Last time in Lecture 7

- Multi-level cache hierarchies reduce miss penalty
  - 3 levels common in modern systems (some have 4!)
  - Can change design tradeoffs of L1 cache if known to have L2
  - Inclusive versus exclusive cache hierarchies

- Reducing impact of associativity
  - way-predicting caches
  - victim caches
  - (microtags in problem set)

- Prefetching, hardware or software
  - correctness, timeliness
  - instructions easier to prefetch than data
  - software difficult to use ideally

- Software memory hierarchy optimizations
  - Loop interchange
  - Loop fusion
  - Cache tiling
In a bare machine, the only kind of address is a physical address, corresponding to address lines of actual hardware memory.
Managing Memory in Bare Machines

- Early machines only ran one program at a time, with this program having unrestricted access to all memory and all I/O devices
  - This simple memory management model was also used in turn by the first minicomputer and first microcomputer systems

- Subroutine libraries became popular, were written in location-independent form
  - Different programs use different combination of routines

- To run program on bare machines, use linker or loader program to relocate library modules to actual locations in physical memory
Dynamic Address Translation

- Motivation
  - In early machines, I/O was slow and each I/O transfer involved the CPU (programmed I/O)
  - Higher throughput possible if CPU and I/O of 2 or more programs were overlapped, how?
    → multiprogramming with DMA I/O devices, interrupts

- Location-independent programs
  - Programming and storage management ease
    → need for a base register

- Protection
  - Independent programs should not affect each other inadvertently
    → need for a bound register

- Multiprogramming drives requirement for resident supervisor software to manage context switches between multiple programs
Base and bounds registers are visible/accessible only when processor is running in the *supervisor mode*
Separate Areas for Program and Data
(Scheme used on all Cray vector supercomputers prior to X1, 2002)

Load X

Program Address Space

Data Bound Register

≥

Logical Address

Data Base Register

+

Physical Address

Physical Memory

Data Segment

≥

Bounds Violation?

Program Bound Register

≥

Logical Address

Program Counter

+

Physical Address

Program Segment

What is an advantage of this separation?
What about more base/bound pairs?
Can fold addition of base register into (register+immediate) address calculation using a carry-save adder (sums three numbers with only a few gate delays more than adding two numbers)
External Fragmentation with Segments

Job 1
32K

Job 2
24K

Job 3
64K

72K

Job 1
32K

Job 2
24K

24K

Job 3
64K

8K

Job 3
starts

Job 2
finishes

Can’t run Job 4, as not enough contiguous space. Must compact.

Job 4
32K

Job 4
arrives
Paged Memory Systems

- Program-generated (*virtual* or *logical*) address split into:

<table>
<thead>
<tr>
<th>Page Number</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

- Page Table contains physical address of start of each fixed-sized page in virtual address space

- Paging makes it possible to store a large contiguous virtual memory space using non-contiguous physical memory pages
Paging Simplifies Allocation

- Fixed-size pages can be kept on OS free list and allocated as needed to any process
- Process memory usage can easily grow and shrink dynamically
- Paging suffers from "internal fragmentation" where not all bytes on a page are used
  - Much less of an issue than external fragmentation or compaction for common page sizes (4-8KB)
  - But one reason that many oppose move to larger page sizes
Simple linear page tables are too large, so hierarchical page tables are commonly used (see later).

Common for modern OS to place page tables in kernel’s virtual memory (page tables can be swapped to secondary storage).
Coping with Limited Primary Storage

- Paging reduces fragmentation, but still many problems would not fit into primary memory, have to copy data to and from secondary storage (drum, disk)

- Two early approaches:
  - **Manual overlays**, programmer explicitly copies code and data in and out of primary memory
    - Tedious coding, error-prone (jumping to non-resident code?)
  - **Software interpretive coding** (Brooker 1960). Dynamic interpreter detects variables that are swapped out to drum and brings them back in
    - Simple for programmer, but inefficient

*Not just ancient black art, e.g., IBM Cell microprocessor using in Playstation-3 had explicitly managed local store! Many new “deep learning” accelerators have similar structure.*
Demand Paging in Atlas (1962)

“A page from secondary storage is brought into the primary storage whenever it is (implicitly) demanded by the processor.”

Tom Kilburn

Primary memory as a cache for secondary memory

User sees $32 \times 6 \times 512$ words of storage
Hardware Organization of Atlas

Effective Address

1. Initial Address Decode

2. PARs

3. 16 ROM pages
   0.4-1 µsec
   system code (not swapped)

4. 2 subsidiary pages
   1.4 µsec
   system data (not swapped)

5. Main
   32 pages
   1.4 µsec

6. Drum (4)
   192 pages

7. 8 Tape decks
   88 sec/word

8. 48-bit words
9. 512-word pages

1 Page Address Register (PAR) per page frame

<effective PN, status>

Compare the effective page address against all 32 PARs

- match ⇒ normal access
- no match ⇒ page fault
  save the state of the partially executed instruction
Atlas Demand-Paging Scheme

On a page fault:

- Input transfer into a free page is initiated
- The Page Address Register (PAR) is updated
- If no free page is left, a page is selected to be replaced (based on usage)
- The replaced page is written on the drum
  - to minimize drum latency effect, the first empty page on the drum was selected
- The page table is updated to point to the new location of the page on the drum
CS152 Administrivia

