Virtual memory

- Came out of work in late 1960s by Peter Denning
  - Established working set model
  - Led directly to virtual memory
Want processes to co-exist

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>0x9000</td>
</tr>
<tr>
<td>gcc</td>
<td>0x7000</td>
</tr>
<tr>
<td>vim</td>
<td>0x4000</td>
</tr>
<tr>
<td>firefox</td>
<td>0x3000</td>
</tr>
<tr>
<td></td>
<td>0x0000</td>
</tr>
</tbody>
</table>

- **Consider multiprogramming on physical memory**
  - What happens if firefox needs to expand?
  - If firefox needs more memory than is on the machine?
  - If gcc has an error and writes to address 0x7100?
  - When does gcc have to know it will run at 0x4000?
  - What if vim isn’t using its memory?
Issues in sharing physical memory

• Protection
  - A bug in one process can corrupt memory in another
  - Must somehow prevent process A from trashing B’s memory
  - Also prevent A from even observing B’s memory (ssh-agent)

• Transparency
  - A process shouldn’t require particular physical memory bits
  - Yet processes often require large amounts of contiguous memory (for stack, large data structures, etc.)
  - Compilers often want fixed memory layout for performance

• Resource exhaustion
  - Programmers typically assume machine has “enough” memory
  - Sum of sizes of all processes often greater than physical memory
Virtual memory goals

- Give each program its own *virtual* address space
  - At runtime, *Memory-Management Unit* relocates each load/store
  - Application doesn’t see *physical* memory addresses

- Also enforce protection
  - Prevent one app from messing with another’s memory

- And allow programs to see more memory than exists
  - Somehow relocate some memory accesses to disk
Virtual memory goals

- Give each program its own *virtual* address space
  - At runtime, *Memory-Management Unit* relocates each load/store
  - Application doesn’t see *physical* memory addresses

- Also enforce protection
  - Prevent one app from messing with another’s memory

- And allow programs to see more memory than exists
  - Somehow relocate some memory accesses to disk
Virtual memory advantages

- Can re-locate program while running
  - Run partially in memory, partially on disk

- Most of a process’s memory may be idle (80/20 rule).
  - Write idle parts to disk until needed
  - Let other processes use memory of idle part
  - Like CPU virtualization: when process not using CPU, switch
    (Not using a memory region? switch it to another process)

- Challenge: VM = extra layer, could be slow
Software Approach: load-time linking

- **Linker** patches addresses of symbols like `printf`
- Idea: link when process executed, not at compile time
  - Determine where process will reside in memory
  - Adjust all references within program (using addition)
- Problems?
Software Approach: load-time linking

- **Linker** patches addresses of symbols like `printf`
- **Idea**: link when process executed, not at compile time
  - Determine where process will reside in memory
  - Adjust all references within program (using addition)
- **Problems?**
  - How to enforce protection?
  - How to move once already in memory? (consider data pointers)
  - What if no contiguous free region fits program?
Dynamic Relocation: base + bound register

- Two special privileged registers: base and bound
- On each load/store/jump:
  - Physical address = virtual address + base
  - Check $0 \leq$ virtual address < bound, else trap to kernel
- How to move process in memory?
- What happens on context switch?
Dynamic Relocation: base + bound register

- Two special privileged registers: base and bound
- On each load/store/jump:
  - Physical address = virtual address + base
  - Check $0 \leq$ virtual address < bound, else trap to kernel
- How to move process in memory?
  - Change base register
- What happens on context switch?
Dynamic Relocation: base + bound register

- Two special privileged registers: base and bound
- On each load/store/jump:
  - Physical address = virtual address + base
  - Check $0 \leq$ virtual address $<$ bound, else trap to kernel
- How to move process in memory?
  - Change base register
- What happens on context switch?
  - OS must re-load base and bound register

![Diagram showing the relocation process]
Definitions

- Programs load/store to **virtual addresses**
- Actual memory uses **physical addresses**
- VM Hardware is **Memory Management Unit (MMU)**

