The Limits of Physical Addressing

<table>
<thead>
<tr>
<th>Programs use &quot;Physical addresses&quot; of memory locations</th>
</tr>
</thead>
<tbody>
<tr>
<td>All programs shared one address space:</td>
</tr>
<tr>
<td>The physical address space</td>
</tr>
<tr>
<td>Machine language programs had to be aware of the machine organization</td>
</tr>
<tr>
<td>No way to prevent a program from accessing any machine resource in memory</td>
</tr>
</tbody>
</table>

In the olden days ...

- The loader would locate an unused set of main memory addresses and load the program and data there
- There would be a special register called the relocation register, and all addresses that the program used would be interpreted as addresses relative to the base address in that register
- So if the program jumped to location 54, the jump would really be to 54 + contents of relocation register. A similar thing, perhaps with a second register, would happen for data references

(From Sussman)
In the less-olden days...

- It became difficult to find a contiguous segment of memory big enough to hold program and data, so the program was divided into pages, with each page stored contiguously, but different pages in any available spot, either in main memory or on disk.
- This is the virtual addressing scheme.
- To the program, memory looks like a contiguous segment, but actually, data is scattered in main memory and perhaps on disk.

Solution: Add a Layer of Indirection

All user programs run in an standardized virtual address space starting at zero.

- Needs fast(?) Address Translation hardware, managed by the operating system (OS), to map each virtual address to physical memory.
- Hardware supports “modern” OS features: Memory protection, Address translation, Sharing.

But we know all about this!

- Already know that a program and data can be scattered between cache memory and main memory.
- Now add the reality that its location in main memory is also determined in a scattered way, and some pages may also be located on disk.
- So each page has its own relocation value.

Three Advantages of Virtual Memory

- Translation:
  - Program can be given consistent view of memory, even though physical memory is scrambled (pages of programs in any order in physical RAM).
  - Makes multithreading reasonable (now used a lot!).
  - Only the most important part of each program (“the Working Set”) must be in physical memory at any one time.
  - Contiguous structures (like stacks) use only as much physical memory as necessary, yet still can grow later as needed, without recopying.
- Protection (most important now):
  - Different threads (or processes) protected from each other.
  - Different pages can be given special behavior (Read Only, Invisible to User, …)
  - Kernel and OS data are protected from access by User programs.
  - Very important for protection from malicious programs.
- Sharing:
  - Can map same physical page to multiple users (“Shared memory” holding C++ compiler for many users at once).
Virtual Memory – Fig. B.19

Parameter First-level cache Virtual memory

<table>
<thead>
<tr>
<th>Parameter</th>
<th>First-level cache</th>
<th>Virtual memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block (page) size</td>
<td>16-128 bytes</td>
<td>4096-65,536 bytes</td>
</tr>
<tr>
<td>Hit time</td>
<td>1-3 clock cycles</td>
<td>100-200 cc</td>
</tr>
<tr>
<td>Miss penalty</td>
<td>8-200 cc</td>
<td>10^6-10^7 cc</td>
</tr>
<tr>
<td>(access time)</td>
<td>(6-160 cc)</td>
<td>(.8-8) * 10^6 cc</td>
</tr>
<tr>
<td>(transfer time)</td>
<td>(2-40 cc)</td>
<td>(.2-2) * 10^6 cc</td>
</tr>
<tr>
<td>Miss rate</td>
<td>0.1-10%</td>
<td>0.00001-0.001%</td>
</tr>
<tr>
<td>Address mapping</td>
<td>25-45 bit physical address to 14-20 bit cache address</td>
<td>32-64 bit virtual address to 25-45 bit physical address</td>
</tr>
</tbody>
</table>

Cache vs. Virtual Memory

<table>
<thead>
<tr>
<th>Cache</th>
<th>Virtual memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache miss handled by hardware</td>
<td>Page faults handled by operating system</td>
</tr>
<tr>
<td>Cache size fixed for a particular machine</td>
<td>Virtual memory size fixed for a particular program</td>
</tr>
<tr>
<td>Fundamental unit is a block</td>
<td>Fundamental unit is a fixed-length page or a variable-length segment</td>
</tr>
<tr>
<td>cache fault</td>
<td>page fault</td>
</tr>
</tbody>
</table>

Old Protection Mode: Base & Bound

- User processes need to be protected from each other
- Two registers, base and bound test whether this virtual address belongs to this process (Built in Cray-1)
- If not, a memory protection violation exception is raised
- Users cannot change the base and bound registers
Who can change them?

- The operating system needs access to the base and bound registers
- So a process that is labeled kernel (also called supervisor or executive) can access any memory location and change the registers
- Kernel processes are accessed through system calls, and a return to user mode is like a subroutine return, restoring the state of the user process

Segmentation

- Basically multiple base & bounds
  - w/ virtual memory

Each segment can be located anywhere in physical memory

Virtual Memory - Page Tables

- Encode mappings of code’s virtual addresses to physical memory address space
- A valid page table entry codes the present physical memory “frame” address for the page

A virtual address space (V.A.S.) is divided into blocks of memory called pages

A page table is indexed by a virtual address

Details of Page Table

- Page table maps virtual page numbers to physical frames (“PTE” = Page Table Entry)
- Virtual memory treats main memory ≈ cache for disk

OS manages the page table for each V.A.S. ID

Page Table

Physical Memory Space

Physical Address

Virtual Address (for 4,096 Bytes/page)
Use of Page Table – another view

A table for 4KB pages for a 32-bit physical address space (max 4GB) has 1M entries!

