Storage Hierarchy II: Main Memory

main memory

- memory technology (DRAM)
- interleaving
- special DRAMs
- processor/memory integration

virtual memory and address translation
Readings

H+P

- chapter 5 (5.8 to 5.13)
History

“...the one single development that put computers on their feet was the invention of a reliable form of memory, namely, the core memory. Its cost was reasonable, it was reliable and, because it was reliable, it could in due course be made large.”

–Maurice Wilkes
DRAM (Dynamic Random Access Memory)

- bit stored as charge in capacitor
  - optimized for density (1 transistor for DRAM vs. 6 for SRAM)
  - capacitor discharges on a read (destructive read)
    - read is automatically followed by a write (to restore bit)
  - charge leaks away over time (not static)
    - refresh by reading/writing every bit once every 2ms (row at a time)
- access time = time to read
- cycle time = time between reads > access time
DRAM Chip Organization

- square row/column matrix
- multiplexed address lines
- internal row buffer
- operation
  - put row address on lines
  - set row address strobe (RAS)
  - read row into row buffer
  - put column address on lines
  - set column address strobe (CAS)
  - read column bits out of row buffer
  - write row buffer contents to row
- usually narrow interface (data)
Comparison with SRAM

SRAM

- optimized for speed, then density
  + 1/4–1/8 access time of DRAM
  - 1/4 density of DRAM
- bits stored as flip-flops (4-6 transistors per bit)
- static: bit not erased on a read
  + no need to refresh
  - greater power dissipated than DRAM
  + access time = cycle time
- non-multiplexed address/data lines
### DRAM Chip Specs

<table>
<thead>
<tr>
<th>Year</th>
<th>#bits</th>
<th>Access Time</th>
<th>Cycle Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1980</td>
<td>64Kb</td>
<td>150ns</td>
<td>300ns</td>
</tr>
<tr>
<td>1990</td>
<td>1Mb</td>
<td>80ns</td>
<td>160ns</td>
</tr>
<tr>
<td>1993</td>
<td>4Mb</td>
<td>60ns</td>
<td>120ns</td>
</tr>
<tr>
<td>2000</td>
<td>64Mb</td>
<td>50ns</td>
<td>100ns</td>
</tr>
<tr>
<td>2004</td>
<td>1Gb</td>
<td>45ns</td>
<td>75ns</td>
</tr>
</tbody>
</table>

- **density**: +60% annual
  - Moore’s law: density doubles every 18 months
- **speed**: %7 annual
Example: Simple Main Memory

- 32-bit wide DRAM (1 word of data at a time)
  - pretty wide for an actual DRAM
- access time: 2 cycles (A)
- transfer time: 1 cycle (T)
  - time on the bus
- cycle time: 4 cycles (B = cycle time - access time)
  - B includes time to refresh after a read
- what is the miss penalty for a 4-word block?
### Simple Main Memory

<table>
<thead>
<tr>
<th>cycle</th>
<th>addr</th>
<th>mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>T/B</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td>A</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>T/B</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>9</td>
<td>14</td>
<td>A</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>T/B</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>13</td>
<td>15</td>
<td>A</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>T/B</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>B</td>
</tr>
</tbody>
</table>

4-word access = 15 cycles
4-word cycle = 16 cycles

can we speed this up?

- lower latency?
  - no
- higher bandwidth?

© 2004 by Lebeck, Sorin, Roth, Hill, Wood, Sohi, Smith, Vijaykumar, Lipasti
Bandwidth: Wider DRAMs

<table>
<thead>
<tr>
<th>cycle</th>
<th>addr</th>
<th>mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>T/B</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
<td>A</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>T/B</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>B</td>
</tr>
</tbody>
</table>

new parameter

- 64-bit DRAMs

4-word access = 7 cycles

4-word cycle = 8 cycles

- 64-bit bus
  - wide buses (especially off-chip) are hard
  - electrical problems

- 64-bit DRAM is probably too wide

– 64-bit bus
  - wide buses (especially off-chip) are hard
  - electrical problems

– 64-bit DRAM is probably too wide
use multiple DRAMs, exploit their aggregate bandwidth

