Goals of Virtual Memory

• Allow physical memory to be smaller than virtual memory—applications receive illusion of huge address spaces!

• At any given time, a process’ virtual address space may be fully in RAM, partially in RAM, or not in RAM at all

• Automate the chore of moving pages between memory and disk

• Provide memory isolation between processes and the OS memory (but allow sharing when desired!)

• How do systems implement paging in real life?
Case study: x86 (Hardware-defined page tables)

<table>
<thead>
<tr>
<th>Directory number</th>
<th>Page table number</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>22 21</td>
<td>12 11 0</td>
</tr>
</tbody>
</table>

- **PDE (Page Directory Entry)**
  - 32 bits
  - Holds physical address of directory

- **PTE (Page Table Entry)**
  - 10 bits
  - User-mode accessible?

- **%cr3**
  - Bookkeeping stuff

- **4KB-aligned pgTable addr**
  - 31 12 11 0
Case study: x86 (Hardware-defined page tables)

- **Directory number**
- **Page table number**
- **Offset**

<table>
<thead>
<tr>
<th>31</th>
<th>22</th>
<th>21</th>
<th>12</th>
<th>11</th>
<th>0</th>
</tr>
</thead>
</table>

- **PDE**
- **Page table directory**
- **%cr3**

- **Page table**
- **PhysAddr**
- **4KB page**

- **Present?**
- **Writeable?**
- **Accessed recently?**
- **Dirty?**

- Automatically set by hardware
Case study: x86 (Hardware-defined page tables)

Q1: How many pages can a process contain?
Q2: How much memory does a single page table cover?
Q3: What is the minimum size of a machine’s physical memory?
Q4: What is the maximum size of a machine’s physical memory?
x86 Physical Address Extension (PAE)

<table>
<thead>
<tr>
<th>Ptr#</th>
<th>Directory number</th>
<th>Page table number</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>21</td>
</tr>
</tbody>
</table>

- **PDPTE (64 bits)**
- **PDE (64 bits)**
- **PTE (64 bits)**
- **4KB page**

- **Page directory**
- **Page table directory**
- **%cr3**

Note that:
- Virtual address space is still 32-bits wide
- Physical address space is now 36-bits wide (i.e., only 36 bits of the 64-bit paging entries are actually used for physical addressing)
x86: Segmentation plus Paging

- x86 (32-bits) and x64 (when running in 32-bit mode) support both segmentation and paging!
- Strictly speaking, the “linear” address space is what gets paged.
x86: Segmentation plus Paging

Segment selector → Virtual address → Local or global descriptor table → Segment descriptor → Segment base → Linear address space → Segment → Linear addr → Page directory → PDPTE → Page table → PTE → Offset → Linear address

%cr3 → Physical address space → Page → Physical addr

Segmentation + Paging
x86: Segmentation plus Paging

- Modern OSes like Windows and Linux configure `%cs`, `%ds`, and `%ss` to have a base of 0 and a bounds of $2^{32}$ bytes.
- So, segmentation is a no-op.
x86: Segmentation plus Paging

• On 32-bit x86, modern OSes like Windows and Linux configure `%cs`, `%ds`, and `%ss` to have a base of 0 and a bounds of $2^{32}$ bytes

• However, `%fs` and `%gs` used for systems chicanery
  • Ex: x86 Linux uses the `%fs` segment to store per-CPU information (remember that segment registers are per-core!); so, an instruction like `inc %gs: (%eax)` will increment a per-CPU memory location

• When x64 runs in 64-bit mode, the hardware forces `%cs`, `%ds`, and `%ss` to have a base of 0 and a bounds of $2^{64}$ bytes
  • `%fs` and `%gs` still available for systems chicanery
Q: What do page tables look like on MIPS R3000?

