From Physical Memory To Virtual Memory: Understanding the Memory Hierarchy

- ref: [O'H&Bryant, sect 6.1–6.4] or [Null&Lobur, sect 6.2–6.4]; additionally [O'H&Bryant, sect 6.5–6.7]
- recall the memory hierarchy
- (main) memory: types, organization of chips
- memory controllers and memory access
- disks: anatomy, operation, access time and configuration
- trends in memory technologies
- caching (memory hierarchy) in the broader context
- memory systems: caches and their organization, virtual memory concerns
Memory Types

- **SRAM** (Static Random Access Memory)
  - basis is the D-latch [Tanenbaum, fig 3.24]
  - can hold 1 bit
  - requires 11 transistors (can be done in 6)
  - state persists providing power is on

- **DRAM** (Dynamic RAM)
  - requires 1 capacitor and 1 transistor!
  - capacitor stores just 40,000 electrons!
  - has a time constant of $10^{-3}$ sec
  - hence requires refresh (typically 10-100ms)

- **Read Only Memory** (ROM) – non-volatile (state persists if power is off)
  - programs (e.g. the BIOS) stored in ROM are termed firmware

<table>
<thead>
<tr>
<th></th>
<th>transistors / bit</th>
<th>rel. access time</th>
<th>persistent</th>
<th>sensitive</th>
<th>rel. cost</th>
<th>use</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM</td>
<td>6</td>
<td>$1 \times$</td>
<td>yes</td>
<td>no</td>
<td>100×</td>
<td>?</td>
</tr>
<tr>
<td>DRAM</td>
<td>1</td>
<td>$10 \times$</td>
<td>no</td>
<td>yes</td>
<td>1×</td>
<td>?</td>
</tr>
</tbody>
</table>
Organization and Access of (DRAM) Memory Chip

([O'H&Bryant, fig 6.4])

1. select row address \((RAS = 2)\)
2. select column address \((CAS = 1)\)
3. access the bits of the selected supercell

- why is the address bus 2 bits wide?
- why not a single index (0–15) per bit?
The Memory Module and Controller

Dual Inline Memory Module (DIMM, 168 pins; [O’H&Bryant, fig 6.5])
Accessing Memory

([O’H&Bryant, fig 6.6])

- processor/memory communicate over a shared bus (transactions)

- read transaction:
  - CPU places address on system bus; I/O bridge forwards to memory bus
  - main memory reads address and places content on memory bus
  - CPU reads word from system bus and copies to register

- write transaction:
  - as above, but main memory waits to receive data
Disks: Anatomy, Capacity and Operation

- Coated with magnetic material; all tracks are equidistant from spindle.
- Typically sectors are 512 bytes, tracks have 400 sectors.
- Capacity = $\frac{\text{bytes}}{\text{sector}} \times \frac{\text{avg. sectors}}{\text{track}} \times \frac{\text{tracks}}{\text{surface}} \times \frac{\text{surfaces}}{\text{platter}} \times \frac{\text{platters}}{\text{disk}}$
- Operation: read/write head connected to actuator arm.
  - Arm is $10^{-4}$ mm above surface and moves at 80km/h!
  - Hence disks come in airtight containers.

[O’H&Bryant, fig 6.9]: single platter multiple platters
Disk Access Times and Configuration

- Total access time is made up from:
  - Seek time: move arm to track (typically 9 ms)
  - Rotation latency: wait till sector reaches head (typically, with a 7200 RPM rotation rate, 4 ms)
  - Transfer time per sector (typically, with 400 tracks/sector, 0.02 ms (25 MB/s))

- What dominates this? Comparison with SRAM and DRAM?

- Disk configuration [O’H&Bryant, fig 6.11] (will consider access details later under Module 0)
The Take Home Message:

- different storage technologies have different prices and performance
- price / performance of different technologies is changing at different rates
- DRAM and disk access are lagging CPU cycle times
- how do we bridge the processor memory performance gap? (the ‘memory wall’)

[O’H&Bryant, fig 6.16]:
Caching in the Broader Context

- recall the memory hierarchy [O’H&Bryant, fig 6.21]
- requires spatial and/or temporal locality of data accesses to be effective
- data organized into ‘blocks’ to reduce management overheads
- this can be extended beyond a single computer system [O’H&Bryant, fig 6.23]:

