Lecture 13: Virtual Memory

Vassilis Papaefstathiou
Iakovos Mavroidis

Computer Science Department
University of Crete
Outline

- Virtual Memory
  - Basics
  - Address Translation
  - Cache vs VM
  - Paging
  - Replacement
  - TLBs
  - Segmentation
  - Page Tables
Memory Hierarchy

**CPU Registers**
- 100s Bytes
- <10ns ns

**Cache**
- K Bytes
- 10-100 ns
- 1-0.1 cents/bit

**Main Memory**
- M Bytes
- 200ns-500ns
- $.0001-.00001 cents/bit

**Disk**
- G Bytes, 10 ms
- (10,000,000 ns)
- -5 -6
- 10 - 10 cents/bit

**Tape**
- infinite sec-min
- -8 10

**Capacity - Access Time - Cost**

**Upper Level**
- Faster

**Lower Level**
- Larger

**Staging Xfer Unit**
- prog./compiler
- 1-8 bytes
- cache cntl
- 8-128 bytes
- OS
- 4K-16K bytes
- user/operator
- Mbytes

**Registers**
- Instr. Operands

**Cache**
- Blocks

**Memory**
- Pages

**Disk**
- Files

**Tape**
- Files
Simple View of Memory

- Single program runs at a time
- Code and static data are at fixed locations
  - code starts at fixed location, e.g., 0x100
  - subroutines may be at fixed locations (absolute jumps)
- Data locations may be **wired** into code
- Stack accesses relative to stack pointer.
Running Two Programs (Relocation)
No Protection

- Need to relocate *logical* addresses to *physical* locations
- Stack is already relocatable
  - all accesses relative to SP
- Code can be made relocatable
  - allow only relative jumps
  - all accesses relative to PC
- Data segment
  - can calculate all addresses relative to a DP
    - expensive
  - faster with hardware support
    - base register
Virtual Memory

• Some facts of computer life…
  – Computers run lots of processes simultaneously
  – No full address space of memory for each process
  – Must share smaller amounts of physical memory among many processes

• Virtual memory is the answer!
  – Divides physical memory into blocks (physical pages), assigns them to different processes
  – Virtual memory (VM) allows main memory (DRAM) to act like a cache for secondary storage (magnetic disk).
  – VM address translation provides a mapping from the virtual address of the processor to the physical address in main memory or on disk.
Three Advantages of Virtual Memory

• Translation:
  – Program can be given consistent view of memory, even though physical memory is scrambled
  – Makes multithreading reasonable (now used a lot!)
  – 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”)

Protection with Virtual Memory

• Virtual memory allows protection without the requirement that pages be pre-allocated in contiguous chunks

• Physical pages are allocated based on program needs and physical pages belonging to different processes may be adjacent – efficient use of memory

• Each page has certain read/write properties for user/kernel that is checked on every access
  - a program’s executable can not be modified
  - part of kernel data cannot be modified/read by user
  - page tables can be modified by kernel and read by user
Basics

- Programs reference “virtual” addresses in a non-existent memory
  - These are then translated into real “physical” addresses
  - Virtual address space may be bigger than physical address space
- Divide physical memory into blocks, called pages
  - Anywhere from 512B to 16MB (4k typical)
- Virtual-to-physical translation by indexed table lookup
  - Add another cache for recent translations (the TLB)
- Invisible to the programmer
  - Looks to your application like you have a lot of memory!
A Load to Virtual Memory

- **Translate from virtual space to physical space**
  - $VA \Rightarrow PA$
  - *May need to go to disk*
VM: Page Mapping

- Process 1’s Virtual Address Space
- Process 2’s Virtual Address Space
- Physical Memory
- Page Frames
- Disk
Virtual Address Translation

- Main Memory = 1 GB
- Page Size = 4KB
- VPN = 52 bits
- PPN = 18 bits

- Translation table
  - aka “Page Table”
Virtual Address Translation

Virtual address translation involves converting a virtual address into a physical address. The process includes:

1. **Virtual Address**: The address in virtual memory.
2. **Page Table Register**: Contains the index of the page table entry.
3. **Virtual Page Number**: The page number in the virtual memory.
4. **Page Offset**: The offset within the page.
5. **Physical Page Number**: The page number in physical memory.
6. **Page Table**: This contains entries for each page in physical memory.
7. **Valid**: Indicates whether the page is present in memory.
8. **Physical Address**: The final address in physical memory.

