CS162 - Operating Systems and Systems Programming

Address Translation => Paging

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

Lecture #15

Oct 3, 2014

Reading: A&D 8.1-2, 8.3.1. 9.7
HW 3 out (due 10/13)
Proj 1 Final 10/8, 10/10

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

Memory

Operating System

fault

protection fault

not present

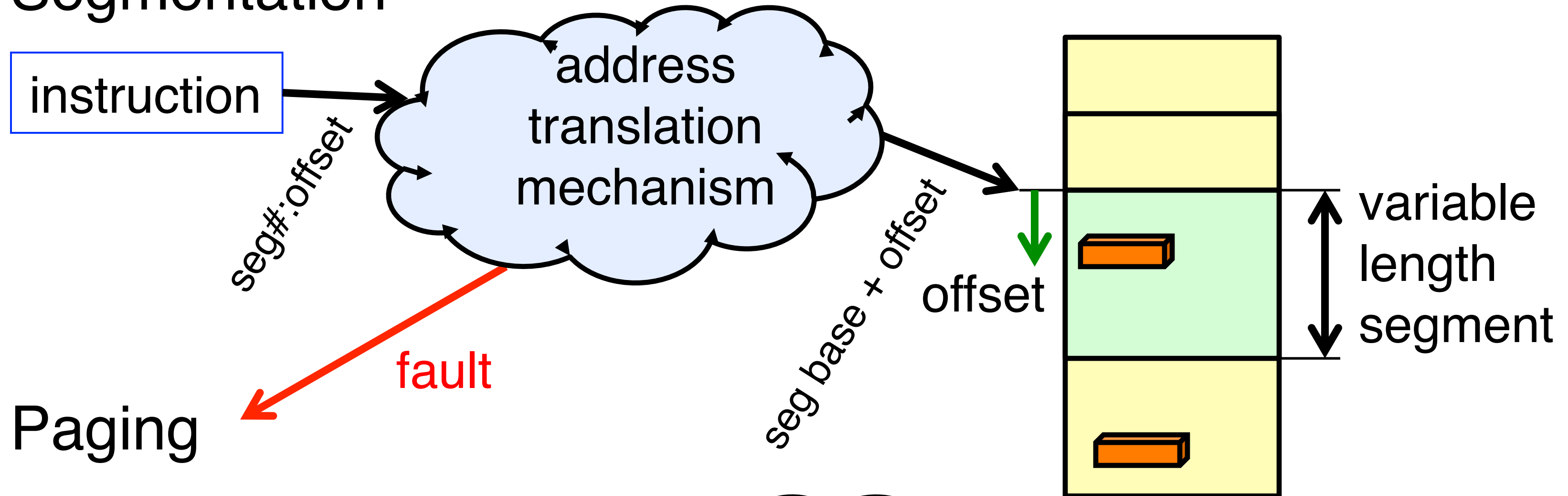
Disk

What Mechanism for Translation?

