

CSE 120

Summer, 2005

Day 4
Memory
Virtual Memory

Instructor: Neil Rhodes

Memory

For multiprogramming, must have multiple processes

Each process must be in memory in order to execute

Possibilities

- One process in memory. Swap it out to disk, swap in a new one
- Multiple processes in memory, each in their own partition
 - Fixed-size equal partitions

Internal Fragmentation

- Fixed-size non-equal partitions

External Fragmentation

- Variable partitions

2

Placement Algorithms

Need to allocate process of size n

First-fit

- Find first block $\geq n$

Best-fit

- Find smallest open memory block ($\geq n$)

Worst-fit

- Find largest open memory block ($\geq n$)

Next-fit

- Starting with last open block considered, find next block $\geq n$

Buddy system

- Round n up to a power of 2. Free memory is kept in powers of 2. If no block of size n , split a larger chunk in half until size n is found.
- When a block is freed, if its buddy (of the same size) is free, merge them together (recursively)

3

Processes Relocation

Processes may be at different places in memory (or may change when swapped out)

- Protect from other processes
- Refer to this process' data/code/etc.

Solution:

- Relocatable code: all references are relative
 - E.g., on Mac OS 9, relative to A5 register
 - Doesn't help protection
- Hardware
 - Memory Management Unit: convert logical address to physical address
 - Base/limit register

Swapping:

- When too many processes in memory, write one out to disk. Bring it in again later
 - likely to a different location

4

Segmentation

Provides multiple address spaces

- Handy for separate data/stack/code
- Good for sharing code/data between processes
- Easy way to specify protection
 - No execute on stack!
- Address is (segment_number offset within segment)
- Need segment base/segment limit registers
- Programmer (or compiler) must specify different segments

5

Paging

Break up logical address space into pages

Virtual memory: logical address space > physical memory

- Page may not be actually present in memory. Brought in as needed
- Provides illusion of all of logical address space present
- Programmer does nothing.

Break up physical address space into page frames

- Size of a physical page frame = size of page

Need mapping from logical address to physical address

- MMU converts *virtual address* to physical address
 - virtual address has (page number, offset within page)
 - Looks up page number in page table
 - If marked present, converts into page frame. Adds offset to get real address
 - If not marked present, generates page fault

6

Page Table

How page table is read

- MMU reads from dedicated page table
 - Dedicated registers
- MMU reads from main memory
 - Via a pointer to page tables
- Software reads page table
 - We'll look at in more detail in Software TLB Management

Page table must be swapped out on process switch

Page Table Entry

- Caching disabled (needed for memory-mapped I/O)
- Referenced (set by MMU when read from/written to)
- Modified (Dirty) (set by MMU when written to)
- Permissions (Read/Write/Execute)
- Present
- Page Frame number

Page table (or parts of it) may itself be paged out

7

Multilevel Page Table

With large address space (and small pages), page table can be very large

- 32-bit virtual addresses, 4KB page: 2^{20} page table entries

Multilevel page table

- Virtual address broken up into multiple page numbers: PT1 and PT2.
- PT1 used as index into top-level page table to find second-level page table
- PT2 used as index into second-level page table to find page table entry
- If parts of address space are unused, top-level page table can show second-level page table not present.
- Example
 - 32-bit virtual addresses: 10 bits for top-level page number, 10 bits for second-level page number.

8

Translation Lookaside Buffer (TLB)

Cache to map virtual page numbers to page frame

- Associative memory: HW looks up in all cache entries simultaneously
 - Usually not big: 64-128 entries
- TLB entry:
 - page number
 - Valid
 - Modified
 - Protection
 - Page frame
- If not present, do ordinary lookup, then evict entry from TLB and add new one

Cost:

- Direct memory access: 100ns
- Without TLB: 200ns (lookup in Page Table first)
- With TLB
 - Assume cost of TLB lookup is 10ns
 - Assume TLB hit rate is 90%
 - Average cost = $.9 * 10ns + .1 * 200ns = 119ns$

Note that TLB must be flushed on context switch

9

Software TLB Management

MMU doesn't handle page tables; software does

On a TLB miss, generate a TLB fault and let OS deal with it

- Search a larger memory cache. Page containing cache must be in TLB for speed
- If not in cache, searches page table
- Once page frame, etc. found, updates TLB

Why not use hardware?