- each DRAM called a bank
  - not true: sometimes collection of DRAMs together called a bank

- M 32-bit banks

- simple interleaving: banks share address lines

- word A in bank \((A \mod \ M)\) at \((A \div \ M)\)
  - e.g., \(M=4, A=9\): bank 1, location 2

\[
\begin{array}{c|c|c|c}
\hline
0 & 1 & 2 & 3 \\
4 & 5 & 6 & 7 \\
8 & 9 & 10 & 11 \\
\hline
\end{array}
\]
Simple Interleaving

<table>
<thead>
<tr>
<th>cycle</th>
<th>addr</th>
<th>bank0</th>
<th>bank1</th>
<th>bank2</th>
<th>bank3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>T/B</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>B</td>
<td>T/B</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>T</td>
<td>B</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T</td>
</tr>
</tbody>
</table>

4-word access = 6 cycles

4-word cycle = 4 cycles
  + can start a new access in cycle 5
  + overlap access with transfer
  + and still use a 32-bit bus!
Bandwidth: Complex Interleaving

simple interleaving: banks share address lines

complex interleaving: banks are independent
  – more expensive (separate address lines for each bank)
Complex Interleaving

<table>
<thead>
<tr>
<th>cycle</th>
<th>addr</th>
<th>bank0</th>
<th>bank1</th>
<th>bank2</th>
<th>bank3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td>A</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>14</td>
<td>T/B</td>
<td>A</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td>B</td>
<td>T/B</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>B</td>
<td>T/B</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>B</td>
<td></td>
<td>T/B</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>B</td>
</tr>
</tbody>
</table>

4-word access = 6 cycles

4-word cycle = 4 cycles

• same as simple interleaving
• so why use complex interleaving?
Simple with Non-Sequential Access

what if the 4 words are *not sequential*?

- e.g., stride = 3, addrs = 12,15,18,21 (12&15 share block)
  - 4-word access = 11 cycles, 4-word cycle = 12 cycles!

<table>
<thead>
<tr>
<th>cycle</th>
<th>addr</th>
<th>bank0</th>
<th>bank1</th>
<th>bank2</th>
<th>bank3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12 (15)</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>T/B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>T/B</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>18</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>7</td>
<td>B</td>
<td>B</td>
<td>T/B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>B</td>
<td>B</td>
<td>T/B</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>21</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>11</td>
<td>B</td>
<td>T/B</td>
<td>B</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>
Complex with Non-Sequential Access

non-sequential (stride = 3) access with complex interleaving
+ 4-word access = 6, 4-word cycle = 4

aren’t all accesses sequential anyway (e.g., cache lines)?
  • DMA isn’t, vector accesses (later) aren’t
  • want more banks than words in a cache line (superbanks)
    • why? multiple cache misses in parallel (non-blocking caches)
Complex Interleaving

problem: power of 2 strides (very common)
  • e.g., same 4 banks, stride = 8, addresses = 12, 20, 28, 36
  • 4-word access = 15 cycles, 4-word cycle = 16 cycle

<table>
<thead>
<tr>
<th>cycle</th>
<th>addr</th>
<th>bank0</th>
<th>bank1</th>
<th>bank2</th>
<th>bank3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>T/B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>T/B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>B</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  • problem: all addresses map to the same bank
  • solution: use prime number of banks (BSP: 17 banks)
Interleaving Summary

banks
  + high bandwidth with a narrow (cheap) bus

superbank
  • collection of banks that make up a cache line
  + multiple superbanks = good for multiple line accesses

how many banks to “eliminate” conflicts?
  • r.o.t. answer = 2 * banks required for b/w purposes
aggressive configurations need a lot of banks

- 120ns DRAM (assume 64-bit banks)
  - processor 1: 4ns clock, no cache ⇒ 1 64-bit ref / cycle
    - at least 32 banks (64 bits/4ns ~ 64bits/120ns * 32 banks)
  - processor 2: add write-back cache ⇒ 1 64-bit ref / 4 cycles
    - at least 8 banks (64 bits/16 ns ~ 64 bits/120ns * 8 banks)

  hard to make this many banks from narrow DRAMs
  - e.g., 32 64-bit banks from 1x64Mb DRAMS ⇒ 2048 DRAMS (16GB)
  - e.g., 32 64-bit banks from 4x16Mb DRAMS ⇒ 512 DRAMS (1GB)
  - can’t force people to buy that much memory just to get bandwidth

  - use wide DRAMs (32-bit) or optimize narrow DRAMs
DRAM Optimizations

normal operation: read row into buffer, read column from buffer

observation: why not do multiple accesses from row buffer?

