

Virtual Memory

Dave Eckhardt

de0u@andrew.cmu.edu

Virtual Memory: Motivations

- Avoid fragmentation issues of contiguous segments
- Avoid “final relocation”
- Enable “partial swapping”
- Share memory regions, files efficiently
- Big speed hack for `fork()`

Partial Memory Residence

- Error-handling code not used in every run
- Tables may be allocated larger than used
- Can run *very* large programs
 - Much larger than physical memory
 - As long as “active” footprint fits in RAM
 - Swapping can't do this
- Programs can launch faster
 - Needn't load whole thing

Demand Paging

- RAM is a cache for the set of all pages
- Page tables indicate which pages are resident
 - “**valid**” bit in page table entry (PTE)
 - otherwise, **page fault**

Page fault - Why?

- Address is invalid/illegal
 - Raise exception
- Process is growing stack
- “Cache misses”
 - Page never used
 - Fetch from executable file
 - Page “swapped” to disk
 - Bring it back in!

Page fault story - 1

- Process issues memory reference
 - TLB: miss
 - PT: invalid
- *Trap* to OS kernel!
 - Save registers
 - Load new registers
 - Switch to kernel's page table
 - Run trap handler

Page fault story – 2

- Classify fault address: legal/illegal
- Code/data region of executable?
 - simulate read() into a blank frame
- Heap/modified-data/stack?
 - “somewhere on the paging partition”
 - schedule disk read into blank frame
- Growing stack?
 - Allocate a zero frame, insert into PT

Page fault story – 3

- Put process to sleep (probably)
 - Switch to running another
- Complete I/O, schedule process
- Handle I/O-complete interrupt
 - mark process runnable
- Restore registers, switch page table
 - Faulting instruction re-started transparently
 - May fault more than once!

Demand Paging Performance

- **Effective access time** of memory word
 - $(1 - p_{\text{miss}}) * T_{\text{memory}} + p_{\text{miss}} * T_{\text{disk}}$
- **Textbook example**
 - T_{memory} 100 ns
 - T_{disk} 25 ms
 - $p = 1/1,000$ slows down by factor of 250
 - slowdown of 10% needs $p < 1/2,500,000$

Copy-on-Write

- fork() produces two very-similar processes
 - Same code, data, stack
- Expensive to copy pages
 - Many will never be modified by new process
 - Especially in fork(), exec() case
- *Share* instead of copy?
 - Easy: code pages – read-only
 - Dangerous: stack pages!

Copy-on-Write

- *Simulated* copy
 - Copy page table entries to new process
 - Mark PTEs read-only in old & new
 - Done! (saving factor: 1000)
- Making it real
 - Process writes to page (*oops!*)
 - Page fault handler responsible
 - Copy page into empty frame
 - Mark read-write in both PTEs

Zero pages

- Very special case of copy-on-write
- Many process pages are “blank”
 - All of bss
 - New heap pages
 - New stack pages
- Have one *system-wide* all-zero page
 - Everybody points to it
 - Cloned as needed

Memory-Mapped Files

- Alternate interface
 - `mmap(addr, len, prot, flags, fd, offset)`
 - new memory region presents file contents
 - write-back policy typically unspecified
- Benefits
 - Avoid serializing pointer-based data structures
 - Reads and Writes may be much cheaper
 - Look, Ma, no syscalls!

Memory-Mapped Files

- Implementation
 - Memory region remembers map parameters
 - Page faults trigger read() calls
 - Pages evicted via write() to file
- Shared memory
 - Two processes mmap() “the same way”
 - Pointer to same memory region

Memory Regions vs. Page Tables

- What's a poor page fault handler to do?
 - Kill process?
 - Copy page, mark read-write?
 - Fetch page from file? Which? Where?
- Page Table not a good data structure
 - Format defined by hardware
 - Per-page nature is repetitive
 - Not enough bits to encode OS metadata

Dual-view Memory Model

- Logical
 - Process memory is a list of **regions**
 - “Holes” between regions are *illegal*
 - Per-region methods
 - `fault()`, `evict()`, `unmap()`
- Physical
 - Process memory is a list of **pages**
 - Many “invalid” pages can be made valid
 - Faults delegated to region methods

Page-fault story (for real)

- Examine fault address
- Look up: address -> region
- Call: region->fault(addr, access_mode)
 - *Quickly* fix up problem
 - Or put process to sleep, run scheduler