Each process needs its own address space tables!

Two-level Page Tables

Choosing page size

- A large page size
  - keeps page table small.
  - reduces cache miss times, if accesses have locality
  - reduces start-up overhead in moving data from disk to memory
  - means fewer TLB misses
- but also
  - wastes memory (internal fragmentation)
  - increases the time to start up a program
VM and Disk: Page Replacement Policy

Set of all pages in memory

Head pointer: Place page on free list if used bit is still clear (=0).
Schedule freed pages with dirty bit set to be written to disk.

Tail pointer: Clear (=0) "used bits" in page table, says maybe not used recently.

Architect’s role: support setting dirty and used bits

Free Pages

Page Table

dirty/used |  Page Table
-------|-------
0 1 0  ... 0
0 1 1 0 0
0 0 0 1 1

Virtual Address Translation: How does it work?

- Per-Process Translation Look-Aside Buffer (TLB)
- A small very fast fully-associative cache of mappings from virtual to physical addresses
- TLB also contains protection bits for virtual address
- Fast common case: If virtual address is in TLB, process has permission to read/write it.

MIPS Address Translation: How does it work?

- "Virtual Addresses"
- "Physical Addresses"

The TLB Caches Page Table Entries

Physical and virtual pages must be the same size! Here, 1024 Bytes/page each.

Translation Lookaside Buffer – Another view

MIPS handles TLB misses in software (random replacement). Other machines use hardware.
Can TLB translation overlap cache indexing? 

<table>
<thead>
<tr>
<th>Virtual Page Number</th>
<th>Page Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag Part of Physical Address</td>
<td>Physical Page Number</td>
</tr>
</tbody>
</table>

- **Index**
- **Byte Select**

**Virtual Translation Look-Aside Buffer (TLB)**

**Physical**

**Having cache index in page offset works, but**

**Q. What is the downside?**

**A. Inflexibility. Size of cache limited by page size.**

Problems With Overlapped TLB Access

- Overlapped access only works so long as the address bits used to index into the cache do not change as the result of VA translation.
- This usually limits overlapping to small caches, or large page sizes, or high n-way set associative caches if you want a large capacity cache.
- Example: suppose everything the same except that the cache is increased to 8 KB instead of 4 KB:

```
11 2 44 10 2-way set assoc cache

This bit is changed by VA translation, but it is needed for cache lookup.
```

**Solutions:**
- Go to 8KByte page sizes; SW guarantee VA[13]=PA[13]
- Go to 2-way set associative cache; or
**Common Organization**

```
CPU -> TLB
  |    |    |
  L1 Cache | Write Buffer | L2 Cache
```

Even a cache hit requires TLB translation first!

**Can CPU use virtual addresses for cache?**

- If cache index in virtual address, only cache misses use TLB!
- Downside: a subtle, fatal problem. What is it?
- Answer: Synonym problem (Aliasing)
  - If two (virtual) address spaces share a physical frame, data may be in cache twice. Maintaining consistency is a nightmare.

**Paging vs. Segmentation**

<table>
<thead>
<tr>
<th></th>
<th>Page</th>
<th>Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Words per address</td>
<td>One</td>
<td>Two (segment/offset)</td>
</tr>
<tr>
<td>Programmer visible?</td>
<td>Invisible to app programmer</td>
<td>May be visible to app programmer</td>
</tr>
<tr>
<td>Replacing a block</td>
<td>Trivial (all blocks same size)</td>
<td>Hard (must find contiguous, variable-sized chunk)</td>
</tr>
<tr>
<td>Memory use inefficiency</td>
<td>Internal fragmentation (within page)</td>
<td>External fragmentation (in unused memory)</td>
</tr>
<tr>
<td>Efficient disk traffic</td>
<td>Yes (can adjust page size)</td>
<td>Not always (small segment problem)</td>
</tr>
</tbody>
</table>

**Summary - The Cache Design Space**

- Several interacting dimensions
  - cache size
  - block size
  - associativity
  - replacement policy
  - write-through vs. write-back
  - write allocation
- The optimal choice is a compromise
  - depends on access characteristics
    - workload
    - use (I-cache, D-cache, TLB)
  - depends on technology / cost
- Simplicity often wins
Summary - Caches

- The Principle of Locality:
  - Program access a relatively small portion of the address space at any instant of time.
  - Temporal / Spatial Locality: Locality in Time and Space

- Three Major Uniprocessor Categories of Cache Misses:
  - Compulsory Misses: sad facts of life. Example: cold start misses.
  - Capacity Misses: increase cache size
  - Conflict Misses: increase cache size and/or associativity.
  - Coherence Misses (in multiprocessors): Avoid frequent use of shared variables in parallel programs.

- Write Policy: Write Through vs. Write Back

Summary - TLB, Virtual Memory

- Page tables map virtual address to physical address
- TLBs are important for fast translation
- TLB misses are significant in processor performance
  - This decade is a funny time, since most systems cannot access all of 2nd level cache without TLB misses! The answer in newer processors is 2-levels of TLB.
- Questions to understand Caches, TLBs, VM:
  - Where can a block be placed?
  - How is a block found?
  - What block is replaced on a miss?
  - How are writes handled?
- Today, VM allows many processes to share single memory without having to swap all processes to disk.
- Today, VM protection is maybe even more important than memory hierarchy benefits, but computers still insecure