• nibble mode: additional bits per access (narrow DRAMs)
• page mode: change column address
• static column mode (SCRAM): don’t toggle CAS
• cached DRAMs: multiple row buffers

orthogonally: synchronous DRAMs (SDRAM)

• clock replaces RAS/CAS
  + faster
• just now becoming standard (DRAMs are commodities)
RAMBUS

a completely new memory interface [Horowitz]

- excellent engineering: transfers data both edges of clock
  - RAMBUS channel uses high-speed electrical signalling
- synchronous, no CAS/RAS
- internal caching (4–16 row buffers)
- split transaction (address queuing)
- 8-bit data
  - narrow (fix w/ multiple RAMBUS channels)
+ 2ns/byte transfer time
  - 5GB/s: very high bandwidth!

- expensive
Processor/Memory Integration

the next logical step: processor and memory on same chip

• move on-chip: FP, L2 caches, graphics. why not memory?
  – problem: processor/memory technologies incompatible
    • different number/kinds of metal layers
    • DRAM: capacitance is a good thing, logic: capacitance a bad thing

what needs to be done?

• use some DRAM area for simple processor (10% enough)
• eliminate external memory bus, milk performance from that
• integrate interconnect interfaces (processor/memory unit)
• re-examine tradeoffs: technology, cost, performance
• research projects: PIM, IRAM
address generated by program != physical memory address
Virtual Memory (VM)

**virtual**: something that appears to be there, but isn’t

**original** motivation: make more memory “appear to be there”
- physical memory expensive & not very dense ⇒ too small
+ business: common software on wide product line
  - without VM, software is sensitive to physical memory size (overlays)

**current** motivation: *use indirection in VM as a feature*
- physical memories are big now
- multiprogramming, sharing, relocation, protection
- fast start-up, sparse use
- memory mapped files, networks
Virtual Memory: The Story

• blocks called *pages*
• processes use *virtual addresses* (VA)
• physical memory uses *physical addresses* (PA)
• address divided into page offset, page number
  • virtual: virtual page number (VPN)
  • physical: page frame number (PFN)

• *address translation*: system maps VA to PA (VPN to PFN)
• e.g., 4KB pages, 32-bit machine, 64MB physical memory
  • 32-bit VA, 26-bit PA (log$_2$64MB), 12-bit page offset (log$_2$4KB)
System Maps VA To PA (VPN to PFN)

key word in that sentence? “system”

- individual processes do not perform mapping
- same VPNs in different processes map to different PFNs
  + *protection*: processes cannot use each other’s PAs
  + *programming made easier*: each process thinks it is alone
  + *relocation*: program can be run anywhere in memory
    - doesn’t have to be physically contiguous
    - can be paged out, paged back in to a different physical location

“system”: something user process can’t directly use via ISA

- OS or purely microarchitectural part of processor
Virtual Memory: The Four Questions

same four questions, different four answers

• **page placement**: fully (or very highly) associative
  • why?

• **page identification**: address translation
  • will discuss soon

• **page replacement**: sophisticated (LRU + “working set”)
  • why?