Page Replacement – When

- Process always want *more* memory frames
 - Explicit deallocation rare
 - Page faults are implicit allocations
- System inevitably runs out
- Solution
 - Pick a frame, store contents to disk
 - Transfer ownership to new process
 - Service fault using this frame

Pick a Frame

- Two-level approach
 - Determine # frames each process “deserves”
 - Process chooses which frame is least-valuable
- System-wide approach
 - Determine globally-least-useful frame

Store Contents to Disk

- Where does it belong?
 - Allocate backing store for each page
 - What if we run out?
- Must we *really* store it?
 - Read-only code/data: no!
 - Can re-fetch from executable
 - Saves space, may be slower
 - Not modified since last page-in: no!
 - Hardware may provide “*page-dirty*” bit

FIFO Page Replacement

- Concept
 - Page queue
 - Page added to queue when created/faulted in
 - Always evict oldest page
- Evaluation
 - Cheap
 - Stupid
 - old page could be startup code
 - FIFO *guaranteed* to evict most-important page!

Optimal Page Replacement

- Concept
 - Evict whichever page will be referenced last
- Evaluation
 - Impossible to implement
- So?
 - Used as upper bound in simulation studies

LRU Page Replacement

- Concept
 - Evict **least-recently-used** page
 - “Past performance *may* not predict future results”
- Evaluation
 - Would work well
 - LRU is computable without fortune teller
 - Bookkeeping *very* expensive
 - Hardware must record every page reference!

Approximating LRU

- Hybrid hardware/software approach
 - 1 **reference** bit per page table entry
 - OS sets reference = 0 for all pages
 - Hardware sets reference=1 when PTE is used
 - OS periodically scans for active pages
- Second-chance algorithm
 - Use FIFO to choose victim page
 - Skip pages with reference == 1

Clock Algorithm

```
static int nextpage = 0;
boolean reference[NPAGES];
int choose_victim() {
    while (reference[nextpage])
        reference[nextpage] = false;
    nextpage = (nextpage+1) % NPAGES;
    return(nextpage);
}
```

Page Buffering

- Maintain a pool of blank pages
 - Page fault handler can be fast
 - Disk write can happen in background
- “page-out daemon”
 - Scan system for dirty pages
 - Write to disk
 - Clear dirty bit
 - Page can be instantly evicted later

“Reclaim” fault

- DEC VAX-11/780 had no reference bit
 - What to page out?
- Approach
 - Store pages to disk according to FIFO
 - Store non-dirty pages in FIFO free-page pool
 - Page fault can “re-claim” page from free-page pool
 - “Yes, I *was* using that page”

Frame Allocation

- How many frames should a process have?
- Minimum
 - Examine worst-case instruction
 - Can multi-byte instruction cross page boundary?
 - Can memory parameter cross page boundary?
 - How many memory parameters?
 - Indirect pointers?

Frame Allocation

- Equal
 - Every process gets same # frames
 - “Fair”
 - Probably wasteful
- Proportional
 - Larger processes get more frames
 - Probably the right approach
 - Encourages greediness

Thrashing

- Problem
 - Process *needs* N pages
 - OS provides $N-1$, $N/2$, etc.
- Result
 - Every page OS evicts generates “immediate” fault
 - More time spent paging than executing
 - Heavy paging slows paging for other processes

Working-Set Model

- Approach
 - Determine necessary # pages
 - If unavailable, start swapping
- How to measure?
 - Periodically scan process reference bits
 - Combine multiple scans (see text)
- Evaluation
 - Expensive

Page-Fault Frequency

- Approach
 - Thrashing == “excessive” paging
 - Adjust each frame quotas to balance fault rates
 - Fault rate “too low”: reduce quota
 - Fault rate “too high”: increase quota
- What if quota increase doesn't help?
 - Start swapping

Program optimizations

- Locality depends on data structures
 - Arrays encourage sequential access
 - Random pointer data structures scatter references
- Compiler & linker can help
 - Don't split a routine across two pages
 - Place helper functions on same page as main routine
- Effects can be *dramatic*

Summary

- Process address space
 - Logical: list of regions
 - Hardware: list of pages
- Fault handler is *complicated*
 - Page-in, copy-on-write, zero-fill, ...
- Understand definition & use of
 - Dirty bit
 - Reference bit