CSS503A — Lecture #14 — 2019-06-03

Administrivia

- 2016 final exam posted
- boxes: single source file: pros/cons?
- linked in policy: former students

- tonight: penultimate lecture
  - last new material
  - Wednesday: review/questions (plus tiny amount of new stuff if we don't get through everything today)

- Schedule: final exam originally scheduled Wednesday
  - ⇒ mistake: said "Monday" last week
  - but: other final on Wednesday
  - ⇒ unless someone objects, we'll make it Monday
  - watch canvas for announcements (if room change is required)

- Ass 3
  - drop dead due date: Friday (will try to grade Sat)
  - it is a design error to print error messages in a library — return error values / throw exception & let client code decide how to handle error

- Ass 4
  - drop dead due date: next Friday
  - will grade final 1st

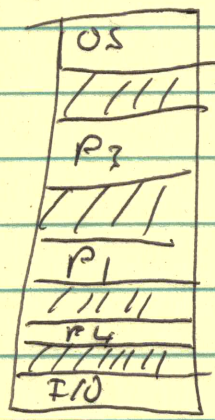
- Final: no news yet
  - will discuss on Wednesday
  - will try to get problem set

Previously on CSS 503...

- modern architecture: hierarchical memory
  - L1, L2, L3 (cache)
  - main memory
  - disk

- memory-mapped I/O

- non-VM system: allocation of main memory to processes



Main Memory (cont.)

- Problem: Programs loaded in memory change from run to run

- Solution: re-link at program load time (linker - loader)

Solution 2: Position-independent code

Solution 3: Hardware support

Segment + $\begin{cases} \text{limit} \\ \text{extent} \end{cases}$ registers

- kernel must reload segment registers on context switch

- still have the problem of external fragmentation

Simple Memory Management Unit (MMU)

- base register specifies starting point
  limit register specifies max address
  - process (virtual) address: start at 0
    add base address to get real address

  ⇒ all this happens in hardware

Memory Allocation

- Contiguous
  - Remarkably easy
  - extremely fragmentable
  - Cannot share memory

- Segmentation
  - multiple contiguous blocks

- Paging
  - Page faults
  - requires multiple memory writes

- fixed-size vs variable-size partition
  - problem: can't handle jumbo process

- variable-size partition
  - problem: memory allocation strategy
    - 1st-fit
    - best fit
    - worst fit
  - external (inter process) fragmentation

=> compaction ? Compaction
=> like defrag util on disk

Segmentation

- different types of data
  - main program
  - procedures / functions
  - library code
  - global variables
  - heap
  - stack

- object file: sections (data structure)

→ use H/W architecture similar to "contiguous", with multiple segment registers
{ base + limit }

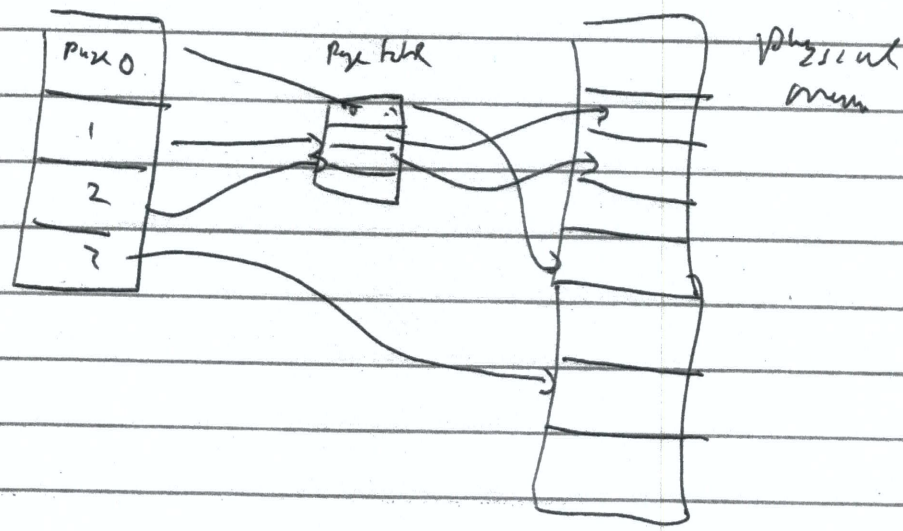
⇒ Segmentation violation

Paging

- divide memory into "frames"
  size 512 Bytes, 1k, 2k, 4k, 8k
  -> size a power of 2
  multiple of disk block size

- logical address space 0 .. N-1
  pages 0 .. n-1
  $n = N / \text{frame size}$

- each process (data structure in kernel) maintains page table mapping virtual pages to physical frames



Address Translation (cont.)

logical address =

page # (e.g. 20 bits)

