18-447 Lecture 17:
VM Concepts: Address Translation

James C. Hoe
Department of ECE
Carnegie Mellon University
Housekeeping

• Your goal today
  – see “Virtual Memory” in easy to digest pieces
  – you will come to see memory as either more or less magical

• Notices
  – HW 4, due 4/8
  – Lab 3, due this week
  – Lab 4 out on Thursday 3:20

• Readings
  – P&H Ch 5
RV32I Programmer-Visible State

- **program counter**: 32-bit “byte” address of current instruction

- **M[0]**
- **M[1]**
- **M[2]**
- **M[3]**
- **M[N-1]**

2³² by 8-bit locations (4 GBytes) indexed using 32-bit “byte” addresses

*(take this literally for now; magic to come)*

**note**: x0=0
- x1
- x2

**general purpose register file**
- 32x 32-bit words
- named x0...x31
A RISC-V hart has a single byte-addressable address space of $2^{XLEN}$ bytes for all memory accesses. A word of memory is defined as 32 bits (4 bytes). Correspondingly, a halfword is 16 bits (2 bytes), a doubleword is 64 bits (8 bytes), and a quadword is 128 bits (16 bytes). The memory address space is circular, so that the byte at address $2^{XLEN} - 1$ is adjacent to the byte at address zero. Accordingly, memory address computations done by the hardware ignore overflow and instead wrap around modulo $2^{XLEN}$.

The execution environment determines the mapping of hardware resources into a hart’s address space. Different address ranges of a hart’s address space may (1) be vacant, or (2) contain main memory, or (3) contain one or more I/O devices. Reads and writes of I/O devices may have visible side effects, but accesses to main memory cannot. Although it is possible for the execution environment to call everything in a hart’s address space an I/O device, it is usually expected that some portion will be specified as main memory.
2 Parts to Modern VM

• In a multi-tasking system, **virtual** memory supports the **illusion** of a **large**, **private**, and **uniform** memory space to each process

• Ingredient A: naming and protection
  – each process sees a large, contiguous address space without holes **(for convenience)**
  – each process’s memory is private, i.e., protected from access by other processes **(for sharing)**

• Ingredient B: demand paging **(for hierarchy)**
  – capacity of secondary storage (disk)
  – speed of primary storage (DRAM)
The Common Denominator: Address Translation

• Large, private, and uniform abstraction achieved through address translation
  – user process operates on effective address (EA)
  – HW translates from EA to physical address (PA) on every memory reference

• Through address translation
  – control which physical locations (DRAM and/or disk) can be referred to by a process
  – allow dynamic allocation and relocation of physical backing store (where in DRAM and/or disk)

• Address translation HW and policies controlled by the OS and protected from user
Beginnings of Memory Protection

• No need for protection or translation early on
  – single process, single user at a time
  – access all locations directly with PA

• Cooperative Multitasking
  – each process limited to a non-overlapping, contiguous physical memory region
    (space doesn’t start from addr 0 . . . )
  – everything must fit in the region
  – how to keep one process from reading or trashing another process’s code and data? *(see corewars.org)*
Base and Bound

• A process’s private memory region defined by
  – **base**: starting address of region
  – **bound**: size of region

• User process issue “effective” address (EA) between 0 and the size of its allocated region (private and uniform)
Base and Bound Registers

• Translation and protection check in hardware on every user memory reference
  – $PA = EA + base$
  – if ($EA < bound$) then okay else violation
• When switching user processes, OS sets base and bound registers
• User processes cannot be allowed to modify base and bound registers themselves

Requires at least 2 privilege levels with protected instruction and state for OS only
Segmented Memory

- Limitations of single base-and-bound region
  - hard to find large contiguous space after a while—free space become fragmented
  - can two processes shared some memory regions but not others?
- A “base-and-bound” pair is a unit of protection
  ⇒ give user multiple memory “segments”
  - each segment is a contiguous memory region
  - each segment is defined by a base and bound pair
- Earliest use, separate code and data segments
  - 2 sets of base/bound for code vs data
  - forked processes can share code segments
  more elaborate later: code, data, stack, etc.
Segmented Address Space

- **EA** partitioned into segment number (**SN**) and segment offset (**SO**)
  - max segment size limited by the range of **SO**
  - active segment size set by **bound**
- Per-process segment translation table
  - map **SN** to corresponding **base** and **bound**
  - separate mapping for each process
  - privileged structure if used to enforce protection
Access Protection

- Per-segment access permissions can be specified as protection bits in segment table entries.
- Generic options include:
  - **Readable?**
  - **Writeable?**
  - **Executable?**
- For example:
  - Normal data segment $\Rightarrow$ **RW(!E)**
  - Static shared data segment $\Rightarrow$ **R(!W)(!E)**
  - Code segment $\Rightarrow$ **R(!W)E** *(self modifying code?)*
  - Illegal segment $\Rightarrow$ **(!R)(!W)(!E)** *(what for?)*

Access violation exception brings OS into play.
Aside: Another (ab)use of segments

- Extend old ISA to give new applications a large address space while stay compatible with old
- “User-managed” segmented addressing $SA \equiv EA_{small}$
  - old application use identity mapping in table; unaware of segments; can’t use more memory
  - new application reloads table at run time to access different regions in $EA_{large}$; unequal access to active vs inactive regions (what about pointers?)
Paged Address Space

