Main memory management (review)

- 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?

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
- Extra costs:
  - a wider memory bus: hardware to deliver 32\( n \) bits in parallel, instead of 32 bits
  - a multiplexor 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

```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];
```

• 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

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 Memory – Fig. C.18

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

Parameters – Fig. C.19

<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>$(.8-8)\times10^6$ cc</td>
</tr>
<tr>
<td>(transfer time)</td>
<td>(2-40 cc)</td>
<td>$(.2-2)\times10^6$ 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>
Who can change them?

- The operating system needs access to the base and bound registers
- So a process that is labeled \textit{kernel} (also called \textit{supervisor} or \textit{executive}) can access any memory location and change the registers
- Kernel processes are accessed through \textit{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

Segmentation

- Basically multiple base&bounds
  - w/ virtual memory

The Limits of Physical Addressing

“Physical addresses” of memory locations

All programs share one address space:
The \textit{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

“Virtual Addresses”

A0-A31

CPU

D0-D31

Data

Address Translation

Virtual

Physical

“Physical Addresses”

A0-A31

Memory

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)

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

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.
- 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 valid page table entry encodes physical memory “frame” address for the page.

Details of Page Table

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

Two-Level Page Tables

Each process needs its own address space!

- [Diagram: 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