- Lab 2 out on Friday in Section
- PS2 due on Wednesday Feb 27
- Midterm in class Monday March 4
  - Covers lectures 1 – 9, plus assigned problem sets, labs, book readings
CS252 Administrivia

- Project Proposal due Wednesday Feb 27th
- Proposal should be one page PDF including:
  - Title
  - Team member names
  - What are you trying to do?
  - How is it done today?
  - What is your idea for improvement and why do you think you’ll be successful
  - What infrastructure are you going to use for your project?
  - Project timeline with milestones
- Mail PDF of proposal to instructors
- Give a <5-minute presentation in class in discussion section time on March 11th
Size of Linear Page Table

- With 32-bit addresses, 4-KB pages & 4-byte PTEs:
  - 220 PTEs, i.e., 4 MB page table per user
  - 4 GB of swap needed to back up full virtual address space

- Larger pages?
  - Internal fragmentation (Not all memory in page is used)
  - Larger page fault penalty (more time to read from disk)

- What about 64-bit virtual address space???
  - Even 1MB pages would require 244 8-byte PTEs (35 TB!)

What is the “saving grace”??
Hierarchical Page Table

Virtual Address from CPU

31  22  21  12  11  0

p1  p2  offset

10-bit 10-bit
L1 index  L2 index

Root of Current Page Table

(Processor Register, satp in RISC-V)

Level 1 Page Table

p1

Level 2 Page Tables

RISC-V Sv32 Virtual Memory Scheme
Two-Level Page Tables in Physical Memory

Virtual Address Spaces

User 1

User 2

Virtual Address
Spaces

VA1

Level 1 PT
User 1

Level 1 PT
User 2

Level 2 PT
User 2

User2/VA1

User1/VA1

Physical Memory
• Every instruction and data access needs address translation and protection checks

A good VM design needs to be fast (~ one cycle) and space efficient
Translation-Lookaside Buffers (TLB)

Address translation is very expensive!
In a two-level page table, each reference becomes several memory accesses

Solution: *Cache translations in TLB*

- **TLB hit**  ⇒ *Single-Cycle Translation*
- **TLB miss**  ⇒ *Page-Table Walk to refill*

![Diagram of TLB and page table](chart)

- **VPN** = virtual page number
- **PPN** = physical page number
- **offset**

---

*VPN*  virtual address

*PPN*  physical address

hit?  virtual address

- **VRWD**
- **tag**
- **PPN**

- **VPN**
- **offset**

- **VPN**
- **offset**

- **VPN**
- **offset**
TLB Designs

- Typically 32-128 entries, usually fully associative
  - Each entry maps a large page, hence less spatial locality across pages ➔ more likely that two entries conflict
  - Sometimes larger TLBs (256-512 entries) are 4-8 way set-associative
  - Larger systems sometimes have multi-level (L1 and L2) TLBs

- Random or FIFO replacement policy

- TLB Reach: Size of largest virtual address space that can be simultaneously mapped by TLB
  - Example: 64 TLB entries, 4KB pages, one page per entry

  - TLB Reach = 
    \[ 64 \text{ entries} \times 4 \text{ KB} = 256 \text{ KB} \] (if contiguous)
Handling a TLB Miss

- **Software (MIPS, Alpha)**
  - TLB miss causes an exception and the operating system walks the page tables and reloads TLB. A privileged “untranslated” addressing mode used for walk.
  - Software TLB miss can be very expensive on out-of-order superscalar processor as requires a flush of pipeline to jump to trap handler.

- **Hardware (SPARC v8, x86, PowerPC, RISC-V)**
  - A memory management unit (MMU) walks the page tables and reloads the TLB.
  - If a missing (data or PT) page is encountered during the TLB reloading, MMU gives up and signals a Page Fault exception for the original instruction.

- **NOTE:** A given ISA can use either TLB miss strategy
MMU does this table walk in hardware on a TLB miss
Assumes page tables held in untranslated physical memory
Page Fault Handler

- When the referenced page is not in DRAM:
  - The missing page is located (or created)
  - It is brought in from disk, and page table is updated
    - Another job may be run on the CPU while the first job waits for the requested page to be read from disk
  - If no free pages are left, a page is swapped out
    - Pseudo-LRU replacement policy, implemented in software

- Since it takes a long time to transfer a page (msecs), page faults are handled completely in software by OS
  - Untranslated addressing mode is essential to allow kernel to access page tables

- Keeping TLBs coherent with page table changes might require expensive “TLB shootdown”
  - Interrupt other processors to invalidate stale TLB entries
  - Some mainframes had hardware TLB coherence
Handling VM-related exceptions

- Handling a TLB miss needs a hardware or software mechanism to refill TLB.
- Handling page fault (e.g., page is on disk) needs *restartable* exception so software handler can resume after retrieving page.
  - Precise exceptions are easy to restart.
  - Can be imprecise but restartable, but this complicates OS software.
- A protection violation may abort process.
  - But often handled the same as a page fault.
Acknowledgements

- This course is partly inspired by previous MIT 6.823 and Berkeley CS252 computer architecture courses created by my collaborators and colleagues:
  - Arvind (MIT)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)