Memory Management:
From Absolute Addresses to Demand Paging

Daniel Sanchez
Computer Science and Artificial Intelligence Laboratory
M.I.T.
Recap: Cache Organization

• Caches are small and fast memories that transparently retain recently accessed data.

• Cache organizations
  – Direct-mapped
  – Set-associative
  – Fully associative

• Cache performance
  – $AMAT = HitLatency + MissRate \times MissLatency$
  – Minimizing AMAT requires balancing competing tradeoffs.
Multilevel Caches

- A memory cannot be large and fast
- Add level of cache to reduce miss penalty
  - Each level can have longer latency than level above
  - So, increase sizes of cache at each level

**Metrics:**

Local miss rate = misses in cache / accesses to cache
Global miss rate = misses in cache / CPU memory accesses
Misses per instruction = misses in cache / number of instructions
Inclusion Policy

• Inclusive multilevel cache:
  – Inner cache holds copies of data in outer cache
  – External access need only check outer cache
  – Most common case

• Exclusive multilevel caches:
  – Inner cache may hold data not in outer cache
  – Swap lines between inner/outer caches on miss
  – Used in AMD Athlon with 64KB primary and 256KB secondary cache

*Why choose one type or the other?*

  Exclusive: Outer cache retains more data
  Inclusive: Less traffic, easier coherence
Victim cache is a small associative back up cache, added to a direct mapped cache, which holds recently evicted lines

- First look up in direct-mapped cache
- If miss, look in victim cache
- If hit in victim cache, swap hit line with line now evicted from L1
- If miss in victim cache, L1 victim -> VC, VC victim-?>?

Fast hit time of direct-mapped but with reduced conflict misses
Typical memory hierarchies

(a) Memory hierarchy for a server

- CPU
  - Registers
  - Register reference
  - Size: 1000 bytes
  - Speed: 300 ps

- L1 Cache
  - Level 1 Cache reference
  - Size: 64 KB
  - Speed: 1 ns

- L2 Cache
  - Level 2 Cache reference
  - Size: 256 KB
  - Speed: 3–10 ns

- L3 Cache
  - Level 3 Cache reference
  - Size: 2–4 MB
  - Speed: 10–20 ns

- Memory
  - Memory reference
  - Size: 4–16 GB
  - Speed: 50–100 ns

- I/O bus
- Disk storage
  - Disk memory reference
  - Size: 4–16 TB
  - Speed: 5–10 ms

(b) Memory hierarchy for a personal mobile device

- CPU
  - Registers
  - Register reference
  - Size: 500 bytes
  - Speed: 500 ps

- L1 Cache
  - Level 1 Cache reference
  - Size: 64 KB
  - Speed: 2 ns

- L2 Cache
  - Level 2 Cache reference
  - Size: 256 KB
  - Speed: 10–20 ns

- Memory
  - Memory reference
  - Size: 256–512 MB
  - Speed: 50–100 ns

- Storage
  - FLASH memory reference
  - Size: 4–8 GB
  - Speed: 25–50 us
HBM DRAM or MCDRAM

Source: AMD
Memory Management

• The Fifties
  - Absolute Addresses
  - Dynamic address translation

• The Sixties
  - Atlas and Demand Paging
  - Paged memory systems and TLBs

• Modern Virtual Memory Systems
Names for Memory Locations

- **Machine language address**
  - as specified in machine code

- **Virtual address**
  - ISA specifies translation of machine code address into virtual address of program variable (sometimes called *effective* address)

- **Physical address**
  - operating system specifies mapping of virtual address into name for a physical memory location
Absolute Addresses

*EDSAC, early 50’s*

\[
\text{virtual address} = \text{physical memory address}
\]
Absolute Addresses

EDSAC, early 50’s

virtual address = physical memory address

• Only one program ran at a time, with unrestricted access to entire machine (RAM + I/O devices)
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

virtual address = physical memory address
Absolute Addresses

**EDSAC, early 50’s**

virtual address = physical memory address

- 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
Absolute Addresses

EDSAC, early 50’s

```
virtual address = physical memory address
```

- 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?*
Absolute Addresses

EDSAC, early 50’s

virtual address = physical memory address

• 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
Multiprogramming

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

Base and bounds registers are visible/accessible only when processor is running in supervisor mode.
What is an advantage of this separation? (Scheme used on all Cray vector supercomputers prior to X1, 2002)
Memory Fragmentation

OS Space

user 1
16K

user 2
24K
24K
24K

user 3
32K
24K

free
Memory Fragmentation

Users 4 & 5 arrive
Memory Fragmentation

Users 4 & 5 arrive

<table>
<thead>
<tr>
<th>OS Space</th>
<th>user 1</th>
<th>user 2</th>
<th>user 3</th>
<th>user 5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16K</td>
<td>24K</td>
<td>32K</td>
<td>24K</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OS Space</th>
<th>user 1</th>
<th>user 2</th>
<th>user 4</th>
<th>user 3</th>
<th>user 5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16K</td>
<td>24K</td>
<td>16K</td>
<td>32K</td>
<td>24K</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OS Space</th>
<th>user 1</th>
<th>user 2</th>
<th>user 4</th>
<th>user 3</th>
<th>user 5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16K</td>
<td>24K</td>
<td>8K</td>
<td>32K</td>
<td>24K</td>
</tr>
</tbody>
</table>

Free space
Memory Fragmentation

OS Space

user 1
16K
user 2
24K
user 3
32K
user 4
24K
user 5
24K

Users 4 & 5 arrive

OS Space

user 1
16K
user 2
24K
user 4
16K
user 3
32K
user 5
24K

Users 2 & 5 leave

free
Memory Fragmentation

OS Space

- Users 4 & 5 arrive
  - user 1: 16K
  - user 2: 24K
  - user 3: 24K
  - user 4: 16K
  - user 5: 32K

- Users 2 & 5 leave
  - user 1: 16K
  - user 2: 24K
  - user 4: 16K
  - user 3: 32K
  - user 5: 24K
  - free space

February 19, 2019

MIT 6.823 Spring 2019
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 interpreted as a pair `<page number, offset>`

<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>

- 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.
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
Where Should Page Tables Reside?

