Address Translation => Paging

David E. Culler
http://cs162.eecs.berkeley.edu/

Lecture #15
Oct 3, 2014
Virtual Memory Concepts

• Segmentation
  – virtual addressing scheme constructed as a collection of variable sized objects
    » big objects (code, static data, heap, stack)
    » smaller objects (???)
  – addresses of the form <seg id><offset>
  – are translated into
    » a physical memory address (holding the data),
    » an address translation fault, or
    » a violation (seg fault) due to range or mode
  – by indexing into a segment table for STE
    » base : bounds : access bits
  – or through segment registers (ala x86)
Virtual Memory Concepts

• Segmentation
  – virtual addressing scheme constructed as a collection of *variable sized objects*

• Paging
  – virtual addressing scheme in which a flat address space is broken into *fixed size chunks*
  – addresses are of the form `<page#><offset>`
    » no particular semantic content
  – are translated into
    » a physical memory address (holding the data),
    » an address translation fault (page fault), or
    » a violation (seg fault) due to range or mode
  – by indexing into a page table for PTE
    » frame #: access bits
Where does a process live when it is not in memory?
Virtual-Physical Address Translation

Process

virtual address

instruction

address translation mechanism

physical address

Operating System

fault

not present

Disk

Memory
What Mechanism for Translation?

• Segmentation

![Diagram showing segmentation mechanism with instruction, address translation mechanism, seg base + offset, and variable length segment. There is also a red arrow indicating a fault.](image)
What Mechanism for Translation?

- Segmentation
  - Instruction
  - Seg #: Offset
  - Address translation mechanism
  - Fault
  - Seg base + Offset
  - Offset
  - Variable length segment

- Paging
  - Instruction
  - Page #: Offset
  - Address translation mechanism
  - Fault
  - Frame #: Offset
  - Fixed length page
Address Translation Structures

• Segment table
  – ST[seg#] := | base addr | length | flags |
  – VA(s, o) => PA = ST[s].base + o
• Page Table
  – PT[pg#] = | frame # | flags |
  – VA(p : o) => PA = PT[p].frame : o
• Paged Segments
• 2-Level Page Table
• Inverted Page Table
Who does what when?

Process

Virtual address

MMU

physical address

PT

frame#

offset

update PT entry

load page from disk

Page Fault Handler

scheduler

retry

exception

page fault

instruction

Operating System

Page Fault Handler
Issues for address translation mechanism

- Fault occurs if any step along the VA => PA translation cannot complete
  - protection or length violation
  - page or segment not present (non-existent or on disk)
  - internal lookup steps
- Page tables (and segment tables) reside in memory
  - how much memory do they take?
- Virtual address space is (typically) large compared to physical memory space
Bit of historical perspective

• 60’s Multics – Timesharing & Segmentation
• 70’s Unix on PDP-11 16-bit mini computerer
• vax780 32-bit minicomputer => VMS &BSD Unix
  – 32-bit virtual addresses (4 GB), MBs of RAM, ~GB of disk
• <1980 personal computer, i8086
  – 16 bit word size
  – < 640kb physical memory (2^20)
  – segments provided additional 4 bits
    » PA_{20} = SegReg_{16} * 16 + Addr_{16}
• 1982 workstation:
  – MC68000 32/16 bit machine, large (24 bit) PA
  – i80286 16 bit, segment descriptors => seg registers, complex
• mid 80s: 32-bit microprocessor arrives
  – i80386 (segments + paging)
Admin break

• Project
• Slip days
• Pressure Relief Valve
Bit of historical perspective

• vax780 32-bit minicomputer
  – few MBs of RAM (PA ~20+ bits), GB disk, 4 GB VA space

• 16-bit micros

• 32-bit microprocessor arrives
  – i80386 (segments + paging), MC680x0
  – RISC, SPARC, MIPS, M88000
  – 10s MBs of RAM, GBs of disk

• => Mapping GBs of Virt. Address Space requires MBs of RAM for page tables!
  – multi-level translation (page the page table !!!)
Page Table Resources

- MMU hardware performs 2 memory operations for every inst fetch, load, or store
- PT for each process in memory
  - 4 GB VAS / 4 KB page => 1 M PTEs = 4 MB
  - used sparsely
How has OS design choices been influenced by technological change?
two-level page table

- Tree of Page Tables
- Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register
- Valid bits on Page Table Entries
  - Don’t need every 2nd-level table
  - Even when exist, 2nd-level tables can reside on disk if not in use
Example: Two-Level Paging

Virtual memory view

Page Table (level 1)

Page Tables (level 2)

Physical memory view

Page Tables

Virtual memory view
Example: Two-Level Paging

Virtual memory view

Page Table (level 1)

1001 0000 (0x90)

heap

data

code

Page Tables (level 2)

11 11101
10 11100
01 10111
00 10110

Physical memory view

stack 1110 0000

stack 1000 0000 (0x80)

heap

data

code

11 01101
10 01100
01 01011
00 01010

11 00101
10 00100
01 00011
00 00010

In best case, total size of page tables ≈ number of pages used by program. Requires two additional memory access!
Question

• How many memory accesses per fetch, load, or store with 2-level page table?

