Virtual Memory

If I can see it and I can touch it, it’s real.
If I can’t see it but I can touch it, it’s invisible.
If I can see it but I can’t touch it, it’s virtual.
And if I can’t see it and I can’t touch it’s…gone!

Review: The Principle of Locality

The Principle of Locality:
- Program access a relatively small portion of the address space at any instant of time.
- Example: 90% of time in 10% of the code
Review: The Need to Make a Decision!

- **Direct Mapped Cache:**
  - Each memory location can only mapped to 1 cache location
  - No need to make any decision :-)  
    - Current item replaced the previous item in that cache location
- **N-way Set Associative Cache:**
  - Each memory location have a choice of N cache locations
- **Fully Associative Cache:**
  - Each memory location can be placed in ANY cache location
- **Cache miss in a N-way Set Associative or Fully Associative Cache:**
  - Bring in new block from memory
  - Throw out a cache block to make room for the new block
  - Damn! We need to make a decision which block to throw out!

Review: Summary

- **The Principle of Locality: Temporal Locality vs Spatial Locality**
- **Four Questions For Any Cache**
  - Where to place in the cache
  - How to locate a block in the cache
  - Replacement
  - Write policy: Write through vs Write back
    - Write miss:
  - **Three Major Categories of Cache Misses:**
    - Compulsory Misses: sad facts of life. Example: cold start misses.
    - Conflict Misses: increase cache size and/or associativity.  
      Nightmare Scenario: ping pong effect!
    - Capacity Misses: increase cache size
Today's Topic --- Virtual Memory

Provides *illusion* of very large memory
- sum of the memory of many jobs greater than physical memory
- address space of each job larger than physical memory

Allows available (fast and expensive) physical memory to be very well utilized

Simplifies memory management (*main reason today*)

Exploits memory hierarchy to keep average access time low.

Involves at least two storage levels: *main* and *secondary*

**Virtual Address** -- address used by the programmer

**Virtual Address Space** -- collection of such addresses

**Memory Address** -- address of word in physical memory
also known as "physical address" or "real address"
Basic Issues in VM System Design

- size of information blocks that are transferred from secondary to main storage
- block of information brought into M, and M is full, then some region of M must be released to make room for the new block --> replacement policy
- which region of M is to hold the new block --> placement policy
- missing item fetched from secondary memory only on the occurrence of a fault --> fetch/load policy

Paging Organization

virtual and physical address space partitioned into blocks of equal size

Address Map

\[ V = \{0, 1, \ldots, n-1\} \text{ virtual address space} \]
\[ M = \{0, 1, \ldots, m-1\} \text{ physical address space} \]

\[ n > m \]

\[ \text{MAP: } V \rightarrow M \cup \{\emptyset\} \text{ address mapping function} \]

\[ \text{MAP}(a) = a' \text{ if data at virtual address } a \text{ is present in physical address } a' \text{ in } M \]

\[ = \emptyset \text{ if data at virtual address } a \text{ is not present in } M \]
Page Table

We often use page table to implement the Address Translation mechanism.

Paging Organization

Address Mapping

actually, concatenation is more likely
**Address Mapping Algorithm**

If $V = 1$
- then page is in main memory at frame address stored in table
- else address located page in secondary memory

Access Rights
- $R =$ Read-only, $R/W =$ read/write, $X =$ execute only

If kind of access not compatible with specified access rights,
- then *protectionViolationFault*

If valid bit not set then *page fault*

*Protection Fault*: access rights violation; causes trap to hardware,
- microcode, or software fault handler

*Page Fault*: page not resident in physical memory, also causes a trap;
- usually accompanied by a *context switch*: current process
- suspended while page is fetched from secondary storage

---

**Fragmentation & Relocation**

*Fragmentation* is when areas of memory space become unavailable for
some reason

*Relocation*: move program or data to a new region of the address
space (possibly fixing all the pointers)

*External Fragmentation*: Space left between blocks.

*Internal Fragmentation:*
- program is not an integral # of pages, part of the last page frame is
  “wasted” (obviously less of an issue as physical memories get
  larger)
- diagram 0 1 occupied k-1
Optimal Page Size

Choose page that minimizes fragmentation

- large page size => internal fragmentation more severe
- BUT increases the # of pages / name space => larger page tables

In general, the trend is towards larger page sizes because

- memories get larger as the price of RAM drops
- the gap between processor speed and disk speed grow wider
- programmers desire larger virtual address spaces

Most machines at 4K byte pages today, with page sizes likely to increase

Page Replacement Algorithms

Just like cache block replacement!

**Least Recently Used (LRU):**
- selects the least recently used page for replacement

- requires knowledge about past references, more difficult to implement
  (thread thru page table entries from most recently referenced to least recently referenced; when a page is referenced it is placed at the head of the list; the end of the list is the page to replace)

- good performance, recognizes principle of locality
Example:

Suppose the most recent page references (in order) were
10, 12, 9, 7, 11, 10
When page 9 is referenced, which was not present in memory, and the memory is full. Which page should be replace in LRU?