If the page is not present in memory, a page fault occurs.
Cache terms vs. VM terms

So, some definitions/“analogies”

- A “page” or “segment” of memory is analogous to a “block” in a cache
- A “page fault” or “address fault” is analogous to a cache miss

so, if we go to main memory and our data isn’t there, we need to get it from disk…

![Diagram of memory management process](image_url)
Page Fault

- What happens when a process references a virtual address in a page that has been evicted (or never loaded)?
  - when the page was evicted, the OS set the PTE as invalid and noted the disk location of the page in a data structure (that looks like a page table but holds disk addresses)
  - when a process tries to access the page, the invalid PTE will cause an exception (page fault) to be thrown
    - OK, it’s actually an interrupt!
  - the OS will run the page fault handler in response
    - handler uses the “like a page table” data structure to locate the page on disk
    - handler reads page into a physical frame, updates PTE to point to it and to be valid
    - OS restarts the faulting process
    - there are a million and one details …
      for example: demand paging loads pages into memory only as they are needed by the process
Virtual Address Translation Details

1 table per process
Part of process's state

Contents:
- Flags — dirty bit, resident bit, clock/reference bit
- Frame number

CPU

Virtual Addresses

Page Table

Physical Addresses

PTBR: Page Table Base Register
Segmentation VS Paging

### Hybrid solution:
1. **Paged segments**, segment is an integral number of pages
2. **Multiple page sizes**, with larger sizes being powers of 2 times

<table>
<thead>
<tr>
<th>Page</th>
<th>Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Words per address</td>
<td>One</td>
</tr>
<tr>
<td>Programmer visible?</td>
<td>Invisible to application programmer</td>
</tr>
<tr>
<td>Replacing a block</td>
<td>Trivial (all blocks are the same size)</td>
</tr>
<tr>
<td>Memory use inefficiency</td>
<td>Internal fragmentation (unused portion of page)</td>
</tr>
<tr>
<td>Efficient disk traffic</td>
<td>Yes (adjust page size to balance access time and transfer time)</td>
</tr>
</tbody>
</table>
## Cache VS VM

<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>12-128 bytes</td>
<td>4096-65,536 bytes</td>
</tr>
<tr>
<td>Hit time</td>
<td>1-2 clock cycles</td>
<td>40-100 clock cycles</td>
</tr>
<tr>
<td>Miss penalty (Access time)</td>
<td>8-100 clock cycles</td>
<td>700,000 – 6,000,000 clock cycles</td>
</tr>
<tr>
<td></td>
<td>(6-60 clock cycles)</td>
<td>(500,000 – 4,000,000 clock cycles)</td>
</tr>
<tr>
<td></td>
<td>(2-40 clock cycles)</td>
<td>(200,000 – 2,000,000 clock cycles)</td>
</tr>
<tr>
<td>Miss rate</td>
<td>0.5 – 10%</td>
<td>0.00001 – 0.001%</td>
</tr>
<tr>
<td>Data memory size</td>
<td>0.016 – 1 MB</td>
<td>4MB – 4GB</td>
</tr>
</tbody>
</table>

- Replacement on cache misses is primarily controlled by hardware
- The size of the processor address determines the size of virtual memory
- Secondary storage is also used for the file system
Page Table Organization

- Flat page table has size proportional to size of virtual address space
  - can be very large for a machine with 64-bit addresses and several processes
- Three solutions
  - page the page table (fixed mapping)
    - what really needs to be locked down?
  - multi-level page table (lower levels paged - Tree)
  - inverted page table (hash table)
Multi-Level Page Table

- Dir1
- Dir2
- Page
- offset

PTBR

Directory Directory

Page Directory

Page Table

Each of these is actually one page (4K) in size!