• Divide **PA** and **EA** space into equal, fixed size segments known as “page frames”
  
  historically 4KByte pages

• **EA** and **PA** are interpreted as page number (**PN**) and page offset (**PO**)
  
  – page table translates **EPN** to **PPN**; **EPO**=**PPO**
  
  – **PA**={**PPN**, **PO**}

```plaintext
  EPN  PO
  \downarrow
  \hline
  \hline
  page table

  PPN perm’s?

  concat

  PA
```
Fragmentation

• External fragmentation by segments
  – plenty of unallocated DRAM but none in contiguous region of a sufficient size
  – paged memory eliminates external fragmentation

• Internal fragmentation of pages
  – entire page (4KByte) is allocated; unused bytes go to waste
  – smaller page size reduces internal fragmentation
  – modern ISA moving to larger page sizes (MBytes) in addition to 4KBytes

Segments and pages not meant for the same role
Demand Paging

• Use main memory and disk (swap vs. mmap file) as automatically managed memory hierarchies analogous to cache vs. main memory

• Early attempts
  – von Neumann already described manual memory hierarchies
  – Brookner’s interpretive coding, 1960: *program interpreter managed paging between a 40KByte main memory and a 640KByte drum*
  – Atlas, 1962: *hardware managed paging between 32-page core memory and 192-page drum (512 word/page)*
Demand Paging: just like caching

- $M$ bytes of storage (DRAM+Disk), keep most frequently used $C$ bytes in DRAM where $C < M$

- Same basic issues as before
  1. where to place a page in DRAM or disk?
  2. how to find a page in DRAM or disk?
  3. when to bring a page into DRAM from disk?
  4. which page to evict from DRAM to disk to free-up DRAM for new pages?

- Conceptual difference in swap vs. cache
  - DRAM doesn’t hold “copies” of what is on disk
  - a page in $M$ either in DRAM or disk (or non-existent)
  - address not bound to 1 location for all time

  "virtual"

  DRAM is cache for mmap’ed file
Demand Paging: not at all like caching

- Drastically different size and time scale leads to drastically different implementation choices

<table>
<thead>
<tr>
<th></th>
<th>L1 Cache</th>
<th>L2 Cache</th>
<th>Demand Paging</th>
</tr>
</thead>
<tbody>
<tr>
<td>capacity</td>
<td>10s KByte</td>
<td>MByte</td>
<td>GByte</td>
</tr>
<tr>
<td>block size</td>
<td>10s Byte</td>
<td>≥ L1</td>
<td>4K~4M Byte</td>
</tr>
<tr>
<td>hit time</td>
<td>few cyc</td>
<td>few 10s cyc</td>
<td>few 100s cyc</td>
</tr>
<tr>
<td>miss penalty</td>
<td>few 10s cyc</td>
<td>few 100s cyc</td>
<td>10 msec</td>
</tr>
<tr>
<td>miss rate</td>
<td>0.1~10%</td>
<td>&lt;0.1%</td>
<td>0.00001~0.001%</td>
</tr>
</tbody>
</table>

(Per mem reference not per cache access)

- Hit time, miss penalty, miss rate not independent!!
Don’t use “VM” to mean everything

- Effective Address (EA): emitted by user instructions in a *per-process* space (protection)
- Physical Address (PA): corresponds to actual storage locations on DRAM or on disk
- Virtual Address (VA): refers to locations in a *single system-wide*, large, linear address space; not all locations in VA space have physical backing (demand paging)
**EA, VA and PA (IBM Power view)**

64-bit EA<sub>0</sub> divided into X fixed-size segments

64-bit EA<sub>1</sub> divided into X fixed-size segments

80~90-bit VA divided into Y segments (Y>>X); also divided as Z pages

40~50-bit PA divided into W pages (Z>>W)

Swap space divided into V pages (Z>>V, V>>W)

Segmented EA:
- private, contiguous + sharing

Demand paged VA:
- capacity of disk, speed of DRAM
EA, VA and PA (almost everyone else)

EA₀ with unique ASID=0

EAᵢ with unique ASID=i

EA divided into N “address spaces” indexed by ASID;
also divided as Z pages

VA divided into N “address spaces” indexed by ASID;
also divided as Z pages

PA divided into W pages (Z>>W)

Swap space divided into V pages (Z>>V, V>?W)

Easy to blur EA and VA colloquially but full VA is \{ASID, EA\}!!!

how do processes share pages?
Just one more thing:
How large is the page table?

- A page table holds mapping from \textbf{VPN} to \textbf{PPN}
- Suppose 64-bit \textbf{VA} and 40-bit \textbf{PA}, how large is the page table?
  \[2^{52} \text{ entries} \times \sim 4 \text{ bytes} \approx 16 \times 10^{15} \text{ Bytes}\]

And that is for just one process!!?
How large should it be?

• Don’t need to track entire VA space
  – total allocated VA space is $2^{64}$ bytes x # processes, but most of which not backed by storage
  – can’t use more memory locations than physically exist (DRAM and disk)

• A clever page table should scale linearly with physical storage size and not VA space size

• Table cannot be too convoluted
  – a page table is accessed not infrequently
  – a page table should be “walkable” quickly in HW

Two dominant schemes in use today:

* hierarchical page table and hashed page table