Main memory management

- Questions:
  - How big should main memory be?
  - How to handle reads and writes?
  - How to find something in main memory?
  - How to decide what to put in main memory?
  - If main memory is full, how to decide what to replace?
The scale of things

• Typically (as of 2012):
  – Registers: < 1 KB, access time .25–.5 ns
  – L1 Cache: < 8 MB, access time 1–5 ns
  – Main Memory: ~4 GB, access time 10–50 ns
  – Disk Storage: ~2TB, access time 10 ms

• Memory Technology
  – CMOS (Complementary Metal Oxide Semiconductor)
    – Uses a combination of n- and p-doped semiconductor material to achieve low power dissipation.

Memory hardware

• DRAM: dynamic random access memory, typically used for main memory
  – One transistor per data bit
  – Each bit must be refreshed periodically (e.g., every 8 milliseconds), so maybe 5% of time is spent in refresh
  – Access time < cycle time
  – Address sent in two halves so that fewer pins are needed on chip (row and column access)
Memory hardware (cont.)

- SRAM: static random access, typically used for cache memory
  - 4-6 transistors per data bit
  - No need for refresh
  - Access time = cycle time
  - Address sent all at once, for speed

Bottleneck

- Main memory access will slow down the CPU unless the hardware designer is careful
- Some techniques can improve memory bandwidth, the amount of data that can be delivered from memory in a given amount of time:
  - Wider main memory
  - Interleaved memory
  - Independent memory banks
  - Avoiding memory bank conflicts
Wider main memory

- Wider cache lines
  - Cache miss: If a cache block contains k words, then each cache miss involves these steps repeated k times:
    - Send the address to main memory
    - Access the word (i.e., locate it)
    - Send the word to cache, with the bits transmitted in parallel
  - Idea behind wider memory: the user thinks about 32 bit words, but physical memory can have longer words
  - Then the operations above are done only k/n times, where n is the number of 32 bit words in a physical word

Wider main memory (cont.)

- Extra costs:
  - A wider memory bus: hardware to deliver 32n bits in parallel, instead of 32 bits
  - A multiplexer to choose the correct 32 bits to transmit from the cache to the CPU
Interleaved memory

- Partition memory into banks, with each bank able to access a word and send it to cache in parallel
- Organize address space so that adjacent words live in different banks - called **interleaving**
- For example, 4 banks might have words with the following octal addresses:

<table>
<thead>
<tr>
<th>Bank 0</th>
<th>Bank 1</th>
<th>Bank 2</th>
<th>Bank 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>02</td>
<td>03</td>
</tr>
<tr>
<td>04</td>
<td>05</td>
<td>06</td>
<td>07</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Interleaved memory (cont.)

- Note how nice interleaving is for write-through
- Also helps speed read and write-back
- **Note:** Interleaved memory acts like wide memory, except that words are transmitted through the bus sequentially, not in parallel
Independent memory banks

- Each bank of memory has its own address lines and (usually) a bus
- Can have several independent banks: perhaps
  - One for instructions
  - One for data
  - This is called a Harvard architecture
- Banks can operate independently without slowing others

Avoid memory bank conflicts

- By having a prime number of memory banks
- Since arrays frequently have even dimension sizes - and often dimension sizes that are a power of 2 - strides that match the number of banks (or a multiple) give very slow access
Example: Interleaving

- First access the first column of $x$:
  - $x[0][0]$, $x[1][0]$, $x[2][0]$, ..., $x[255][0]$

- With addresses
  - $K$, $K+512*4$, $K+512*8$, ...
  - $K+512*4*255$

- With 4 memory banks, all of the elements live in the same memory bank, so the CPU will stall in the worst possible way

```c
int x[256][512];
for (j=0; j<512; j=j+1)
  for (i=0; i<256; i=i+1)
    x[i][j] = 2 * x[i][j];
```

How much good do these techniques do?