Only this table needs to stay in memory at all times.

e.g., 42-bit VA with 12-bit offset
10-bits for each of three fields
1024 4-byte entries in each table (one page)
Translation Table Base Address/Register

ARM A9 Page Table (1MB section)
L1+L2 Translation in A9
Linear Inverted Page Tables

- Store only PTEs for pages in physical memory
- Miss in page table implies page is on disk
- Need KP entries for P page frames (usually $K > 2$)

Requires a large CAM

http://www.cs.berkeley.edu/~kamil/teaching/sp04/040104.pdf
Hashed Inverted Page Tables

- Chaining in order to solve collisions
- Chain is exhausted by hitting an invalid next pointer => page fault
Virtual Address Translation - TLB

- What happens during a memory access?
  - map virtual address into physical address using page table
  - If the page is in memory: access physical memory
  - If the page is on disk: **page fault**
    » Suspend program
    » Get operating system to load the page from disk

- Page table is in memory - this slows down access!
- **Translation lookaside buffer** (TLB) special cache of translated addresses (speeds access back up)
Translation Look-Aside Buffers

- Translation Look-Aside Buffers (TLB)
  - Cache on translations
  - Fully Associative, Set Associative, or Direct Mapped

- TLBs are:
  - Small – typically not more than 128 – 256 entries
  - Fully Associative or 2-way set Associative
    - For example: A9 has 2 (instruction & data) 32-entry fully assoc. Micro TLBs and one 128-entry 2-way associative Main TLB
TLB Structure

Virtual page number

Valid | Tag | Physical page address
--- | --- | ---
1   |     |             
1   |     |             
1   |     |             
1   |     |             
0   |     |             
1   |     |             

Physical memory

Page table

Physical page

Valid or disk address

Disk storage
What Actually Happens on a TLB Miss?

- **Hardware-traversed page tables:**
  - On TLB miss, hardware in MMU looks at current page table to fill TLB (may walk multiple levels)
    » If PTE valid, hardware fills TLB and processor never knows
    » If PTE marked as invalid, causes Page Fault, after which kernel decides what to do afterwards

- **Software-traversed Page tables (like MIPS):**
  - On TLB miss, processor receives TLB fault
  - Kernel traverses page table to find PTE
    » If PTE valid, fills TLB and returns from fault
    » If PTE marked as invalid, internally calls Page Fault handler

- Most chip sets provide hardware traversal
  - Modern operating systems tend to have more TLB faults since they use translation for many things
  - **Examples:**
    » shared segments
    » user-level portions of an operating system
TLB – Cache Interaction

Virtual address

31 30 29  ...............  15 14 13 12 11 10 9 & ....  3 2 1 0

Virtual page number  Page offset

Valid Dirty  Tag  Physical page number

TLB

TLB hit

Physical page number  Page offset

Physical address tag  Cache index

Byte offset

16  14  2

Valid  Tag  Data

Cache

Cache hit

32  Data
TLB and Cache

- Is the cache indexed with virtual or physical address?
  - To index with a physical address, we will have to first look up the TLB, then the cache \(\rightarrow\) longer access time
  - Multiple virtual addresses can map to the same physical address – can we ensure that these different virtual addresses will map to the same location in cache? Else, there will be two different copies of the same physical memory word

- Does the tag array store virtual or physical addresses?
  - Since multiple virtual addresses can map to the same physical address, a virtual tag comparison can flag a miss even if the correct physical memory word is present

C-36 in textbook
Virtual Indexed, Virtually Tagged Cache

- **Protection bits** in cache
- **Cache flushing** on process switch or use **Process-identifier tag** (PID) (or Address Space Identifier=ASID)
- **Aliasing problem**: Two different virtual addresses sharing same physical
  - Page coloring: Forces aliases to share same cache block (i.e. alias addresses should have same cache index), thus aliases cannot co-exist in the cache
Better Alternative: Virtually Indexed, Physically Tagged Cache

What motivation?
- Fast cache hit by parallel TLB access
- No virtual cache shortcomings

How could it be correct?
- Require \#cache set * block size <= page size ⇒ physical index is from page offset
- Then virtual and physical indices are identical ⇒ works like a physically indexed cache!
Virtually Indexed, Physically Tagged Cache
Superpages

• If a program’s working set size is 16 MB and page size is 8KB, there are 2K frequently accessed pages – a 128-entry TLB will not suffice.

• By increasing page size to 128KB, TLB misses will be eliminated – disadvantage: memory wastage, increase in page fault penalty.

• Can we change page size at run-time?

• Note that a single large page has to be contiguous and aligned in physical memory: 128KB (17 bits) page includes 16 8KB (13 bits) pages.
Superpages Implementation

- At run-time, build superpages if you find that contiguous virtual pages are being accessed at the same time.

- For example, virtual pages 64-79 may be frequently accessed – coalesce these pages into a single superpage of size 128KB that has a single entry in the TLB.

- The physical superpage has to be in contiguous physical memory – the 16 physical pages have to be moved so they are contiguous.