Modern Virtual Memory Systems

Mengjia Yan
Computer Science and Artificial Intelligence Laboratory
M.I.T.
Reminder: How Virtual Memory Systems Evolved in the Past

- Want to write position-independent code
Reminder: How Virtual Memory Systems Evolved in the Past

- Want to write position-independent code
  - Base and Bound Translation
Reminder: How Virtual Memory Systems Evolved in the Past

- Want to write position-independent code
  - Base and Bound Translation
- The fragmentation issue
Reminder: How Virtual Memory Systems Evolved in the Past

- Want to write position-independent code
  - Base and Bound Translation
- The fragmentation issue
  - Paged memory system
Reminder: How Virtual Memory Systems Evolved in the Past

- Want to write position-independent code
  - Base and Bound Translation
- The fragmentation issue
  - Paged memory system
- Program data cannot fit in the primary memory
Reminder: How Virtual Memory Systems Evolved in the Past

- Want to write position-independent code
  - Base and Bound Translation
- The fragmentation issue
  - Paged memory system
- Program data cannot fit in the primary memory
  - Manual overlay => demand paging
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 \(=\) memory view \(=\) 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**
Private Address Space per User

- Each user has a page table
- *page table* ⇔ *memory view* ⇔ *name space*
Private Address Space per User

- Each user has a page table
- *page table* $\equiv$ *memory view* $\equiv$ *name space*
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

- User 1
  - VA1

- User 2
  - VA1

- PFN for VA1

- PT User 1
- PT User 2
Page Tables in Physical Memory

User 1

User 2

VA1

VA1

PFN for VA1

PT User 1

PT User 2
Page Tables in Physical Memory

User 1

User 2

PT User 1

PT User 2

PFN for VA1
Idea: cache the address translation of frequently used pages – TLBs (translation lookaside buffer)
Linear Page Table

- Page Table Entry (PTE) contains:
  - A bit to indicate if a page exists
  - \textbf{PPN} (physical page number) for a memory-resident page
  - \textbf{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:
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?
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???
Size of Linear Page Table

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

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

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!)
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

<table>
<thead>
<tr>
<th>0</th>
<th>11</th>
<th>12</th>
<th>21</th>
<th>22</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>offset</td>
<td>p2</td>
<td>p1</td>
<td>10-bit L1 index</td>
<td>10-bit L2 index</td>
<td></td>
</tr>
</tbody>
</table>

Root of the Current Page Table

(Processor Register)

Level 1 Page Table

Level 2 Page Tables

Data Pages

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
Address Translation & Protection

- Every instruction and data access needs address translation and protection checks

A good Virtual Memory design needs to be fast (~ one cycle) and space-efficient
Translation Lookaside Buffers

Address translation is very expensive!
In a hierarchical page table, each reference becomes several memory accesses
Translation Lookaside Buffers

Address translation is very expensive!
In a hierarchical 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*
Translation Lookaside Buffers

Address translation is very expensive!
In a hierarchical 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*

\[
\begin{array}{c|c|c|c}
V & R & W & D \\
\hline
\text{tag} & \text{PPN} & \\
\end{array}
\]

(VPN = virtual page number)

\[
\begin{array}{c|c}
\text{VPN} & \text{offset} \\
\end{array}
\]

(PPN = physical page number)

\[
\begin{array}{c|c}
\text{VPN} & \text{offset} \\
\end{array}
\]

hit?  physical address
TLB Designs

• Keep process information in TLB?
TLB Designs

• Keep process information in TLB?
  – No process id → Must flush on context switch
  – Tag each entry with process id → No flush, but costlier
TLB Designs

• Keep process information in TLB?
  – No process id → Must flush on context switch
  – Tag each entry with process id → No flush, but costlier

• Size and Associativity
TLB Designs

• Keep process information in TLB?
  – No process id → Must flush on context switch
  – Tag each entry with process id → No flush, but costlier

• Size and Associativity
  – Typically 32-128 entries, usually highly associative
TLB Designs

• Keep process information in TLB?
  – No process id → Must flush on context switch
  – Tag each entry with process id → No flush, but costlier

• Size and Associativity
  – Typically 32-128 entries, usually highly associative

• 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

• Keep process information in TLB?
  – No process id → Must flush on context switch
  – Tag each entry with process id → No flush, but costlier

• Size and Associativity
  – Typically 32-128 entries, usually highly associative

• 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 = \(64 \text{ entries} \times 4 \text{ KB} = 256 \text{ KB (if contiguous)}\)
TLB Designs

• Keep process information in TLB?
  – No process id \(\rightarrow\) Must flush on context switch
  – Tag each entry with process id \(\rightarrow\) No flush, but costlier

• Size and Associativity
  – Typically 32-128 entries, usually highly associative

• 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 = \(64\ \text{entries} * 4\ \text{KB} = 256\ \text{KB} (\text{if contiguous})\) ?