Page Replacement (Continued)

*Not Recently Used:*
Associated with each page is a reference flag such that
ref flag = 1 if the page has been referenced in recent past
= 0 otherwise

-- if replacement is necessary, choose any page frame such that its reference bit is 0. This is a page that has not been referenced in the recent past

-- clock implementation of NRU:

<table>
<thead>
<tr>
<th>page entry</th>
<th>ref bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0</td>
<td>0</td>
</tr>
<tr>
<td>1 0</td>
<td>0</td>
</tr>
<tr>
<td>1 0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

last replaced pointer (lrp) if replacement is to take place, advance lrp to next entry (mod table size) until one with a 0 bit is found; this is the target for replacement; As a side effect, all examined PTE’s have their reference bits set to zero.

An optimization is to search for the a page that is both not recently referenced AND not dirty.
### Demand Paging and Prefetching Pages

**Fetch Policy**
when is the page brought into memory?
if pages are loaded solely in response to page faults, then the policy is **demand paging**

An alternative is **prefetching**:
- anticipate future references and load such pages before their actual use
  - reduces page transfer overhead
  - removes pages already in page frames, which could adversely affect the page fault rate
  - predicting future references usually difficult

Most systems implement demand paging without prepaging
(One way to obtain effect of prefetching behavior is increasing the page size)

### Virtual Address and a Cache

It takes an extra memory access to translate VA to PA
This makes cache access very expensive, and this is the "innermost loop" that you want to go as fast as possible

**ASIDE:** Why access cache with PA at all? VA caches have a problem, i.e. synonym problem: two different virtual addresses map to same physical address ➔ two different cache entries holding data for the same physical address!

- for update: must update all cache entries with same physical address or memory becomes inconsistent
- determining this requires significant hardware, essentially an associative lookup on the physical address tags to see if you have multiple hits
TLBs --- Making Address Translation Fast

A way to speed up translation is to use a special cache of recently used page table entries — this has many names, but the most frequently used is Translation Lookaside Buffer or TLB.

<table>
<thead>
<tr>
<th>Virtual Address (or tag)</th>
<th>Physical Address</th>
<th>Dirty</th>
<th>Ref</th>
<th>Valid</th>
<th>Access</th>
</tr>
</thead>
</table>

Translation Look-Aside Buffers

Just like any other cache, the TLB can be organized as fully associative, set associative, or direct mapped.

TLBs are usually small, typically not more than 128 - 256 entries even on high end machines. This permits fully associative lookup on these machines. Most mid-range machines use small n-way set associative organizations.
**Segmentation (see x86)**

Alternative to paging (often combined with paging)

Segments allocated for each program module; may be different sizes
segment is unit of transfer between physical memory and disk

```
| seg # | disp | Present | Access | Length | Phy Addr |
```

Faults:
- missing segment (Present = 0)
- overflow (Displacement exceeds segment length)
- protection violation (access incompatible with segment protection)

Segment-based addressing is sometimes used to implement capabilities,
i.e., hardware support for sophisticated protection mechanisms

**Segment Based Addressing**

*Three Serious Drawbacks:*

1. storage allocation with variable sized blocks
   (best fit vs. first fit vs. buddy system)

2. external fragmentation: physical memory allocated in such a
   fashion that all remaining pieces are too small to be allocated
   to any segment. Solved be expensive run-time memory compaction.

3. Non-linear address matching pointer arithmetic in C?

**The best of both worlds: paged segmentation schemes**

**virtual address:** seg # page # displacement

used by IBM: 4K byte pages, 16 x 1 Mbyte or 64 x 64 Kbyte segments
Conclusion #1

° Virtual Memory invented as another level of the hierarchy
° Today VM allows many processes to share single memory without having to swap all processes to disk, protection more important
° (Multi-level) page tables to map virtual address to physical address
° TLBs are important for fast translation
° TLB misses are significant in performance

Conclusion #2

° Theory of Algorithms & Compilers based on number of operations
° Compiler remove operations and “simplify” ops:
  Integer adds << Integer multiplies << FP adds << FP multiplies
  • Advanced pipelines => these operations take similar time
° As Clock rates get higher and pipelines are longer, instructions take less time but DRAMs only slightly faster (although much larger)
° Today time is a function of (ops, cache misses)
° Given importance of caches, what does this mean to:
  • Compilers?
  • Data structures?
  • Algorithms?
Common Framework for Memory Hierarchy

- Question 1: Where can a Block be Placed
  - Cache:
    - direct mapped, n-way set associative
  - VM:
    - fully associative

- Question 2: How is a block found
  - index,
  - index the set and search among elements
  - search all cache entries or separate lookup table

- Question 3: Which block be replaced
  - Random, LRU, NRU

- What happens on a write
  - write through vs write back
  - write allocate vs write no-allocate on a write miss