- Usually part of CPU
- Configured through privileged instructions (e.g., load bound reg)
- Translates from virtual to physical addresses
- Gives per-process view of memory called **address space**
Definitions

- Programs load/store to **virtual addresses**
- Actual memory uses **physical addresses**
- VM Hardware is Memory Management Unit (MMU)

- Usually part of CPU
- Configured through privileged instructions (e.g., load bound reg)
- Translates from virtual to physical addresses
- Gives per-process view of memory called **address space**
Dynamic Relocation: trade-offs

- **Advantages**
  - Cheap in terms of hardware: only two registers
  - Cheap in terms of cycles: do add and compare in parallel
  - Examples: Cray-1 used this scheme

- **Disadvantages**
  - Growing a process is expensive or impossible
  - No way to share code or data (E.g., two copies of vim)
  - One solution: Multiple segments
    - E.g., separate code, stack, data segments
    - Possibly multiple data segments
Dynamic Relocation: trade-offs

- **Advantages**
  - Cheap in terms of hardware: only two registers
  - Cheap in terms of cycles: do add and compare in parallel
  - Examples: Cray-1 used this scheme

- **Disadvantages**
  - Growing a process is expensive or impossible
  - No way to share code or data (E.g., two copies of vim)

- **One solution: Multiple segments**
  - E.g., separate code, stack, data segments
  - Possibly multiple data segments
• Let processes have many base/bound regs
  - Address space built from many segments
  - Can share/protect memory at segment granularity

• Must specify segment as part of virtual address
Each process has a segment table

Each VA indicates a segment and offset:
- Top bits of addr select segment, low bits select offset (PDP-10)
- Or segment selected by instruction or operand (means you need wider “far” pointers to specify segment)
Segmentation example

- 2-bit segment number (1st digit), 12 bit offset (last 3)
  - Where is 0x0240? 0x1108? 0x265c? 0x3002? 0x1600?
**Segmentation trade-offs**

- **Advantages**
  - Multiple segments per process
  - Allows sharing! (how?)
  - Don’t need entire process in memory

- **Disadvantages**
  - Requires translation hardware, which could limit performance
  - Segments not completely transparent to program (e.g., default segment faster or uses shorter instruction)
  - $n$ byte segment needs $n$ contiguous bytes of physical memory
  - Makes fragmentation a real problem.
**Fragmentation**

- **Fragmentation** → Inability to use free memory

- **Over time:**
  - Variable-sized pieces = many small holes (external fragmentation)
  - Fixed-sized pieces = no external holes, but force internal waste (internal fragmentation)
Alternatives to hardware MMU

- Language-level protection (Java)
  - Single address space for different modules
  - Language enforces isolation
  - Singularity OS does this [Hunt]

- Software fault isolation
  - Instrument compiler output
  - Checks before every store operation prevents modules from trashing each other
  - Google Native Client does this
Paging

- Divide memory up into small *pages*
- Map virtual pages to physical pages
  - Each process has separate mapping
- Allow OS to gain control on certain operations
  - Read-only pages trap to OS on write
  - Invalid pages trap to OS on read or write
  - OS can change mapping and resume application
- Other features sometimes found:
  - Hardware can set “accessed” and “dirty” bits
  - Control page execute permission separately from read/write
  - Control caching or memory consistency of page
### Paging trade-offs

- Eliminates external fragmentation
- Simplifies allocation, free, and backing storage (swap)
- Average internal fragmentation of .5 pages per “segment”
• Allocate any physical page to any process
• Can store idle virtual pages on disk
Paging data structures

- Pages are fixed size, e.g., 4K
  - Least significant 12 (\(\log_2 4K\)) bits of address are page offset
  - Most significant bits are page number

- Each process has a page table
  - Maps virtual page numbers (VPNs) to physical page numbers (PPNs)
  - Also includes bits for protection, validity, etc.