• Ways to increase TLB reach
TLB Designs

• Keep process information in TLB?
  – No process id → Must flush on context switch
  – Tag each entry with process id → No flush, but costlier

• Size and Associativity
  – Typically 32-128 entries, usually highly associative

• 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)

• Ways to increase TLB reach
  – Multi-level TLBs (e.g., Intel Skylake: 64-entry L1 data TLB, 128-entry L1 instruction TLB, 1.5K-entry L2 TLB)
  – Multiple page sizes, e.g., x86-64: 4KB, 2MB, 1GB
Variable-Size Page TLB

virtual address – small page

large page

How to organize TLBs? Which bits to index TLB?
Variable-Size Page TLB

virtual address – small page
large page

hit?

physical address

Large page?
### Variable-Size Page TLB

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

**Step 1:** Assume 4KB page size, calculate index and probe

**Step 2:** If miss, assume 2MB page, re-calculate index and probe

---

**virtual address** – small page

**large page**

- VPN
- Offset

Large page?
Variable-Size Page TLB

Alternatively, have a separate TLB for each page size
Variable-Size Page TLB

virtual address – small page
large page

Alternatively, have a separate TLB for each page size (pros/cons compared to unified TLB?)

September 18, 2023
Variable-Size Page TLB

Alternatively, have a separate TLB for each page size (pros/cons compared to unified TLB?)

Example: Intel Skylake

<table>
<thead>
<tr>
<th></th>
<th>4KB</th>
<th>2MB</th>
<th>1GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1-D TLB</td>
<td>64</td>
<td>32</td>
<td>4</td>
</tr>
<tr>
<td>L1-I TLB</td>
<td>128</td>
<td>8</td>
<td>/</td>
</tr>
<tr>
<td>L2 STLB</td>
<td>1536</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>
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
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.

What is the trade-off?
Hierarchical Page Table Walk: SPARC v8

Virtual Address

<table>
<thead>
<tr>
<th>Context Table Register</th>
<th>Context Table</th>
<th>Index 1</th>
<th>Index 2</th>
<th>Index 3</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>31</td>
<td>23</td>
<td>17</td>
<td>11</td>
</tr>
</tbody>
</table>

Context Register

root ptr

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

September 18, 2023
Address Translation: *putting it all together*

**Virtual Address**

- **TLB Lookup**
  - hit
  - **Protection Check**
    - denied
    - permitted
    - Physical Address (to cache)
  - miss
    - **Page Table Walk**
      - the page is
        - $\notin$ memory
        - $\in$ memory
      - **Page Fault** (OS loads page)
    - **Update TLB**

Where?

SEGFAULT
Address Translation: *putting it all together*

Virtual Address

- TLB Lookup
  - hit
  - miss

  - Page Table Walk
    - the page is not in memory
      - Page Fault (OS loads page)
    - the page is in memory
      - Update TLB

  - Protection Check
    - denied
      - Protection Fault
    - permitted
      - Physical Address (to cache)

Where?

- Re-execute the instruction that causes the page fault/TLB miss

SEGFAULT
Topics

• Speeding up the common case:
  – TLB & Cache organization

• Interrupts

• Modern Usage
Address Translation in CPU
Address Translation in CPU
Address Translation in CPU

- Need mechanisms to cope with the additional latency of TLB:
Address Translation in CPU

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

CPU → TLB → Physical Cache → Primary Memory

VA → PA
Virtual-Address Caches

Alternative: place the cache before the TLB
Virtual-Address Caches

Alternative: place the cache before the TLB

Pros and cons?
Virtual-Address Caches

Alternative: place the cache before the TLB

Pros and cons?

- one-step process in case of a hit (+)
Virtual-Address Caches

Alternative: place the cache before the TLB

Pros and cons?

- 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 (-)
Virtual-Address Caches

Alternative: place the cache before the TLB

Pros and cons?

- 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 (-)
Aliasing in Virtual-Address Caches

Two virtual pages share one physical page
Aliasing in Virtual-Address Caches

Two virtual pages share one physical page

Virtual cache can have two copies of same physical data. Writes to one copy not visible to reads of other!
Aliasing in Virtual-Address Caches

Two virtual pages share one physical page

Virtual cache can have two copies of same physical data. Writes to one copy not visible to reads of other!