• **write strategy**: always write-back + write-allocate
  • why?
The Answer Behind the Four Answers

backing store to main memory is *disk*

- memory is 50 to 100 slower than processor
- disk is 20 to 100 *thousand* times slower than memory
  - disk is 1 to 10 *million* times slower than processor

a VA miss (VPN has no PFN) is called a *page fault*

- high cost of page fault determines design
- full associativity + OS replacement $\Rightarrow$ reduce miss rate
  - have time to let software get involved, make better decisions
- write-back reduces disk traffic
- page size usually large (4KB to 16KB) to amortize reads
## Compare Levels of Memory Hierarchy

<table>
<thead>
<tr>
<th>parameter</th>
<th>L1</th>
<th>L2</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{hit}$</td>
<td>1,2 cycles</td>
<td>5-15 cycles</td>
<td>10-150 cycles</td>
</tr>
<tr>
<td>$t_{miss}$</td>
<td>6-50 cycles</td>
<td>20-200 cycles</td>
<td>0.5-5M cycles</td>
</tr>
<tr>
<td>capacity</td>
<td>4-128KB</td>
<td>128KB-8MB</td>
<td>16MB-8GB</td>
</tr>
<tr>
<td>block size</td>
<td>8-64B</td>
<td>32-256B</td>
<td>4KB-16KB</td>
</tr>
<tr>
<td>associativity</td>
<td>1,2</td>
<td>2,4,8,16</td>
<td>full</td>
</tr>
<tr>
<td>write strategy</td>
<td>write-thru/back</td>
<td>write-back</td>
<td>write-back</td>
</tr>
</tbody>
</table>

$t_{hit}$ and $t_{miss}$ determine everything else.
Address Translation: Page Tables

OS performs address translation using a page table

- each process has its own page table
  - OS knows address of each process’ page table

- a page table is an array of page table entries (PTEs)
  - one for each VPN of each process, indexed by VPN

- each PTE contains
  - PFN
  - permission
  - dirty bit
  - LRU state
  - e.g., 4-bytes total
Page Table Size

page table size

- example #1: 32-bit VA, 4KB pages, 4-byte PTE
  - 1M pages (32 bits = 4 GB address space / 4 KB page = 1M pages)
  - 1M pages*4bytes = 4MB page table (bad, but could be worse)

- example #2: 64-bit VA, 4KB pages, 4-byte PTE
  - 4P pages, 16PB page table (not a viable option)

- upshot: can’t have page tables of this size in memory

techniques for reducing page table size

- multi-level page tables
- inverted page tables
Multi-Level Page Tables

tree of page tables

- L1 table points to L2 tables (etc.)
  - different VPN bits are offsets at different levels

+ save space: not all tables at all levels need to exist
  - exploits “sparse use” of virtual address space

- slow: multi-hop chain of translations
  + overwhelmed by space savings

- e.g., Alpha
Multi-Level Page Table Example

- 32-bit address space, 4KB pages, 4 byte PTEs (L1 & L2)
- 2 level virtual page table
- 2nd-level tables are each the size of 1 data page
- program uses only upper and lower 1MB of address space
- how much memory does page table take?
  - 4GB VM / 4KB pages ⇒ 1M pages (max)
  - 4KB pages / 4B PTEs ⇒ 1K PTEs per 2nd level table
  - 1M pages/1K PTEs per 2nd level table ⇒ 1K 2nd-level tables (max)
  - 1K 2nd level tables ⇒ 4KB first level table (assume L1 PTE = 4B)
  - 2MB VA space / 4KB pages ⇒ 512 PTEs ⇒ 1 2nd level table
  - PT = 1*1st level table (4KB) + 1*2nd level table (4KB) = 8KB!!
- remember: with 1 level PT, it was 4 MB
Inverted Page Table

**observe**: don’t need more PTEs than *physical memory pages*

- hash virtual address into array of PTEs
  - deal with collisions via chaining

+ small (proportional to memory size $<<$ VA space size)
  - page table size = (memory size / page size) * (PTE size + pointer)

– slow searches (PTE pointer chasing)
  - use extra levels of hashing to mitigate

- e.g., IBM POWER1
Mechanics of Address Translation

so how does address translation actually work?

• does *process* read page table & translate every VA to PA?
  – would be REALLY SLOW (esp. with 2-level page table)
  – is actually not allowed (implies *process* can access PAs)

• “system” performs translation & access on process behalf
  + legal from a protection standpoint

  • who is “system”?