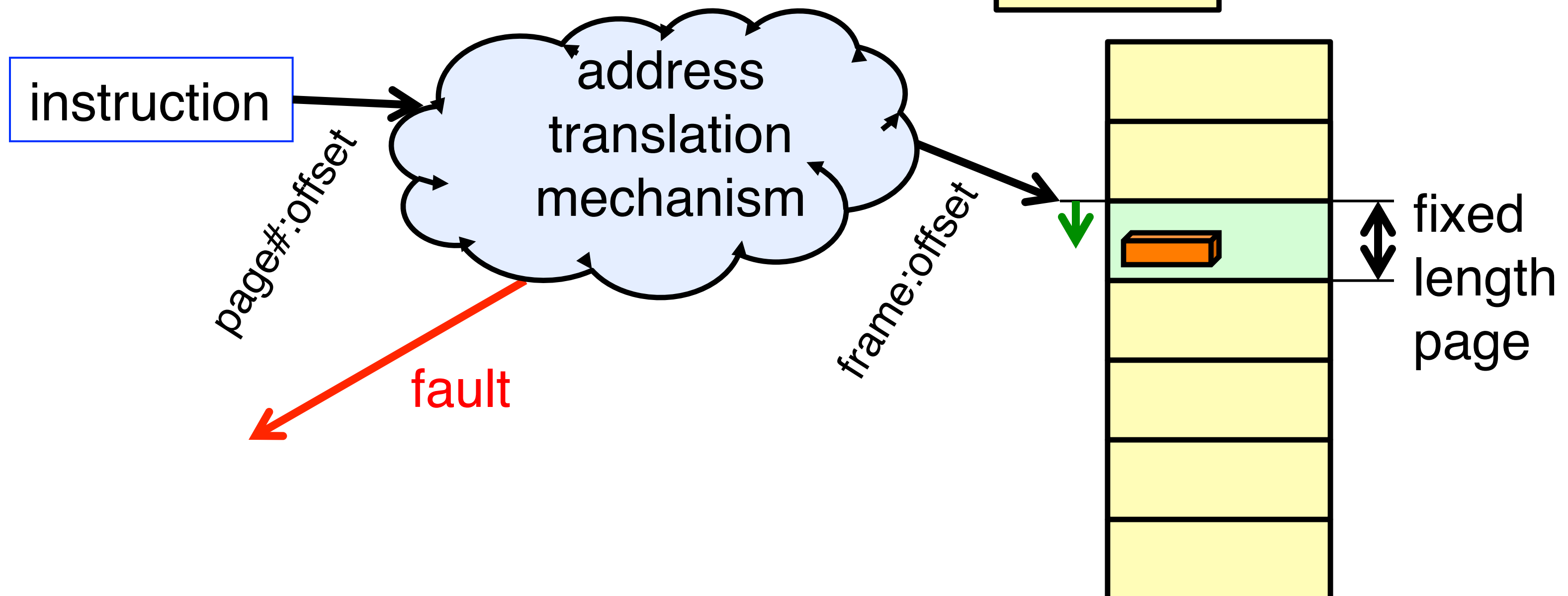
- Segmentation

What Mechanism for Translation?

- Segmentation



- Paging



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 ?

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 to 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 Resouces

- 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?

Transistors

10,000,000,000
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
1,000

Sandy Bridge
G6
Nehalem
Ivy
Pentinum M
Pentium 4
G5
G4
Pentinum III
Petinum MMX
Pentium Pro
Pentium
G3
604e
601
80486
68040
80386
68030
80286
68020
68000
8086
8088
8085
8080
8008
6800
4004

1970 1975 1980 1985 1990 1995 2000 2005 2010 2015

Year

two-level page table

Physical Address: | Physical Page # | Offset |

Virtual Address: | Virtual P1 index (10 bits) | Virtual P2 index (10 bits) | Offset (12 bits) |

PageTablePtr

4KB

→ 4 bytes ←

→ 4 bytes ←

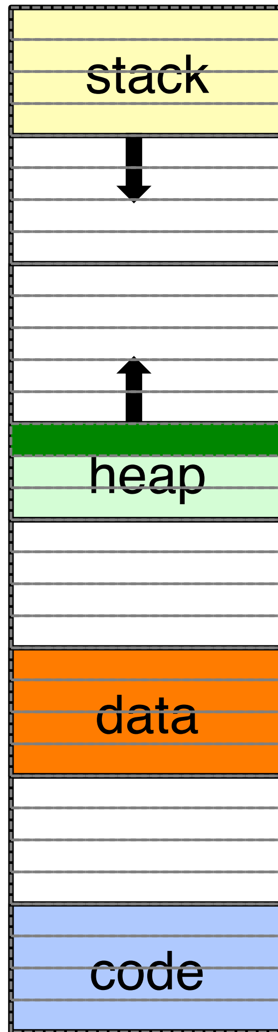
- Tree of Page Tables
- Tables fixed size (1024 entries)
  - On context-switch: save single PageTablePtr register
- Valid bits on Page Table Entries
  - Don't need every 2nd-level table
  - Even when exist, 2nd-level tables can reside on disk if not in use