General Solution: *Disallow aliases to coexist in cache*
Aliasing in Virtual-Address Caches

Two virtual pages share one physical page

Virtual cache can have two copies of same physical data. Writes to one copy not visible to reads of other!

General Solution: *Disallow aliases to coexist in cache*

Software (i.e., OS) solution for direct-mapped cache

VAs of shared pages must agree in cache index bits; this ensures all VAs accessing same PA will conflict in direct-mapped cache (early SPARCs)
Index L is available without consulting the TLB
\( \Rightarrow \) cache and TLB accesses can begin simultaneously
Tag comparison is made after both accesses are completed
Index L is available without consulting the TLB

⇒ cache and TLB accesses can begin simultaneously

Tag comparison is made after both accesses are completed
Index L is available without consulting the TLB
→ cache and TLB accesses can begin simultaneously
Tag comparison is made after both accesses are completed

When does this work? $L + b < k \quad L + b = k \quad L + b > k$
Concurrent Access to TLB & Cache

Index $L$ is available without consulting the TLB

⇒ cache and TLB accesses can begin simultaneously

Tag comparison is made after both accesses are completed

When does this work? $L + b < k \checkmark \quad L + b = k \_ \quad L + b > k \_\_ \_ \_ \_ \_ \_ \_ \_ \_ \_
Concurrent Access to TLB & Cache

Index \( L \) is available without consulting the TLB
\( \Rightarrow \text{cache and TLB accesses can begin simultaneously} \)
Tag comparison is made after both accesses are completed

When does this work? \( L + b < k \) ✓  \( L + b = k \) ✓  \( L + b > k \)
Concurrent Access to TLB & Cache

Index L is available without consulting the TLB

⇒ *cache and TLB accesses can begin simultaneously*

Tag comparison is made after both accesses are completed

*When does this work?* $L + b < k \checkmark$  $L + b = k \checkmark$  $L + b > k \times$
Concurrent Access to TLB & Large L1

The problem with L1 > Page size

Diagram:
- VA (Virtual Address) → VPN → TLB → L1 PA cache
- PA (Physical Address) → PPN → TLB → L1 PA cache

Virtual Index

Direct-map

hit?
Concurrent Access to TLB & Large L1

The problem with L1 > Page size

Virtual Index

L1 PA cache
Direct-map

hit?

VA
VPN
Page Offset
TLB

Tag

PA
PPN
Page Offset

L

September 18, 2023

MIT 6.5900 Fall 2023

L04-26
Concurrent Access to TLB & Large L1

The problem with L1 > Page size

Can \( VA_1 \) and \( VA_2 \) both map to PA?
Concurrent Access to TLB & Large L1
The problem with \( L_1 > \text{Page size} \)

Can \( VA_1 \) and \( VA_2 \) both map to \( PA \)?

Yes
Virtual-Index Physical-Tag Caches: Associative Organization

After the PPN is known, $2^a$ physical tags are compared.
Virtual-Index Physical-Tag Caches: Associative Organization

After the PPN is known, $2^a$ physical tags are compared

*Is this scheme realistic for larger caches?*
Virtual-Index Physical-Tag Caches: Associative Organization

After the PPN is known, $2^a$ physical tags are compared.

*Is this scheme realistic for larger caches?*

No. Each cache way should not exceed page size, and we cannot scale associativity too much for performance reasons.
A solution via Second-Level Cache

Usually a common L2 cache backs up both Instruction and Data L1 caches

L2 is “inclusive” of both Instruction and Data caches
Anti-Aliasing Using L2: \textit{MIPS R10000}

- Suppose VA1 and VA2 both map to PA and VA1 is already in L1, L2 (VA1 ≠ VA2)
- After VA2 is resolved to PA, collision is detected in L2.
Anti-Aliasing Using L2: MIPS R10000

- Suppose VA1 and VA2 both map to PA and VA1 is already in L1, L2 (VA1 ≠ VA2)
- After VA2 is resolved to PA, collision is detected in L2. → Tag `a` in L2 is different
Anti-Aliasing Using L2: *MIPS R10000*

- Suppose VA1 and VA2 both map to PA and VA1 is already in L1, L2 (VA1 ≠ VA2)
- After VA2 is resolved to PA, collision is detected in L2. → Tag `a` in L2 is different
- VA1 will be purged from L1, and VA2 will be loaded ⇒ *no aliasing!*