- Example: Assume a cache block of 4 words, and
  - 4 cycles to send address to main memory
  - 24 cycles to access a word, once the address arrives
  - 4 cycles to send a word back to cache

- Basic miss penalty: $4*32 = 128$ cycles, since each of 4 words has the full 32 cycle penalty
- Memory with a 2-word width: $2*(32+4) = 72$ cycle miss penalty
- Simple interleaved memory: address can be sent to each bank simultaneously, so miss penalty is $4 + 24 + 4*4$ (for sending words) = 44 cycles
- Independent memory banks: 32 cycle miss penalty, as long as the words are in different banks, since each has its own address lines and bus
**Virtual addressing**

- Computers are designed so that multiple programs can be active at the same time.
- At the time a program is compiled, the compiler has to assign addresses to each data item. But how can it know what memory addresses are being used by other programs?
- Instead, the compiler assigns virtual addresses, and expects the loader/OS to provide the means to map these into physical addresses.

**Memory protection**

- Each program “lives” in its own virtual space, called its process.
- When the CPU is working on one process, others may be partially completed or waiting for attention.
- The CPU is time shared among the processes, working on each in turn.
- And main memory is also shared among processes.
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.

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.
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

Virtual addressing

[Diagram showing virtual addresses, address translation, physical addresses, and physical memory, with a hard drive]
## Virtual memory

**Diagram**: A diagram illustrating virtual memory addressing. It shows a virtual memory space with blocks labeled A through D and physical memory blocks with corresponding addresses. The diagram also includes a disk and a main memory section with physical addresses.

## Parameters

<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>50-150 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
- 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

![Diagram of segmentation]

Each segment can be located anywhere in physical memory
The limits of physical addressing

"Physical addresses" of memory locations

All programs share one address space: The physical address space

Machine language programs must be aware of the machine organization

No way to prevent a program from accessing any machine resource
Solution: Add a layer of indirection

```
<table>
<thead>
<tr>
<th>CPU</th>
<th>Physical Address Translation</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0-A31</td>
<td></td>
<td>A0-A31</td>
</tr>
<tr>
<td>D0-D31</td>
<td></td>
<td>D0-D31</td>
</tr>
</tbody>
</table>
```

User programs run in an standardized virtual address space.

**Address Translation** hardware managed by the operating system (OS) maps virtual address to physical memory.

Hardware supports “modern” OS features: Protection, Translation, Sharing.

---

**Paging overview**

- Fully-associative mapping, because page faults are really, really expensive.
- Page is located using a page table, one entry per page in the virtual address space.
  - Size is sometimes reduced by hashing, to make one entry per physical page in main memory – an inverted page table.
- Since locality says that a page will be used multiple times, address translation usually tests the addresses of the recently referenced pages before looking in other places.
- So address translation information is held in the translation look-aside buffer (TLB).
Paging overview (cont.)

- Most machines replace the LRU page
  - Moving pages between memory and disk is so slow that it’s worth doing something close to real LRU
- Disks are so slow that machines use write-back, not write-through, and keep a dirty bit for each page

Advantages of paged virtual memory

- Translation
  - Program can be given consistent view of memory, even though physical memory is scrambled
  - Only the most important part of program (“Working Set”) must be in physical memory.
  - Contiguous structures (like stacks) use only as much physical memory as necessary yet still grow later.
Advantages of paged virtual memory

- Protection
  - Different threads (or processes) protected from each other.
  - Different pages can be given special behavior (Read Only, Invisible to user programs, etc).
  - Kernel data protected from User programs
    » Very important for protection from malicious programs
- Sharing
  - Can map same physical page to multiple users (“Shared memory”)

Page tables encode virtual address spaces

A virtual address space is divided into blocks of memory called pages

A machine usually supports pages of a few sizes (MIPS R4000):