• physical table: pointers are process PAs
  • *processor* can perform translation (Intel’s page table walker FSM)
  • page-table base register helps here

• virtual table: pointers are kernel VAs (can be paged)
  • processor or OS
Fast Translation: Virtual Caches

solution #1: first level caches (I and D) are “virtual”

- L2 and main memory are “physical”
  + address translation only on a miss (fast)
    - not popular today, but may be coming into vogue

- virtual address space changes
  - e.g., user vs. kernel, different users
  - flush caches on context switches?
  - process IDs in caches?
  - single system-wide virtual address space?

- I/O
  - only deals with physical addresses
  - flush caches on I/O?
Fast Translation: Physical Caches + TBs

solution #2: first level caches are “physical”
  • address translation before every cache access
    + no problems for I/O, address space changes & MP
      – SLOW

solution #2a: cache recent translations
  • not in I$ & D$
    • why not?
  • translation buffer (TB)
    + only go to page table on TB miss
      – still 2 serial accesses on a hit
Fast Translation: Physical Caches + TLBs

solution #3: address translation & L1 cache access in parallel!!

• *translation lookaside buffer (TLB)*
  + fast (one step access)
  + no problems changing virtual address spaces
  + can keep I/O coherent

• but...
Physical Cache with a Virtual Index?

Q: how to access a physical cache with a virtual address?

• A.1: only *cache index* matters for access
• A.2: only *part* of virtual address changes during translation
• A.3: make sure index is in untranslated part
  • index is within page offset
  • virtual index == physical index

+ fast
− restricts cache size? (block size * #sets) <= page size
  • that’s OK, use associativity to increase size (=more frames/set)
Cache + TLB Access

virtually-indexed, virtually-tagged

virtually-indexed, physically-tagged with TLB
Synonyms

sometimes, it is useful to find VA (i.e., cache set) given PA only

• bus events: MP invalidations, DMA, split transaction bus

what happens if (index+offset) > page offset?

• J VPN bits used in index (overlap of index into VPN)
  • same physical block may be in $2^J$ sets
    – impossible to know which given only PA
  • called a *synonym*: intra-cache coherence problem

• solutions
  • search all possible synonymous sets in parallel
  • restrict page placement in OS such that index(VA) == index(PA)
  • eliminate by OS convention: single shared virtual address space
More About TLBs

**TLB miss**

- entry not in TLB, but in page table (soft miss)
  - not quite a page fault (no disk access necessary)
- virtual page table: trap to OS, double penalty of TLB miss
- physical page table: processor can do it in ~30 cycles

why are there no L2 TLBs? (esp. with a physical page table)

**superpages**: variable sized pages for more TLB coverage

- want TLB to cover L2 cache contents (why?)
  - need OS support (not widely implemented)
  - restricts relocation
Protection

goal

• one process should not interfere with another

process model

• “virtual” user processes
  • must access memory through address translation
  • can’t “see” address translation mechanism itself (its own page table)

• OS kernel: a process with special privileges
  • can access memory directly (using physical addresses)
  • hence, can mess with the page tables (someone should be able to)
Protection Primitives

policy vs. mechanism

• h/w provides primitives, problems if h/w implements policy primitives

  • at least one *privileged mode*
    • some bit(s) somewhere in the processor
    • certain resources readable/writable only if processor in this mode

  • a safe facility for switching into this mode (SYSCALL)
    • can’t “call” OS (OS is another process with its own VA space)
    • user process: specifies what it wants done & return address
    • SYSCALL: user process abdicates, OS starts in privileged mode
    • return to process (switch back to unprivileged mode) not a big deal
Protection Primitives

protection bits (R,W,X,K/U) for different memory regions

- in general: base and bound registers + bits
  - check: base <= address <= bounds

- page-level protection: implicit base and bounds
  - cache protection bits in TLB for speed

- segment-level protection: explicit base and bounds
  - like variable size pages

- Intel, paged segments
  - a two-level address space (user visible segments)
  - paging underneath
  - much more
Memory Summary

main memory

- technology: DRAM (slow, but dense)
- interleaving/banking for high bandwidth
  - simple vs. complex

virtual memory, address translation & protection

- larger memory, protection, relocation, multiprogramming
- page tables
  - inverted/multi-level tables save space
- TLB: cache translations for speed
  - access in parallel with cache tags

next up: disks, buses, and I/O