- Spend die size alternatively
 - Increase Memory cache
 - Reduce cost/power consumption

10

Inverted Page Tables

Traditional page tables: 1 entry/virtual page

Inverted page tables: 1 entry/physical frame of memory

Why?

- 64-bit virtual addresses, 4KB page 256MB of RAM. Inverted page table needs 65536 entries

Entry:

- Process ID
- Virtual page number

Slow to search through table with 65536 entries

- Solution: Hash table. Key is virtual page number. Entry contains virtual page, process ID and page frame

11

Page Fault Handling

MMU generates Page Fault (protection violation or page not present)

- Save registers
- Figure out virtual address that caused fault
 - Often in hardware register
- If protection problem, signal or kill process
- If no free page table entry, evict a page from memory (which one?)
 - If modified, write to backing store (dedicated paging space or normal file)
 - Keep disk location of this page (**not** in page table, but some other data structure).
 - MMU doesn't need to know disk location
 - Suspend faulting process (resume when write is complete)
- Read data from backing store for faulting page
 - From backing store or application code or fill-with-zero
 - Suspend faulting process (resume when read complete)
- Update page table
- Restart instruction for faulting process
 - Must undo any partial effects

12

Segmentation vs. Paging

	Segmentation	Paging
Need the programmer be aware the technique is being used?		
How many linear address spaces are there?		
Can the total address space exceed the size of phys. mem?		
Can procedures and data be distinguished and separately protected?		
Can tables whose size fluctuates be accommodated easily?		
Is sharing of procedures between users facilitated?		
Why was this technique invented?		

13

Page Replacement Algorithms

OPT

- Swap out page that will be used farthest in the future
- Not very practical

Not Recently Used (NRU)

- Use Referenced/Modify bits
- Clear Referenced bits every so often
- Remove from (in order):
 - Not Referenced, not Modified
 - Not Referenced, Modified
 - Referenced, not Modified
 - Referenced, Modified

First-In First-Out (FIFO)

- Low overhead
- Anomaly: adding more page frames can cause more faults

Clock

- Pages in circular list
- Hand points to particular page. When a page is needed, it checks R bit of that page
 - If set, clear and move to next page
 - If not set, replace and move to next page
- Two-handed
 - Forehand clears R bit
 - Backhand looks at R bit. If still clear, eligible to be replaced

14

Page Replacement Algorithms

Least Recently Used (LRU)

- Remove page that has been unused for the longest
- Hardware
 - Keep counter in PTE. Increment on use
- Software (approximation = Not Frequently Used (NFU))
 - On clock interrupt, for each page, set counter = counter >> 1. If R bit is set for page, set top bit of counter and clear R bit.
 - Counters with higher numbers have been accessed more recently (within counter-size * interrupt time).

Working Set

- Program uses subset of pages. Exhibit locality
- Which pages have been used over last k virtual CPU time
- Look at page fault frequency
 - If low for a process, decrease # of pages allocated to process
 - If high for a process increase # of pages

15

Factors for Page Replacement Algorithms

Local/Global Replacement

- Local: only consider pages from same process
- Global: consider pages from any process

Fixed/Variable allocation

- Fixed: each process has certain number of page frames determined in advance
- Variable: can change

Demand-/Pre-Cleaning

- Pre-Cleaning: Write dirty pages out prospectively
- Demand-Cleaning: Write dirty pages out only as needed

Demand Paging

- Load pages only as needed
 - For example, data of code segment loaded only as used

16