Example: Two-Level Paging

Example: Two-Level Paging

Virtual memory view

1001 0000
(0x90)



Page Table (level 1)

111	●
110	*null*
101	*null*
100	●
011	*null*
010	●
001	*null*
000	●

Page Tables (level 2)

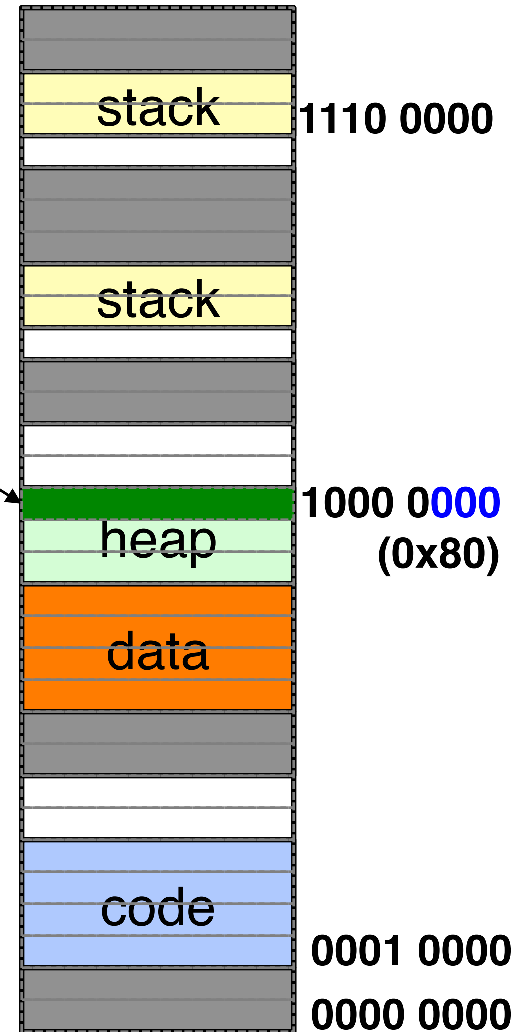
11	11101
10	11100
01	10111
00	10110

11	null
10	10000
01	01111
00	01110

11	01101
10	01100
01	01011
00	01010

11	00101
10	00100
01	00011
00	00010

Physical memory view



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 access things likely to be accessed again soon
  - *spatial locality*: things near recently accessed thing are likely to be accessed soon too
- AMAT = P_{hit} x $Time_{hit}$ + (1-P_{hit}) x $Time_{miss}$
- 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)

Process

virtual address

instruction

MMU

page#

memory

PT

frame#

memory

page fault

TLB

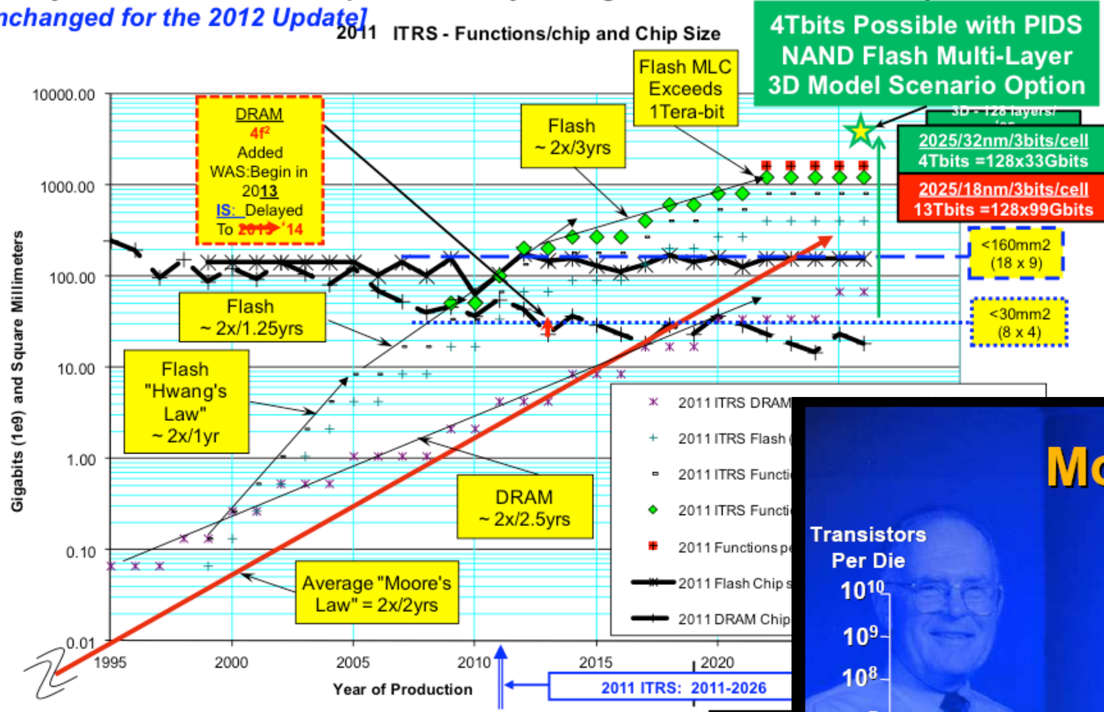
=?

