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
  - \( \text{AMAT} = \text{HitLatency} + \text{MissRate} \times \text{MissLatency} \)
  - Minimizing AMAT requires balancing competing tradeoffs
Replacement Policy

Which block from a set should be evicted?

- Random

- Least Recently Used (LRU)
  - LRU cache state must be updated on every access
  - true implementation only feasible for small sets (2-way)
  - pseudo-LRU binary tree was often used for 4-8 way

- First In, First Out (FIFO) a.k.a. Round-Robin
  - used in highly associative caches

- Not Least Recently Used (NLRU)
  - FIFO with exception for most recently used block or blocks

- One-bit LRU
  - Each way represented by a bit. Set on use, replace first unused.
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

![Multilevel Caches Diagram]

CPU  \(\rightarrow\) L1  \(\rightarrow\) L2  \(\rightarrow\) DRAM
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

![Diagram of CPU, L1, L2, and DRAM]

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 holds data not in outer cache
  - Swap lines between inner/outer caches on miss
  - Used in AMD Athlon with 64KB primary and 256KB secondary cache

- **Non-inclusive multilevel caches:**
  - Some cache lines duplicate in outer cache, and some do not
  - Intel Skylake L3

_Why choose one type or the other?_
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 holds data not in outer cache
  – Swap lines between inner/outer caches on miss
  – Used in AMD Athlon with 64KB primary and 256KB secondary cache

• Non-inclusive multilevel caches:
  – Some cache lines duplicate in outer cache, and some do not
  – Intel Skylake L3

Why choose one type or the other?

Exclusive: Outer cache retains more data
Inclusive: Less traffic, easier coherence
Victim Caches (HP 7200)
Victim Cache is a small associative back up cache, added to a direct mapped cache, which holds recently evicted lines.

Victim Cache

FA, 4 blocks

Evicted data from L1

Hit data (miss in L1)
Victim Caches (HP 7200)

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
Victim Caches (HP 7200)

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->?
Victim Caches (HP 7200)

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

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

*EDSAC, early 50’s*

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)
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
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
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?*
  \[ \Rightarrow \text{multiprogramming} \]
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
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
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?*
  \[\Rightarrow \text{multiprogramming}\]

Location-independent programs
• Programming and storage management ease
  \[\Rightarrow \text{need for a base register}\]

Protection
• Independent programs should not affect each other inadvertently
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*.
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
- user 3: 32K, 24K

free
Memory Fragmentation

Users 4 & 5 arrive

OS Space

user 1
16K

user 2
24K
24K

user 3
32K
24K

free
Memory Fragmentation

Initially:

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

After Users 4 & 5 arrive:

<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>
<tr>
<td></td>
<td></td>
<td></td>
<td>16K</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>8K</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>32K</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Free space: 8K
Memory Fragmentation

OS Space

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

Users 4 & 5 arrive

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

Users 2 & 5 leave

free

OS Space

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

8K
Memory Fragmentation

Users 4 & 5 arrive

Users 2 & 5 leave

free
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 frame number (PN)</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Address Space of User-1
Paged Memory Systems

- Processor-generated address can be interpreted as a pair <page number, offset>
  - page frame number (PN)  offset
- A page table contains the physical address of the base of each page
Paged Memory Systems

• Processor-generated address can be interpreted as a pair \(<\text{page number, offset}>\)
  
  \[
  \begin{array}{|c|c|}
  \hline
  \text{page frame number (PN)} & \text{offset} \\
  \hline
  \text{0} & \text{0x6} \\
  \text{1} & \text{0x7} \\
  \text{2} & \text{0x1} \\
  \text{3} & \text{0x3} \\
  \hline
  \end{array}
  \]

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

\[ \langle \text{virtual PN, physical PN} \rangle \]

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
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 virtual 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 virtual 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
Where Should Page Tables Reside?

- Space required by the page tables (PT) is proportional to the virtual 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
Where Should Page Tables Reside?

• Space required by the page tables (PT) is proportional to the virtual 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
Where Should Page Tables Reside?

