Last time in Lecture 8

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

- Non-blocking caches
  - Allow hits and maybe misses while misses in flight

- Prefetching: retrieve data from memory before CPU request
  - Prefetching can waste bandwidth and cause cache pollution
  - Software vs hardware prefetching

- Software memory hierarchy optimizations
  - Loop interchange, loop fusion, cache tiling
Memory Management

• From early absolute addressing schemes, to modern virtual memory systems with support for virtual machine monitors

• Can separate into orthogonal functions:
  – Translation (mapping of virtual address to physical address)
  – Protection (permission to access word in memory)
  – Virtual memory (transparent extension of memory space using slower disk storage)

• But most modern systems provide support for all the above functions with a single page-based system
Absolute Addresses

EDSAC, early 50’s

- Only one program ran at a time, with unrestricted access to entire machine (RAM + I/O devices)
- Addresses in a program depended upon where the program was to be loaded in memory
- *But* it was more convenient for programmers to write location-independent subroutines

**How could location independence be achieved?**

*Linker and/or loader modify addresses of subroutines and callers when building a program memory image*
Dynamic Address Translation

Motivation

In the early machines, I/O operations were slow and each word transferred involved the CPU.

Higher throughput if CPU and I/O of 2 or more programs were overlapped. 

How? ⇒ multiprogramming

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
Simple Base and Bound Translation

Load X

Program Address Space

Bound Register

≤

Bounds Violation?

Segment Length

Effective Address

Base Register

Base Physical Address

Physical Address

current segment

Physical Memory

Base and bounds registers are visible/accessible only when processor is running in the supervisor mode
Separate Areas for Program and Data

What is an advantage of this separation? (Scheme used on all Cray vector supercomputers prior to X1, 2002)
Memory Fragmentation

As users come and go, the storage is “fragmented”. Therefore, at some stage programs have to be moved around to compact the storage.
Paged Memory Systems

• Processor-generated address can be split into:

<table>
<thead>
<tr>
<th>page number</th>
<th>offset</th>
</tr>
</thead>
</table>

• A page table contains the physical address of the base of each page:

Page tables make it possible to store the pages of a program non-contiguously.
Private Address Space per User

- Each user has a page table
- Page table contains an entry for each user page
Where Should Page Tables Reside?

• Space required by the page tables (PT) is proportional to the address space, number of users, ...
  ⇒ Space requirement is large
  ⇒ Too expensive to keep in registers

• Idea: Keep PTs in the main memory
  – needs one reference to retrieve the page base address and another to access the data word
  ⇒ doubles the number of memory references!
Page Tables in Physical Memory

User 1 Virtual Address Space

User 2 Virtual Address Space

Physical Memory
CS152 Administrivia
A Problem in the Early Sixties

• There were many applications whose data could not fit in the main memory, e.g., payroll
  – *Paged memory system reduced fragmentation but still required the whole program to be resident in the main memory*

• Programmers moved the data back and forth from the secondary store by *overlaying* it repeatedly on the primary store

  *tricky programming!*
Manual Overlays

- Assume an instruction can address all the storage on the drum

- *Method 1*: programmer keeps track of addresses in the main memory and initiates an I/O transfer when required
  - Difficult, error-prone!

- *Method 2*: automatic initiation of I/O transfers by software address translation
  - Brooker’s interpretive coding, 1960
  - Inefficient!

*Not just an ancient black art, e.g., IBM Cell microprocessor using in Playstation-3 has explicitly managed local store!*
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 → Initial Address Decode →
16 ROM pages 0.4 ~1 μsec
2 subsidiary pages 1.4 μsec
Main 32 pages 1.4 μsec
Drum (4) 192 pages

 system code (not swapped)
 system data (not swapped)

48-bit words
512-word pages

1 Page Address Register (PAR) per page frame

Compare the effective page address against all 32 PARs
match ⇒ normal access
no match ⇒ page fault
save the state of the partially executed instruction

<effective PN, status>
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
Caching vs. Demand Paging

**Caching**
- cache entry
- cache block (~32 bytes)
- cache miss rate (1% to 20%)
- cache hit (~1 cycle)
- cache miss (~100 cycles)
- a miss is handled in *hardware*

**Demand paging**
- page frame
- page (~4K bytes)
- page miss rate (<0.001%)
- page hit (~100 cycles)
- page miss (~5M cycles)
- a miss is handled mostly in *software*
Modern Virtual Memory Systems

*Illusion of a large, private, uniform store*

Protection & Privacy
several users, each with their private
address space and one or more
shared address spaces
page table = name space

Demand Paging
Provides the ability to run programs
larger than the primary memory

Hides differences in machine
configurations

*The price is address translation on
each memory reference*
Linear Page Table

- Page Table Entry (PTE) contains:
  - A bit to indicate if a page exists
  - PPN (physical page number) for a memory-resident page
  - DPN (disk page number) for a page on the disk
  - Status bits for protection and usage

- OS sets the Page Table Base Register whenever active user process changes
Size of Linear Page Table

With 32-bit addresses, 4-KB pages & 4-byte PTEs:

⇒ $2^{20}$ 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 $2^{44}$ 8-byte PTEs (35 TB!)

What is the “saving grace”?
Hierarchical Page Table

Virtual Address

31 22 21 12 11 0

p1  p2  offset

10-bit 10-bit
L1 index L2 index

Root of the Current Page Table

(Processor Register)

Level 1 Page Table

p1

Level 2 Page Tables

p2

offset

Physical Memory

Data Pages

page in primary memory
page in secondary memory
PTE of a nonexistent page
Address Translation & Protection

- 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

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

Solution: *Cache translations in TLB*
- TLB hit $\Rightarrow$ *Single Cycle Translation*
- TLB miss $\Rightarrow$ *Page-Table Walk to refill*

![Diagram of TLB and virtual to physical address translation]

- **VPN**: virtual page number
- **PPN**: physical page number

(VM = virtual memory)
**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

- No process information in TLB?

- 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

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

Hardware (SPARC v8, x86, PowerPC)
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
Hierarchical Page Table Walk: SPARC v8

Virtual Address

Context Table Register

Context Table

Index 1

31

Index 2

23

Index 3

17

Offset

11

0

L1 Table

PTP

L2 Table

PTP

L3 Table

PTE

Physical Address

PPN

Offset

MMU does this table walk in hardware on a TLB miss
Address Translation: putting it all together

Virtual Address

TLB Lookup

hit

Page Table Walk

miss

Page Fault (OS loads page)

Update TLB

Protection Check

protected

denied

permitted

Physical Address (to cache)

Where?

SEGFAULT
Acknowledgements

• These slides contain material developed and copyright by:
  – Arvind (MIT)
  – Krste Asanovic (MIT/UCB)
  – Joel Emer (Intel/MIT)
  – James Hoe (CMU)
  – John Kubiatowicz (UCB)
  – David Patterson (UCB)

• MIT material derived from course 6.823
• UCB material derived from course CS252