<table>
<thead>
<tr>
<th>Page Size</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 Kbytes</td>
<td>4 KiB</td>
</tr>
<tr>
<td>16 Kbytes</td>
<td>16 KiB</td>
</tr>
<tr>
<td>64 Kbytes</td>
<td>64 KiB</td>
</tr>
<tr>
<td>256 Kbytes</td>
<td>256 KiB</td>
</tr>
<tr>
<td>1 MiByte</td>
<td>1 MiB</td>
</tr>
<tr>
<td>4 MiBytes</td>
<td>4 MiB</td>
</tr>
<tr>
<td>16 MiBytes</td>
<td>16 MiB</td>
</tr>
</tbody>
</table>

A valid page table entry encodes physical memory “frame” address for the page
Page tables

- Page table maps virtual page numbers to physical frames
  - "PTE" = Page Table Entry
- Virtual memory => treat memory = cache for disk

Details of page table
Two-level page tables

Each process needs its own address space!

Two-level Page Tables

32 bit virtual address

Top-level table *wired* in main memory

Subset of 1024 second-level tables in main memory; rest are on disk or unallocated

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

Page Table

<table>
<thead>
<tr>
<th>Dirty bit: page written.</th>
<th>1 0 ***</th>
<th>1 0</th>
<th>0 1</th>
<th>1 1</th>
<th>0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Used bit: set to 1 on any reference</td>
<td>1     0</td>
<td>0 1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Set of all pages in Memory

Tail pointer: Clear the used bit in the page table

Hardware Architect's role: support setting dirty and used bits

Head pointer
Place pages on free list if used bit is still clear. Schedule pages with dirty bit set to be written to disk.

Freelist
Free Pages

MIPS address translation: How does it work?

“Virtual Addresses”

A0-A31
CPU
00-031
Data

“Physical Addresses”

Virtual
Translation Look-Aside Buffer (TLB)
Physical
Translation Look-Aside Buffer (TLB)
A small fully-associative cache of mappings from virtual to physical addresses

Translation Look-Aside Buffer (TLB)
A small fully-associative cache of mappings from virtual to physical addresses

TLB also contains protection bits for virtual address

Fast common case: Virtual address is in TLB, process has permission to read/write it.
Virtual Address

V page no. offset

Physical frame address

Physical Address

V=0 pages either reside on disk or have not yet been allocated. OS handles V=0 "Page fault"

The TLB caches page table entries

Common organization

Even a cache hit requires TLB translation first!
Can TLB and caching be overlapped?

This 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 as long as the address bits used to index into the cache do not change as the result of VA translation.

This usually limits things to small caches, large page sizes, or high n-way set associative caches if you want a large cache.

Example: suppose everything the same except that the cache is increased to 8 K bytes instead of 4 K:

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

Solutions:
go to 8K byte page sizes;go to 2 way set associative cache; orSW guarantee VA[13]=PA[13]
**Use virtual addresses for cache?**

- **“Virtual Addresses”**
  - CPU
  - Virtual Cache D0-D31

- **“Physical Addresses”**
  - Main Memory
  - Physical Translation Look-Aside Buffer (TLB)

Only use TLB on a cache miss!

**Downside: a subtle, fatal problem. What is it?**

**A. Synonym problem.** If two 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 1: 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 2: Caches

- The Principle of Locality:
  - Program accesses a relatively small portion of the address space in any short interval of time.
    » Temporal Locality: Locality in Time
    » Spatial Locality: Locality in Space
- Three Major Categories of Cache Misses:
  - Compulsory Misses: cold start misses
  - Capacity Misses: increase cache size
  - Conflict Misses: increase cache size and/or associativity. Nightmare Scenario: ping pong effect!
- Write Policy: Write Through vs. Write Back
- Today CPU time is a function of (ops, cache misses) vs. just f(ops): affects Compilers, Data structures, and Algorithms
Summary 3: TLB, virtual memory

- Page tables map virtual address to physical address
- TLBs are important for fast translation
- TLB misses are significant in processor performance
  - Funny times, as most systems can’t access all of 2nd level cache without TLB misses!

Summary 3: TLB, virtual memory (cont.)

- Caches, TLBs, Virtual Memory all understood by examining how they deal with 4 questions:
  - Where can block be placed?
  - How is block found?
  - What block is replaced on 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