- On memory access: Translate VPN to PPN, then add offset
Example: Paging on PDP-11

- **64K virtual memory, 8K pages**
  - Separate address space for instructions & data
  - I.e., can’t read your own instructions with a load

- **Entire page table stored in registers**
  - 8 Instruction page translation registers
  - 8 Data page translations

- **Swap 16 machine registers on each context switch**
x86 Paging

- Paging enabled by bits in a control register (%cr0)
  - Only privileged OS code can manipulate control registers
- Normally 4KB pages
- %cr3: points to 4KB page directory
- Page directory: 1024 PDEs (page directory entries)
  - Each contains physical address of a page table
- Page table: 1024 PTEs (page table entries)
  - Each contains physical address of virtual 4K page
  - Page table covers 4 MB of Virtual mem
- See intel manual for detailed explanation
  - Also volume 2 of AMD64 Architecture docs
  - Also volume 3A of latest Pentium Manual
x86 page translation

Linear Address

Directory Table Offset

31 22 21 12 11 0

Page Directory

Directory Entry

CR3 (PDBR)

Page Table

Page-Table Entry

4-KByte Page

Physical Address

1024 PDE × 1024 PTE = 2²²0 Pages

*32 bits aligned onto a 4-KByte boundary
### x86 page directory entry

**Page-Directory Entry (4-KByte Page Table)**

<p>| | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>12</td>
<td>11</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Page-Table Base Address</td>
<td>Avail</td>
<td>G</td>
<td>P</td>
<td>S</td>
<td>0</td>
<td>A</td>
<td>P</td>
<td>C</td>
<td>D</td>
<td>P</td>
<td>W</td>
<td>T</td>
<td>U</td>
</tr>
</tbody>
</table>

- **Available for system programmer’s use**
- **Global page (Ignored)**
- **Page size (0 indicates 4 KBytes)**
- **Reserved (set to 0)**
- **Accessed**
- **Cache disabled**
- **Write-through**
- **User/Supervisor**
- **Read/Write**
- **Present**
### x86 page table entry

**Page-Table Entry (4-KByte Page)**

<table>
<thead>
<tr>
<th>31</th>
<th>12 11</th>
<th>9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>G P A T D A P C D P W T U S R W P</td>
</tr>
<tr>
<td>Page Base Address</td>
<td>Avail</td>
<td>G P A T D A P C D P W T U S R W P</td>
</tr>
</tbody>
</table>

- Available for system programmer’s use
- Global Page
- Page Table Attribute Index
- Dirty
- Accessed
- Cache Disabled
- Write-Through
- User/Supervisor
- Read/Write
- Present
• **x86 architecture *also* supports segmentation**
  - Segment register base + pointer val = *linear address*
  - Page translation happens on linear addresses

• **Two levels of protection and translation check**
  - Segmentation model has four privilege levels (CPL 0–3)
  - Paging only two, so 0–2 = kernel, 3 = user

• **Why do you want both** paging and segmentation?

---

---
x86 hardware segmentation

- **x86 architecture also supports segmentation**
  - Segment register base + pointer val = *linear address*
  - Page translation happens on linear addresses

- **Two levels of protection and translation check**
  - Segmentation model has four privilege levels (CPL 0–3)
  - Paging only two, so 0–2 = kernel, 3 = user

- **Why do you want both paging and segmentation?**

- **Short answer: You don’t – just adds overhead**
  - Most OSes use “flat mode” – set base = 0, bounds = 0xffffffff in all segment registers, then forget about it
  - x86-64 architecture removes much segmentation support

- **Long answer: Has some fringe/incidental uses**
  - VMware runs guest OS in CPL 1 to trap stack faults
  - OpenBSD used CS limit for $W^X$ when no PTE NX bit
Making paging fast

- x86 PTs require 3 memory references per load/store
  - Look up page table address in page directory
  - Look up physical page number (PPN) in page table
  - Actually access physical page corresponding to virtual address

- For speed, CPU caches recently used translations
  - Called a *translation lookaside buffer* or TLB
  - Typical: 64-2K entries, 4-way to fully associative, 95% hit rate
  - Each TLB entry maps a VPN → PPN + protection information