<table>
<thead>
<tr>
<th>type</th>
<th>what</th>
<th>where</th>
<th>latency (cycles)</th>
<th>managed by</th>
</tr>
</thead>
<tbody>
<tr>
<td>registers</td>
<td>4-byte word</td>
<td>on-chip</td>
<td>0</td>
<td>compiler</td>
</tr>
<tr>
<td>TLB</td>
<td>address translation</td>
<td>on-chip</td>
<td>(0)</td>
<td>MMU / OS</td>
</tr>
<tr>
<td>L1 cache</td>
<td>32-byte block</td>
<td>on-chip</td>
<td>1</td>
<td>hardware</td>
</tr>
<tr>
<td>L2 cache</td>
<td>64-byte block</td>
<td>off-chip</td>
<td>10</td>
<td>hardware</td>
</tr>
<tr>
<td>virtual memory</td>
<td>(8KB) page</td>
<td>main memory</td>
<td>100</td>
<td>hardware / OS</td>
</tr>
<tr>
<td>buffer cache</td>
<td>parts of files</td>
<td>main memory</td>
<td>100</td>
<td>OS</td>
</tr>
<tr>
<td>network buffer cache</td>
<td>parts of files</td>
<td>main memory</td>
<td>10,000,000</td>
<td>AFS/NSF client</td>
</tr>
<tr>
<td>browser cache</td>
<td>web pages</td>
<td>local disk</td>
<td>10,000,000</td>
<td>web browser</td>
</tr>
<tr>
<td>web cache</td>
<td>web pages</td>
<td>remote server disks</td>
<td>1,000,000,000</td>
<td>web proxy server</td>
</tr>
</tbody>
</table>
Memory Systems

- In modern computer systems, we need fast memory access for large memories.

- Can achieve by:
  1. Cache memory (low cost, low access time)
  2. Wide memory access (moderate cost, moderate access time)
  3. Faster technology (?; high cost, high access time)

- Effective CPU speed has improved faster than memory access speed because ...?

- Recall the two main memory technologies:
  - **SRAM (Static RAM)**: high cost, low access time (~5ns), used in off-chip caches.
  - **DRAM (Dynamic RAM)**: low cost, high access time (~50ns), used in most main memories (low pinout).

- Recall memory access involves the stages:
  1. Select row address
  2. Select column address
  3. (R/W) Access selected bit(s)

    (These can however be pipelined)
Cache Memory

- idea: data that is "currently most needed" is brought into a (smaller) faster memory

- observation: memory accesses in most programs exhibit:
  - temporal locality: if access address X, likely to access X again soon
  - spatial locality: if access address X, likely to access X+1 soon
    ⇒ cache organized into lines (blocks) of \( L \) bytes \((L = 2^l, \text{e.g. } 4 \leq l \leq 9)\)
  - \( \sqrt{\text{blocked memory accesses (faster) & less control info needed (per byte)}} \)
  - \( \times \) redundant memory traffic if only ever use 1 byte per line
    - e.g. pointer chasing
      - (large address ‘strides’ guaranteed!)
  - if so, yields good cost-speed tradeoff

- cache hit rates: % of (word) accesses in program when data is in cache
  - need to be high (e.g. > 95%) for good performance
  - only possible if accesses have a sufficient locality

- problem: keeping consistency of data cache & main memory:
  - when a store instruction is executed, the relevant line is updated 1st;
    - when does main memory get updated?
Direct-Mapped Caches

- idea: for a cache of size $C' = 2^{c-l}$ lines,
  all addresses with same $a_1$ are mapped to the same cache line

$$X = \begin{array}{cccc}
31 & c & c-1 & l & l-1 & 0 \\
\hline
a_0 & a_1 & a_2
\end{array}$$

- easy to implement & low chip area / byte (note: cache must store value of $a_0$)
  ⇒ large $C'$ possible (better performance)

- cache conflicts: 2 (or more) words from memory map to the same cache line
  can make large, often unpredictable, performance losses

- alternative: K-way set associative caches
  - like a direct-mapped cache of size $C'/K$, but every line extended to a set of $K$ lines
  - addresses with same $a_1$ can map into any of the $K$ lines in the set
  - typically $K = 1, 2, 4, 6, 8$ (issue: replacement policy)
  - reduces chance of conflicts by factor of $K$
  - some extra cost / byte (more complex H/W needed)
Virtual Memory Concerns

- program addresses are virtual; H/W translates these into physical addresses
  - as for caches, this is (largely) transparent to the programmer
- on PeANUtl, how many memory references are needed to execute a load instruction? (a) 1 (b) 2 (c) 3 (d) 4
- real machines have (in their MMUs) a translation lookaside buffer (TLB) of $T$ entries
  - effectively a ‘cache’ of recent VM translations
  - typically $T = 64 \Rightarrow$ a working set of $>64$ pages causes TLB misses
    - usually causes an O.S. trap to access the page tables; costs $10^2..10^3$ cycles!
  - typically, TLBs are fully associative with an LRU replacement policy
- a working set of $T$ pages is needed to minimize TLB misses
- good locality can also minimize page faults (the swap space is at the very bottom of the memory hierarchy!)
- discussion point: list (at least) 6 similarities between caching and virtual memory