page #	frame #

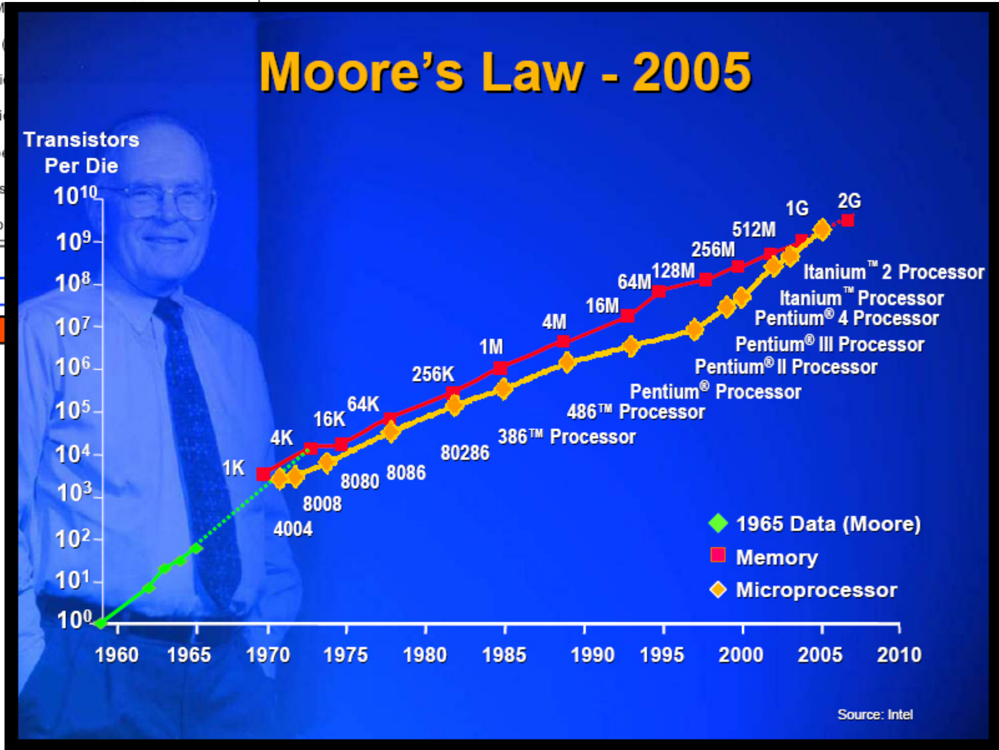
offset

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

RAM?



Source: 2011 ITRS - Executive Summary Fig 7



Costs

How has OS design choices been influenced by technological change?