- On each memory reference
  - Check TLB, if entry present get physical address fast
  - If not, walk page tables, insert in TLB for next time
    (Must evict some entry)
TLB details

- TLB operates at CPU pipeline speed $\Rightarrow$ small, fast
- Complication: what to do when switch address space?
  - Flush TLB on context switch (e.g., old x86)
  - Tag each entry with associated process’s ID (e.g., MIPS)
- In general, OS must manually keep TLB valid
- E.g., x86 `invlpg` instruction
  - Invalidates a page translation in TLB
  - Note: very expensive instruction (100–200 cycles)
  - Must execute after changing a possibly used page table entry
  - Otherwise, hardware will miss page table change
- More Complex on a multiprocessor (TLB shootdown)
  - Requires sending an interprocessor interrupt (IPI)
  - Remote processor must execute `invlpg` instruction
x86 Paging Extensions

- **PSE: Page size extensions**
  - Setting bit 7 in PDE makes a 4MB translation (no PT)

- **PAE Page address extensions**
  - Newer 64-bit PTE format allows 36 bits of physical address
  - Page tables, directories have only 512 entries
  - Use 4-entry Page-Directory-Pointer Table to regain 2 lost bits
  - PDE bit 7 allows 2MB translation

- **Long mode PAE**
  - In Long mode, pointers are 64-bits
  - Extends PAE to map 48 bits of virtual address (next slide)
  - Why aren’t all 64 bits of VA usable?
x86 long mode paging

Virtual Address

<table>
<thead>
<tr>
<th>63</th>
<th>48</th>
<th>47</th>
<th>39 38</th>
<th>30 29</th>
<th>21 20</th>
<th>12 11</th>
<th>0</th>
</tr>
</thead>
</table>

- Page-Map Level-4 Table
- Page-Directory Pointer Table
- Page-Directory Table
- Page Table
- 4-Kbyte Physical Page

Physical Page

CR3

Page-Map L4 Base Addr

PML4E

PDPE

PDE

PTE

Physical Address

51

12

52

52

52

52

1251

9

9

9

9

12
Where does the OS live?

- **In its own address space?**
  - Can’t do this on most hardware (e.g., syscall instruction won’t switch address spaces)
  - Also would make it harder to parse syscall arguments passed as pointers

- **So in the same address space as process**
  - Use protection bits to prohibit user code from writing kernel

- **Typically all kernel text, most data at same VA in every address space**
  - On x86, must manually set up page tables for this
  - Usually just map kernel in contiguous virtual memory when boot loader puts kernel into contiguous physical memory
  - Some hardware puts physical memory (kernel-only) somewhere in virtual address space
Hardware/Software Managed TLBs

- **Intel Evolution:**
  - 80286: Segmentation Only
  - 80386: Segmentation + Paging
  - AMD64/x86-64: Paging (segmentation basically removed)

- **Hardware Managed TLB: x86 hardware reloads the TLB**
  - Hardware reloads the TLB as needed from page tables
  - Violations of protection bits lead to page faults

- **Software Managed TLB: MIPS, SPARC, Alpha, POWER**
  - Page fault triggered to ask OS to reload the TLB
  - OS designs its own paging structure
  - Hardware must provide fast exception handling
Very different MMU: MIPS

- Hardware checks TLB on application load/store
  - References to addresses not in TLB trap to kernel

- Each TLB entry has the following fields:
  Virtual page, Pid, Page frame, NC, D, V, Global

- Kernel itself unpaged
  - All of physical memory contiguously mapped in high VM
    (hardwired in CPU)
  - Kernel uses these pseudo-physical addresses

- User TLB fault hander very efficient
  - Two hardware registers reserved for it
  - utlb miss handler can itself fault—allow paged page tables

- OS is free to choose page table format!
MIPS Memory Layout

FFFF FFFF
C000 0000
BFFF FFFF
A000 0000
9FFF FFFF
8000 0000
7FFF FFFF
0000 0000

kseg2: Paged Kernel
kseg1: Phys. (Uncached)
kseg0: Physical Memory

Kernel Memory
User Memory

kuseg
User Paging