Address Translation and Virtual Memory
CENG331 - Computer Organization

Instructor:
Murat Manguoglu (Sections 1-2)
Fall 2019

Adapted from: http://inst.eecs.berkeley.edu/~cs152
In a bare machine, the only kind of address is a physical address.
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 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)

What is an advantage of this separation?
Base and Bound Machine

[ 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) ]
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:
  
  ![Diagram of page number and offset]

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

  ![Diagram of address space, page table, and physical memory]

Page tables make it possible to store the pages of a program non-contiguously.
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, ...
  \(\Rightarrow\) *Too large 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
  \(\Rightarrow\) *doubles the number of memory references!*
Page Tables in Physical Memory

User 1 Virtual Address Space

User 2 Virtual Address Space

Physical Memory
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*
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

Effectve 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

- 8 Tape decks 88 sec/word

48-bit words
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
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
Two-Level Page Tables in Physical Memory

Virtual Address Spaces

User 1

User 2

Physical Memory

Level 1 PT
User 1

Level 1 PT
User 2

User2/VA1

User1/VA1

Level 2 PT
User 2

VA1

VA1
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** $\Rightarrow$ *Single-Cycle Translation*
- **TLB miss** $\Rightarrow$ *Page-Table Walk to refill*

<table>
<thead>
<tr>
<th>VR WD</th>
<th>tag</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*(VPN = virtual page number)*

*(PPN = physical page number)*

hit?  

virtual address

physical address

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

• 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

TLB Reach = _______________
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, 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
Hierarchical Page Table Walk: SPARC v8

Virtual Address

Context Table Register

Context Table

root ptr

Context Register

Context Table

L1 Table

PTP

L2 Table

PTP

L3 Table

PTP

PTE

Physical Address

PPN

Offset

MMU does this table walk in hardware on a TLB miss
**Page-Based Virtual-Memory Machine**

(Hardware Page-Table Walk)

- Assumes page tables held in untranslated physical memory
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 \(\equiv\) 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*
Address Translation: putting it all together

Virtual Address

TLB Lookup

Restart instruction

Page Table Walk

Page Fault (OS loads page)

Update TLB

Protection Fault

Protection Check

Hardware or software

Hardware

Physical Address (to cache)

SEGFAULT

miss

hit

≠ memory

∈ memory

denied

permitted

the page is

≠ memory

∈ memory

(OS loads page)

Update TLB

Restart instruction

Protection Fault

 SEGFAULT
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*

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

- Handling a TLB miss needs a hardware or software mechanism to refill TLB
- Handling a page fault (e.g., page is on disk) needs a 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
- Handling protection violation may abort process
  - But often handled the same as a page fault
Address Translation in CPU Pipeline

- Need to cope with additional latency of TLB:
  - slow down the clock?
  - pipeline the TLB and cache access?
  - virtual address caches
  - parallel TLB/cache access
Virtual-Address Caches

- one-step process in case of a hit (+)
- cache needs to be flushed on a context switch unless address space identifiers (ASIDs) included in tags (-)
- aliasing problems due to the sharing of pages (-)
- maintaining cache coherence (-) (see later in course)

Alternative: place the cache before the TLB (StrongARM)
Virtually Addressed Cache (Virtual Index/Virtual Tag)

Translate on miss
VM features track historical uses:

• Bare machine, only physical addresses
  – One program owned entire machine

• Batch-style multiprogramming
  – Several programs sharing CPU while waiting for I/O
  – Base & bound: translation and protection between programs (not virtual memory)
  – Problem with external fragmentation (holes in memory), needed occasional memory defragmentation as new jobs arrived

• Time sharing
  – More interactive programs, waiting for user. Also, more jobs/second.
  – Motivated move to fixed-size page translation and protection, no external fragmentation (but now internal fragmentation, wasted bytes in page)
  – Motivated adoption of virtual memory to allow more jobs to share limited physical memory resources while holding working set in memory

• Virtual Machine Monitors
  – Run multiple operating systems on one machine
  – Idea from 1970s IBM mainframes, now common on laptops
    » e.g., run Windows on top of Mac OS X
  – Hardware support for two levels of translation/protection
    » Guest OS virtual -> Guest OS physical -> Host machine physical
Virtual Memory Use Today - 1

- Servers/desktops/laptops/smartphones have full demand-paged virtual memory
  - Portability between machines with different memory sizes
  - Protection between multiple users or multiple tasks
  - Share small physical memory among active tasks
  - Simplifies implementation of some OS features

- Vector supercomputers have translation and protection but rarely complete demand-paging
  - (Older Crays: base&bound, Japanese & Cray X1/X2: pages)
    - Don’t waste expensive CPU time thrashing to disk (make jobs fit in memory)
    - Mostly run in batch mode (run set of jobs that fits in memory)
    - Difficult to implement restartable vector instructions
Most embedded processors and DSPs provide physical addressing only

- Can’t afford area/speed/power budget for virtual memory support
- Often there is no secondary storage to swap to!
- Programs custom written for particular memory configuration in product
- Difficult to implement restartable instructions for exposed architectures
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