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

- Idea: Keep PT of the current user in special registers
  - may not be feasible for large page tables
  - Increases the cost of context swap
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 PT of the current user in special registers
  – may not be feasible for large page tables
  – Increases the cost of context swap

• Idea: Keep PTs in the main memory
  – needs one reference to retrieve the page base address and another to access the data word
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 PT of the current user in special registers
  - may not be feasible for large page tables
  - Increases the cost of context swap

- 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

User 2

PT User 1

PT User 2
Page Tables in Physical Memory

Idea: cache the address translation of frequently used pages -- TLBs
A Problem in 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

Ferranti Mercury 1956

Central Store

40k bits main

640k bits drum
Manual Overlays

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

Ferranti Mercury 1956

40k bits main

640k bits drum

Central Store
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
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

- *Method 2:* automatic initiation of I/O transfers by software address translation
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

• *Method 2:* automatic initiation of I/O transfers by software address translation

*Brooker’s interpretive coding, 1960*
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

- **Method 2**: automatic initiation of I/O transfers by software address translation

  *Brooker’s interpretive coding, 1960*

Problems?
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

- *Method 2*: automatic initiation of I/O transfers by software address translation

Brooker’s interpretive coding, 1960

Ferranti Mercury
1956

40k bits
main

640k bits
drum

Central Store

Problems?
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

- **Method 2**: automatic initiation of I/O transfers by software address translation

  *Brooker’s interpretive coding, 1960*

Problems?  Method 1: Difficult, error prone

Ferranti Mercury 1956

40k bits main

640k bits drum

Central Store
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

• *Method 2*: automatic initiation of I/O transfers by software address translation

*Brooker’s interpretive coding, 1960*

Problems?

Method 1: Difficult, error prone
Method 2: Inefficient
“A page from secondary storage is brought into the primary storage whenever it is (implicitly) demanded by the processor.”

*Tom Kilburn*
“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
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

48-bit words
512-word pages

1 Page Address Register (PAR) per page frame

<effective PN, status>

8 Tape decks
88 sec/word

system code (not swapped)

system data (not swapped)
Hardware Organization of Atlas

Effective Address → Initial Address Decode → PARs

- 16 ROM pages: 0.4 ~ 1 μsec
- 2 subsidiary pages: 1.4 μsec
- Main: 32 pages: 1.4 μsec
- Drum (4): 192 pages

48-bit words
512-word pages

1 Page Address Register (PAR) per page frame

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

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 the 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 $\equiv$ 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*
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 space needed to back up the full virtual address space
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 space needed to back up the full virtual address space

Larger pages?

• Internal fragmentation (Not all memory in a page is used)
• Larger page fault penalty (more time to read from disk)
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 space needed to back up the full virtual address space

Larger pages?

- Internal fragmentation (Not all memory in a 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!)
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 space needed to back up the full virtual address space

Larger pages?

- Internal fragmentation (Not all memory in a 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

p2

Level 2 Page Tables

Data Pages

page in primary memory
page in secondary memory
PTE of a nonexistent page

offset

offset

offset
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** ⇒ *Single-cycle Translation*
- **TLB miss** ⇒ *Page Table Walk to refill*

---

**Virtual Address**

- **VPN** (virtual page number)
- **PPN** (physical page number)

**Physical Address**

- **VPN** and **offset**
- **PPN** and **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
• 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 = ____________________________?
TLB Designs

- **Typically 32-128 entries, usually fully associative**
  - Each entry maps a large page, hence less spatial locality across pages \( \Rightarrow \) more likely that two entries conflict
  - Sometimes larger TLBs (256-512 entries) are 4-8 way set-associative

- **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

\[ \text{TLB Reach} = 64 \text{ entries} \times 4 \text{ KB} = 256 \text{ KB (if contiguous)} \]
Variable-Sized Page Support

Virtual Address

31 22 21 12 11 0

\[ \begin{array}{ccc}
\text{p1} & \text{p2} & \text{offset} \\
\end{array} \]

10-bit 10-bit
L1 index L2 index

Root of the Current Page Table

(Processor Register)

Level 1 Page Table

Level 2 Page Tables

Data Pages

page in primary memory
large page in primary memory
page in secondary memory
PTE of a nonexistent page

offset
Variable-Size Page TLB

Some systems support multiple page sizes.

- virtual address
- VPN
- offset
- hit?
- physical address
- PPN
- offset
- Tag
- PPN
- L
- V R W D

February 19, 2019
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

<table>
<thead>
<tr>
<th>Context Table Register</th>
<th>Context Table</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>root ptr</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L1 Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L2 Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L3 Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTE</td>
</tr>
</tbody>
</table>

Physical Address

<table>
<thead>
<tr>
<th>PPN</th>
<th>Offset</th>
</tr>
</thead>
</table>

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

Diagram:
- Virtual Address
  - TLB Lookup
    - hit
      - Protection Check: permitted
        - Physical Address *(to cache)*
    - miss
      - Page Table Walk
        - the page is $\not\in$ memory
          - Page Fault *(OS loads page)*
        - the page is $\in$ memory
          - Update TLB
          - Protection Fault
            - SEGFAULT
    - Where?
Next lecture:
Modern Virtual Memory Systems