• Space required by the page tables (PT) is proportional to the virtual 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

![Diagram of page tables in physical memory]

- User 1
  - VA1

- User 2
  - VA1

PT User 1
- PFN for VA1

PT User 2
Page Tables in Physical Memory

User 1

VA1

User 2

VA1

PT User 1

PFN for VA1

PT User 2
Page Tables in Physical Memory

Idea: cache the address translation of frequently used pages – TLBs (translation lookaside buffer)
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*
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

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

Ferranti Mercury
1956
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

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
  - Difficult, error prone
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*
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*

---

**Ferranti Mercury**

- 1956
- 40k bits main
- 640k bits drum
- Central Store
“A page from secondary storage is brought into the primary storage whenever it is (implicitly) demanded by the processor.”

*Tom Kilburn*
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

Primary

32 Pages

512 words/page

Secondary

(Drum)

32x6=192 pages

Central

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
- PARs: 32

**1 Page Address Register (PAR) per physical page frame**

- System code (not swapped)
- System data (not swapped)

**Effective Address**

- <effective PN, status>

- 8 Tape decks: 88 sec/word
Hardware Organization of Atlas

Compare the effective page address against all 32 PARs
match \( \equiv \) normal access
no match \( \Rightarrow \) 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
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)

Demand paging
- page frame
- page (~4K bytes)
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)

**Demand paging**
- page frame
- page (~4K bytes)
- page miss rate (<0.001%)
- page hit (~100 cycles)
- page miss (~5M cycles)
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
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
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
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
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
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
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
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

Page Table Entry (PTE) contains:

<table>
<thead>
<tr>
<th>Offset</th>
<th>VPN</th>
<th>Data Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data word</td>
<td>Data PPN</td>
<td></td>
</tr>
<tr>
<td>PPN</td>
<td>PPN</td>
<td></td>
</tr>
<tr>
<td>DPN</td>
<td>DPN</td>
<td></td>
</tr>
</tbody>
</table>

PT Base Register

Virtual address

February 13, 2020
MIT 6.823 Spring 2020
L04-75
Size of Linear Page Table

With 32-bit addresses, 4 KB pages & 4-byte PTEs:
Size of Linear Page Table

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

\[ 2^{20} \text{ PTEs}, \text{i.e., 4 MB page table per user} \]
\[ 4 \text{ GB of swap space needed to back up the full virtual address space} \]
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?
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:

\[ 2^{20} \text{ PTEs, i.e, 4 MB page table per user} \]
\[ 4 \text{ GB of swap space needed to back up the full virtual address space} \]

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

Level 2 Page Tables

offset

Data Pages

page in primary memory
page in secondary memory

PTE of a nonexistent page
Address Translation & Protection

Virtual Address

Virtual Page No. (VPN) offset

Exception?

Physical Address

Physical Page No. (PPN) offset

• Every instruction and data access needs address translation and protection checks
Every instruction and data access needs address translation and protection checks.
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*

![Diagram of virtual and physical address translation]
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*

```
VRWD tag PPN
hit? physical address

virtual address

VPN offset

(VPN = virtual page number)

(PPN = physical page number)
```
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
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 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 ➔ 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 = 64 entries * 4 KB = 256 KB (if contiguous)?
Variable-Sized Page Support

Virtual Address

```
<table>
<thead>
<tr>
<th></th>
<th>31</th>
<th>22 21</th>
<th>12 11</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>p2</td>
<td>offset</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

10-bit 10-bit
L1 index L2 index

Root of the Current Page Table

(p1)

Level 1 Page Table

(Processor Register)

Level 2 Page Tables

offset

Data Pages

- Page in primary memory
- Large page in primary memory
- Page in secondary memory
- PTE of a nonexistent page
Variable-Sized Page Support

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

Level 2 Page Tables

Data Pages

page in primary memory
large page in primary memory
page in secondary memory
PTE of a nonexistent page
Variable-Size Page TLB

Some systems support multiple page sizes.
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

Index 2

Index 3

Offset

31

23

17

11

0

L1 Table

L2 Table

L3 Table

PTP

PTP

PTP

PTE

PPN

Offset

Physical Address

MMU does this table walk in hardware on a TLB miss

Context Register

root ptr
Address Translation: putting it all together

Virtual Address

TLB Lookup

Page Table Walk

Protection Check

Hit

The page is permitted

Where?

Page Fault (OS loads page)

Update TLB

Protection Fault

Miss

The page is denied

SEGFAULT

Physical Address (to cache)
Next lecture:
Modern Virtual Memory Systems