use hard-coded numbers, use the macros provided in "sim.h"!