Transistors

10,000,000,000
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
1,000

Sandy Bridge
G6
Nehalem
Ivy
Pentium 4
Pentinum M
G5
G4
Pentinum III
Petinum MMX
Pentium Pro
Pentium
G3
604e
601
80486
68040
80386
68030
80286
68020
68000
8086
8088
8085
8080
8008
6800
4004

1970 1975 1980 1985 1990 1995 2000 2005 2010 2015

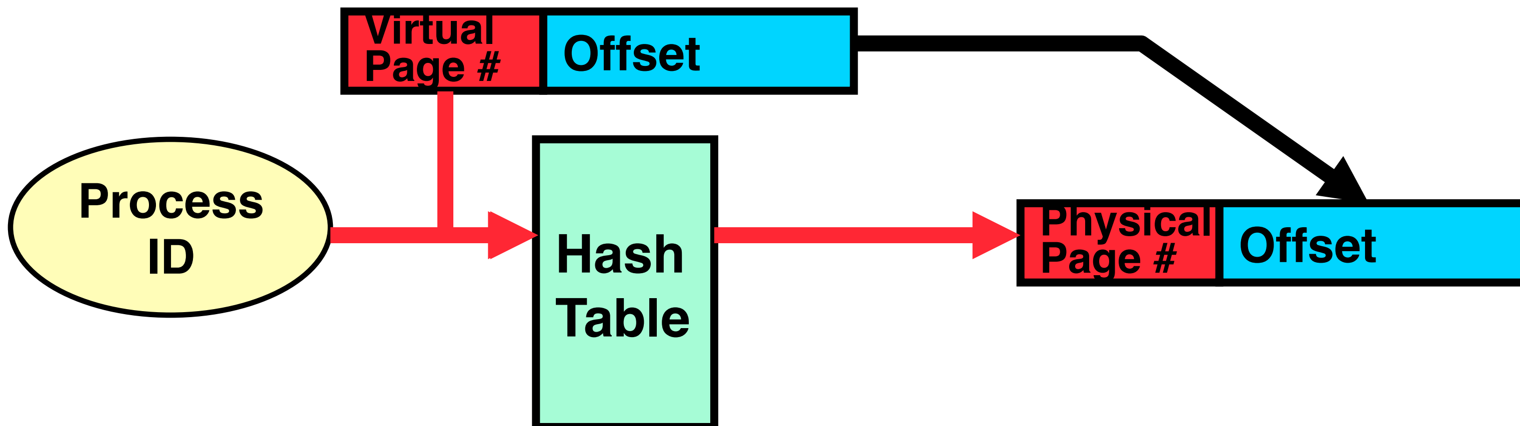
Year

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 (!!!) => I VA I < I PA I again
- ~2005 64-bit processors arrive
- I VA I >> I PA I

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

1111 1111

stack

1110 0000

1100 0000

heap

1000 0000

data

0100 0000

code

0000 0000

page # offset

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

h(11111) = 11101
h(11110) = 11100
h(11101) = 10111
h(11100) = 10110
h(10010)= 10000
h(10001)= 01111
h(10000)= 01110
h(01011)= 01101
h(01010)= 01100
h(01001)= 01011
h(01000)= 01010
h(00011)= 00101
h(00010)= 00100
h(00001)= 00011
h(00000)= 00010

Physical memory view

stack

1110 0000

stack

1011 0000

heap

0111 0000

data

0101 0000

code

0001 0000

0000 0000

Address Translation Comparison

<table>
<tr><th></th><th>Advantages</th><th>Disadvantages</th></tr>
<tr><td>Segmentation</td><td>Fast context switching: Segment mapping maintained by CPU</td><td>External fragmentation</td></tr>
<tr><td>Paging (single-level page)</td><td>No external fragmentation, fast easy allocation</td><td>Large table size ~ virtual memory
Internal fragmentation</td></tr>
<tr><td>Paged segmentation</td><td rowspan="2">Table size ~ # of pages in virtual memory, fast easy allocation</td><td rowspan="2">Multiple memory references per page access</td></tr>
<tr><td>Two-level pages</td></tr>
<tr><td>Inverted Table</td><td>Table size ~ # of pages in physical memory</td><td>Hash function more complex
Aliasing</td></tr>
</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:offest vs addr
  - is it unique?
- Large flat address space is simpler & empty space facilitates sharing