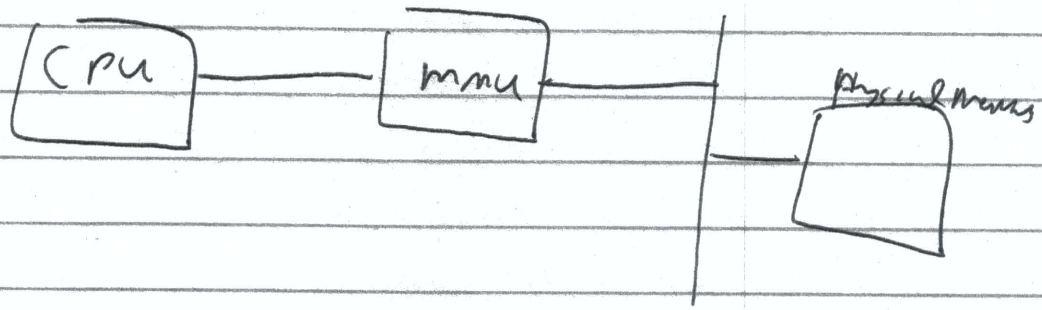
offset within page (e.g. 12 bits)
(displacement)

→ 4 K page size

phys = *(BR + logical >> 12)
| (logical & 0xFFF)

=> done in hardware: very fast

Address Translation (logical → virtual → physical)



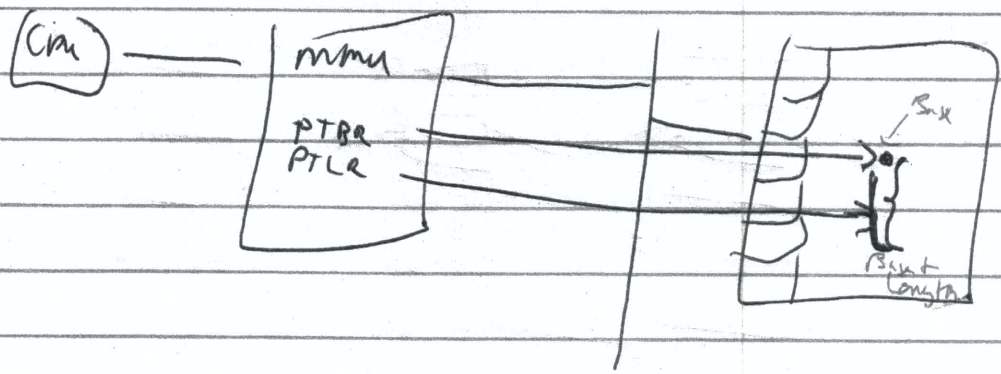
MMU : registers / Operating modes

(Translation mode)

Page table in memory

Base (Page Table Base Register) – physical address

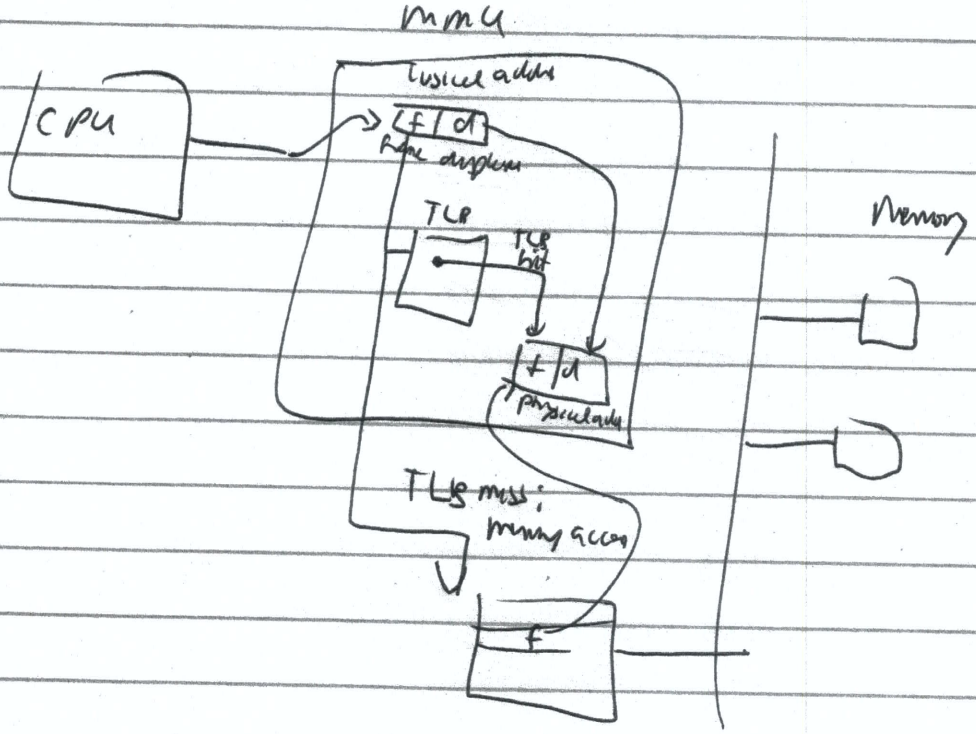
Size (Page Table Length Register)



Paging Hardware with TLB

- problem: loading logical memory value requires 2 memory accesses
  - ~ 200 cycles