September 18, 2023

MIT 6.5900 Fall 2023
Anti-Aliasing Using L2: *MIPS R10000*

- Suppose VA1 and VA2 both map to PA and VA1 is already in L1, L2 (VA1 ≠ VA2)
- After VA2 is resolved to PA, collision is detected in L2. → Tag `a` in L2 is different
- VA1 will be purged from L1, and VA2 will be loaded ⇒ *no aliasing!*
Anti-Aliasing Using L2: *MIPS R10000*

- Suppose VA1 and VA2 both map to PA and VA1 is already in L1, L2 (VA1 ≠ VA2)
- After VA2 is resolved to PA, collision is detected in L2. → Tag `a` in L2 is different
- VA1 will be purged from L1, and VA2 will be loaded ⇒ *no aliasing!*

![Diagram showing the process of anti-aliasing using L2 cache]
Anti-Aliasing Using L2: \textit{MIPS R10000}

- Suppose VA1 and VA2 both map to PA and VA1 is already in L1, L2 (VA1 \(\neq\) VA2)
- After VA2 is resolved to PA, collision is detected in L2. \(\Rightarrow\) Tag `a` in L2 is different
- VA1 will be purged from L1, and VA2 will be loaded \(\Rightarrow no\ aliasing!\)
Topics

• Speeding up the common case:
  – TLB & Cache organization

• Interrupts

• Modern Usage
Address Translation in CPU

- PC
- Inst TLB
- Inst. Cache
- RegFile
- +
- Data TLB
- Data Cache
Address Translation in CPU

TLB miss? Page Fault? Protection violation?

TLB miss? Page Fault? Protection violation?
Address Translation in CPU

- Handling a TLB miss needs a *hardware* or *software* mechanism to refill TLB
Address Translation in CPU

- Handling a TLB miss needs a *hardware* or *software* mechanism to refill TLB
- Software handlers need a *restartable exception* on page fault or protection violation
Interrupts: altering the normal flow of control

An external or internal event that needs to be processed by another (system) program. The event is usually unexpected or rare from program’s point of view.
Causes of Interrupts

Interrupt: an event that requests the attention of the processor

• Asynchronous: an external event
  – input/output device service-request
  – timer expiration
  – power disruptions, hardware failure

• Synchronous: an internal event (a.k.a. exception)
  – undefined opcode, privileged instruction
  – arithmetic overflow, FPU exception
  – misaligned memory access
  – virtual memory exceptions: page faults, TLB misses, protection violations
  – traps: system calls, e.g., jumps into kernel
Asynchronous Interrupts

**Invoking the interrupt handler**

- An I/O device requests attention by asserting one of the *prioritized interrupt request lines*
- Privilege control registers
  - status, epc, evec, cause, ...
- When the processor decides to process interrupt
  - It stops the current program at instruction $I_i$, completing all the instructions up to $I_{i-1}$ (*precise interrupt*)
  - It saves the PC of instruction $I_i$ in a special register (epc)
  - It saves the cause of interrupt to a special register (cause)
  - It disables interrupts and transfers control to a designated interrupt handler running in kernel mode (set pc to evec, set status to supervisor mode)
Synchronous Interrupts

- A synchronous interrupt (exception) is caused by a particular instruction

- In general, the instruction cannot be completed and needs to be restarted after the exception has been handled
  - With pipelining, requires undoing the effect of one or more partially executed instructions
Synchronous Interrupts

- A synchronous interrupt (exception) is caused by a *particular instruction*

- In general, the instruction cannot be completed and needs to be *restarted* after the exception has been handled
  - With pipelining, requires undoing the effect of one or more partially executed instructions

- In case of a trap (system call), the instruction is considered to have been completed
  - A special jump instruction involving a change to privileged kernel mode
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
Topics

• Speeding up the common case:
  – TLB & Cache organization

• Interrupts

• Modern Usage
Virtual Memory Use Today - 1

- Desktop/server/cellphone processors 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 and GPUs have translation and protection but not demand paging
  (Older Crays: base&bound, Japanese & Cray X1: pages)
  - Don’t waste expensive processor 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
Next lecture: Pipelining!
Interrupt Handler

• Saves EPC before enabling interrupts to allow nested interrupts ⇒
  – need an instruction to move EPC into GPRs
  – need a way to mask further interrupts at least until EPC can be saved

• Needs to read a *status register* that indicates the cause of the interrupt

• Uses a special indirect jump instruction *eret* (*exception-return*) that
  – enables interrupts
  – restores the processor to the user mode
  – restores hardware status and control state