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>bochs/pintos</td>
<td>0x4000</td>
</tr>
<tr>
<td>emacs</td>
<td>0x3000</td>
</tr>
<tr>
<td></td>
<td>0x0000</td>
</tr>
</tbody>
</table>

- Consider multiprogramming on physical memory
  - What happens if pintos needs to expand?
  - If emacs needs more memory than is on the machine??
  - If pintos has an error and writes to address 0x7100?
  - When does gcc have to know it will run at 0x4000?
  - What if emacs 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 memory locations
  - Processes often require large amounts of contiguous memory (for stack, large data structures, etc.)

- **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 run time, Memory-Management Unit relocates each load, store to actual memory… App doesn’t see physical memory

- 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 will be idle (80/20 rule).
  - Write idle parts to disk until needed
  - Let other processes use memory for idle part
  - Like CPU virtualization: when process not using CPU, switch. When not using a page switch it to another process.

- Challenge: VM = extra layer, could be slow
Idea 1: load-time linking

- Link as usual, but keep the list of references
- Fix up process when actually executed
  - Determine where process will reside in memory
  - Adjust all references within program (using addition)
- Problems?
Idea 1: load-time linking

- Link as usual, but keep the list of references
- Fix up process when actually executed
  - Determine where process will reside in memory
  - Adjust all references within program (using addition)
- Problems?
  - How to enforce protection
  - How to move once in memory (Consider: data pointers)
  - What if no contiguous free region fits program?
Idea 2: base + bounds register

- Two special privileged registers: base and bound

- On each load/store:
  - Physical address = virtual address + base
  - Check $0 \leq \text{virtual address} < \text{bound}$, else trap to kernel

- How to move process in memory?

- What happens on context switch?
Idea 2: base + bounds register

- Two special privileged registers: base and bound
- On each load/store:
  - 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
Definitions

- Programs load/store to **virtual** (or **logical**) addresses
- Actual memory uses **physical** (or **real**) addresses
- VM Hardware is Memory Management Unit (**MMU**)
Address space
Base+bound 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**
Base+bound 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 bochs, both running pintos)

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

<table>
<thead>
<tr>
<th>Seg</th>
<th>base</th>
<th>bounds</th>
<th>rw</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x4000</td>
<td>0x6ff</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>0x0000</td>
<td>0x4ff</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>0x3000</td>
<td>0xffff</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>00</td>
</tr>
</tbody>
</table>

- **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** $\implies$ Inability to use free memory
- Over time:
  - Variable-sized pieces = many small holes (external frag.)
  - 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 with only about 5% slowdown [Yee]
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 of page
Paging trade-offs

- Eliminates external fragmentation
- Simplifies allocation, free, and backing storage (swap)
- Average internal fragmentation of .5 pages per "segment"
Simplified allocation

- 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 4K) bits of address are page offset
  - Most significant bits are page number

• Each process has a page table
  - Maps virtual page numbers to physical page numbers
  - 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
  - See pagedir_activate in Pintos

- 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 old intel manual for simplest explanation
  - Also volume 2 of AMD64 Architecture docs
  - Also volume 3A of latest Pentium Manual
x86 page translation

32 bits aligned onto a 4 KByte boundary

1024 PDE × 1024 PTE = 2^{20} Pages
**x86 page directory entry**

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

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Available for system programmer’s use</td>
</tr>
<tr>
<td>1</td>
<td>Global page (Ignored)</td>
</tr>
<tr>
<td>2</td>
<td>Page size (0 indicates 4 KBytes)</td>
</tr>
<tr>
<td>3</td>
<td>Reserved (set to 0)</td>
</tr>
<tr>
<td>4</td>
<td>Accessed</td>
</tr>
<tr>
<td>5</td>
<td>Cache disabled</td>
</tr>
<tr>
<td>6</td>
<td>Write-through</td>
</tr>
<tr>
<td>7</td>
<td>User/Supervisor</td>
</tr>
<tr>
<td>8</td>
<td>Read/Write</td>
</tr>
<tr>
<td>9</td>
<td>Present</td>
</tr>
</tbody>
</table>

Page-Table Base Address

Avail

G

P

S

0

A

P

C

D

P

W

T

U

S

R

W

P
# x86 Page Table Entry

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

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-0</td>
<td>Page Base Address</td>
</tr>
<tr>
<td>12-11</td>
<td>Available for system programmer’s use</td>
</tr>
<tr>
<td>9</td>
<td>Global Page</td>
</tr>
<tr>
<td>8</td>
<td>Page Table Attribute Index</td>
</tr>
<tr>
<td>7</td>
<td>Dirty</td>
</tr>
<tr>
<td>6</td>
<td>Accessed</td>
</tr>
<tr>
<td>5</td>
<td>Cache Disabled</td>
</tr>
<tr>
<td>4</td>
<td>Write-Through</td>
</tr>
<tr>
<td>3</td>
<td>User/Supervisor</td>
</tr>
<tr>
<td>2</td>
<td>Read/Write</td>
</tr>
<tr>
<td>1</td>
<td>Present</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
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?**
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 most 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\(^\wedge\)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 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 \textit{invlpg} instruction
  - Invalidates a page translation in TLB
  - Must execute after changing a possibly used page table entry
  - Otherwise, hardware will miss page table change
- More Complex on a multiprocessor (TLB shootdown)
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 are aren’t upper 16 bits of VA used?
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
Pintos memory layout

- Kernel/Pseudo-physical memory
  - User stack
  - BSS / Heap
  - Data segment
  - Code segment
  - Invalid virtual addresses

- PHYS_BASE: 0xc0000000
- PHYS_BASE + 32768: 0x00000000
Very different MMU: MIPS

• Hardware has 64-entry TLB
  - 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
  - Kernel uses these pseudo-physical addresses

• User TLB fault handler 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!
DEC Alpha MMU

- Software managed TLB (like MIPS)
  - 8KB, 64KB, 512KB, 4MB pages all available
  - TLB supports 128 instruction/128 data entries of any size

- But TLB miss handler not part of OS
  - Processor ships with special “PAL code” in ROM
  - Processor-specific, but provides uniform interface to OS
  - Basically firmware that runs from main memory like OS

- Various events vector directly to PAL code
  - call_pal instruction, TLB miss/fault, FP disabled

- PAL code runs in special privileged processor mode
  - Interrupts always disabled
  - Have access to special instructions and registers
PAL code interface details

- **Examples of Digital Unix PALcode entry functions**
  - callsys/retsys - make, return from system call
  - swpctx - change address spaces
  - wrvptptr - write virtual page table pointer
  - tbi - TBL invalidate

- **Some fields in PALcode page table entries**
  - GH - 2-bit granularity hint → $2^N$ pages have same translation
  - ASM - address space match → mapping applies in all processes
Example: Paging to disk

- gcc needs a new page of memory
- OS re-claims an idle page from emacs
- If page is *clean* (i.e., also stored on disk):
  - E.g., page of text from emacs binary on disk
  - Can always re-read same page from binary
  - So okay to discard contents now & give page to gcc
- If page is *dirty* (meaning memory is only copy)
  - Must write page to disk first before giving to gcc
- Either way:
  - Mark page invalid in emacs
  - emacs will fault on next access to virtual page
  - On fault, OS reads page data back from disk into new page, maps new page into emacs, resumes executing
Paging in day-to-day use

- Demand paging
- Growing the stack
- BSS page allocation
- Shared text
- Shared libraries
- Shared memory
- Copy-on-write (fork, mmap, etc.)
- Q: Which pages should have global bit set on x86?