- solution: TLB (translation lookaside buffer)

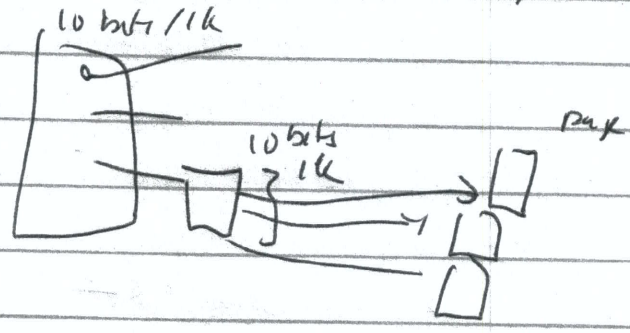


- TLB flush (purge) on context switch

Multi-level page table

- 32-bit architecture
  - 20-bit page
  - 12-bit offset

use 10-bit (1K) top-level super table



- 64-bit machines
  - 3-level table

- tree-like structure with very-high fan-out

- could also hash the table

- effective memory-access time
  TLB hit time * hit rate
  + TLB miss time * (1 - hit rate)

- TLB miss stalls the processor
  (cost of context switch would be higher)

Paging

- logical address space >> (much larger than) physical memory
- many processes running concurrently

- store some pages on disk, load into memory when needed, swapping out unneeded pages to disk
  → context switch (let other process run during lengthy I/O operation)

- internal fragmentation:
  - logical space is not an even multiple of page size
    - last page has unused space

- Minimize fragmentation by using smaller page size
  - but this increases likelihood of page miss due to increased # pages
  => TLB miss (MMU)

Virtual Memory

- Paging architecture:
  page table has valid/invalid flag
  → page fault
- if requested page is not in memory, load page from "swap space" (disk)

- if physical memory is full, find "victim" page & swap it out to disk (write)

- page fault causes a context switch
  - Process moves from run queue to wait queue

- this works because, typically, only a small subset of the program is active at any given time
  init / teardown
  error handling

- shared libraries
- shared memory

Memory Protection

- Page table provides additional info
  - read-only vs read-write
    - code, esp. shared library: read-only
    - constants may be compiled into read-only sections
  - valid / invalid
    - invalid page: not in memory
      "swapped out"
    ⇒ page fault
      ⇒ OS does context switch while disk block read is performed to reload page

- Shared memory: 2 virtual pages share same physical memory
  - read-only (shared library)
  - read-write (data)

- fork: 2 processes share data pages until one page alters data
  "Copy on write"

Clean vs Dirty Pages

- a page that is swapped out & then swapped back in has 2 copies
  - until the program changes the contents

- if the page is to be swapped out again, and has not been changed, i.e. "clean", we can save the cost of a disk write

- if page is "written", page is marked "dirty" and will be written to disk when swapped out

- If we have to swap a page out, the next time we try to use it, it will trigger another page fault
  - swapping out was already caused by a page fault

- memory access : 10^2 - 10^3 cycles
- disk access : 10^7 cycles
* disk : 100,000 times slower

=> paging is no longer tractable

but we still need to understand this because CPU/I/O tradeoffs keep changing, & obsolete techniques do (already here) become relevant again

- even in its heyday, a heavily-loaded system would start "thrashing" - spending all (most of) its time reloading pages & not making forward progress
  - like hashing } works well below max cap
  - like Eth } e.g. 80% capacity

- When thrashing occurs, CPU utilization decreases
- Solution: Suspend some processes

{ RAM is the new ~~disk~~ disk
{ disk is the new tape

Cache: Fast on-chip memory
- Cache aware coding
  1) Step through arrays by row (cache lines)
  2) Step through in 1 pass (if possible)
  3) Prefer arrays of objects over arrays of pointers to objects

Page Replacement Algorithm

- goal: minimize the number of page faults

- optimal algorithm (theoretical / empirical)
  - replace the page that will not be used for the longest period of time into the future
  - problem: this requires an oracle
    ↳ post hoc
- can use optimal results as baseline for evaluating other approaches

=> need a heuristic

First-in / First-out: swap out "oldest" page in memory
- seems fair (?)
=> Belady's Anomaly (see test)
page faults increase as # frame increases
i.e. increasing available memory decreases performance

Page Replacement Algorithm (cont.)

Least-Recently-Used (LRU)
- use doubly-linked list
- when page is used, move to end of list
- requires hardware support

- approximation using reference bits

- Second-chance algorithm
  - circular list
  - if page replacement is required:
    if next candidate ref bit == 0
      swap
    else (reference bit == 1)
      set ref = 0
      advance next-victim pointer
  - if page is used, move it to 1 behind next victim & set ref = 1

- why is LRU heuristic effective?
  ⇒ working set model

- GC destroys the working set

- can determine working set approximation
  - allocate sufficient # pages to a process to manage its fault rate
  - too many: allocate more ~~pages~~ frames
  - too few: reassign ~~pages~~ frames to other process
  - if overall # frames exhausted, de-schedule some process