• Where can a page fault occur?
Multi-level Translation Analysis

• Pros:
  – Only need to allocate as many page table entries as we need for application – size is proportional to usage
    » In other words, sparse address spaces are easy
  – Easy memory allocation
  – Easy Sharing
    » Share at segment or page level (need additional reference counting)

• Cons:
  – One pointer per page (typically 4K – 16K pages today)
  – Page tables need to be contiguous
    » However, previous example keeps tables to exactly one page in size
  – Two (or more, if >2 levels) lookups per reference
    » Seems very expensive!
So how do we make address translation go fast?

- Large memories are slow (larger the slower)
- Fast memories are small
- Really fast storage (registers) are really small
- How do we get a small *average memory access time* for a LARGE memory?
- Harness probability
  - *temporal locality*: recently accessed things likely to be accessed again soon
  - *spatial locality*: things near recently accessed thing are likely to be accessed soon too
- AMAT = P_{hit} \times Time_{hit} + (1-P_{hit}) \times Time_{miss}
- Caching !!!
Where are we depending on caching already?

• When we load a page from disk to memory (page fault)
  • we are likely to access it many times while it is resident
    – ~ 10 ms (0.001 s) to load it
    – @ 1 GHz that is 10 million cycles
• we are likely to access other items in the page
  – 4KB => much larger pages
Translation Look Aside Buffer (TLB)

- TLB holds mapping (page # -> frame #) for recently accessed pages
- on hit, avoid reading PT
- on miss, read PTE into TLB
RAM?

Figure 7  2011 ITRS Product Technology Trends: Memory Product Functions/Chip and Industry Average “Moore’s Law” and Chip Size Trends [unchanged for the 2012 Update]

Source: 2011 ITRS - Executive Summary Fig 7
Costs
How has OS design choices been influenced by technological change?
Bit of historical perspective

• vax780 32-bit minicomputer
  – few MBs of RAM (PA ~20+ bits), GB disk, 4 GB VA space
• 16-bit micros
• mid 80’s 32-bit microprocessor arrives
  – i80386 (segments + paging)
  – RISC, SPARC, MIPS, M8800
  – 10s MBs of RAM, GBs of disk
• => Mapping GBs of Virt. Address Space requires MBs of RAM for page tables!
  – multi-level translation (page the page table !!!)
• ~10 GBs of RAM (!!!) => | VA | < | PA | again
• ~2005 64-bit processors arrive
• | VA | >> | PA |
Inverted Page Table

• With all previous examples (“Forward Page Tables”)
  – Size of page tables is at least as large as amount of virtual memory allocated to ALL processes
  – Physical memory may be much, much less
    » Much of process’ space may be out on disk or not in use

  • Answer: use a hash table
    – Called an “Inverted Page Table”
    – Size is independent of virtual address space
    – Directly related to amount of phy mem (1 entry per phy page)
    – Very attractive option for 64-bit address spaces (IA64, PowerPC, UltraSPARC)

  • Cons: Complexity of managing hash chains in hardware
Summary: Inverted Table

Virtual memory view

Inverted Table
Hash(procID & virt. page #) = phys. page #

Physical memory view

Total size of page table ≈ number of pages used by program in physical memory.

Hash more complex!
<table>
<thead>
<tr>
<th>Address Translation Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Advantages</strong></td>
</tr>
<tr>
<td><strong>Segmentation</strong></td>
</tr>
<tr>
<td><strong>Paging (single-level page)</strong></td>
</tr>
<tr>
<td><strong>Paged segmentation</strong></td>
</tr>
<tr>
<td><strong>Two-level pages</strong></td>
</tr>
<tr>
<td><strong>Inverted Table</strong></td>
</tr>
</tbody>
</table>
Summary of Translation

- Memory is a resource that must be multiplexed
  - Controlled Overlap: only shared when appropriate
  - Translation: Change virtual addresses into physical addresses
  - Protection: Prevent unauthorized sharing of resources

- Simple Protection through segmentation
  - Base + Limit registers restrict memory accessible to user
  - Can be used to translate as well

- Page Tables
  - Memory divided into fixed-sized chunks of memory
  - Offset of virtual address same as physical address

- Multi-Level Tables
  - Virtual address mapped to series of tables
  - Permit sparse population of address space

- Inverted page table: size of page table related to physical memory size
Segments vs Pages

• Segments reflects a design philosophy that hardware capability should closely match software structure.
  – object oriented program => hardware protection of objects => OS management of object placement in the storage hierarchy

• Challenge of segment size
  – large segments => easy translation, memory allocation hard
  – small segments => translation overhead
  ⇒ code, data, stack, heap, shared library (just a few)

• Main value is sharing
  – in a flat address space, where does a shared library go?

• Segments don’t match programming languages well
  – what is the structure of a pointer? seg:offset vs addr
  – is it unique?

• Large flat address space is simpler & empty space facilitates sharing