A: You get to decide!
Paging: The Good and the Bad

- Good: A virtual address space can be bigger than physical memory
- Bad: Each virtual memory access now requires at least two physical memory accesses

Physical memory accesses:
1. Load entry from page table directory
2. Load entry from page table
3. Generate the “real” memory access
Translation Lookaside Buffers (TLBs)

- Idea: Cache some PTEs in small hardware buffer
  - If virtual address has an entry in TLB, don’t need to go to physical memory to fetch PTEs!
  - If virtual address misses in TLB, we must pay at least one physical memory access to fetch PTE

Key/value store: key is high bits of virt addr, value is phys frame number
Translation Lookaside Buffers (TLBs)

- TLBs are effective because programs exhibit locality

  - Temporal locality: When a process accesses virtual address $x$, it will likely access $x$ again in the future (Ex: a function’s local variable that lives on the stack)

  - Spatial locality: When the process accesses something at memory location $x$, the process will likely access other memory locations close to $x$ (Ex: reading elements from an array on the heap)
The Lifecycle of a Memory Reference on x86

Virtual address

TLB lookup

TLB hit?

Yes

Check protection bits

No

Access ok?

Yes

Calculate phys addr, send to L1/L2/L3/RAM

No

HW raises a page fault

No

Hardware raises a page fault

Yes

HW updates TLB

No

HW walks the page table

Found PTE for virt frame?

Yes

HW updates TLB

No

HW raises a page fault
The Lifecycle of a Memory Reference on x86

Virtual address

TLB lookup

No

HW walks the page table

Found PTE for virt frame?

Yes

HW updates TLB

No

HW raises a page fault

Yes

Check protection bits

No

Access ok?

Yes

Calculate phys addr, send to L1/L2/L3/RAM

No

HW raises a page fault

Before raising page fault exception, HW sets %cr2 to faulting address, and pushes an error code onto stack.
- Ex: User process tried to read a non-present page
- Ex: User process tried to write a present but read-only page
MIPS R3000: Interacting with the TLB

A single TLB entry: a TLBHI structure + a TLBLO structure

<table>
<thead>
<tr>
<th>Virtual frame number</th>
<th>Address space ID</th>
<th>Unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>12 11</td>
<td>6 5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Physical frame number</th>
<th>Unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>12 11</td>
</tr>
</tbody>
</table>

- Valid?
- Writable?
- Global?
MIPS R3000: Interacting with the TLB

OS must set ASID during context switch!

Used to select tlb[] entry for:
- TLBR: Read TLB entry into %TLBHI and %TLBLO
- TLBWI: Write %TLBHI and %TLBLO to TLB entry
MIPS R3000: Interacting with the TLB

Set to tlb[] index by:

- TLBP: Search TLB for entry matching %TLBHI, set INDEX to matching TLB entry or -1 if no match found
MIPS R3000: Interacting with the TLB

<table>
<thead>
<tr>
<th>%TLBHI</th>
<th>%TLBLO</th>
<th>%INDEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual frame number</td>
<td>Address space ID</td>
<td>Unused</td>
</tr>
<tr>
<td>Physical frame number</td>
<td>Unused</td>
<td>Unused</td>
</tr>
</tbody>
</table>

Not used by:
- TLBWR: Write %TLBHI and %TLBLO to random TLB entry

```
<table>
<thead>
<tr>
<th>tlb[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual frame number</td>
</tr>
<tr>
<td>Physical frame number</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tlb[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual frame number</td>
</tr>
<tr>
<td>Physical frame number</td>
</tr>
</tbody>
</table>

| ... |
| ... |

<table>
<thead>
<tr>
<th>tlb[63]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual frame number</td>
</tr>
<tr>
<td>Physical frame number</td>
</tr>
</tbody>
</table>
```

TLB
The Lifecycle of a Memory Reference on MIPS

Virtual address and %TLBHI::ASID

TLB lookup

TLB hit?

Yes: Check "writeable" bit

No: Consider "G" bits!

TLB hit?

Yes: Access ok?

No: HW sets %BADVADDR, raises exception #2 ("TLB miss on load") or #3 ("TLB miss on store")

YOUR IMAGINATION CREATES A MIRACLE

No: HW sets %BADVADDR, raises exception #1 ("Read-only fault")

Calculate phys addr, send to L1/L2/L3/RAM
TLBs and Context Switches

• If TLB entries are tagged with ASIDs:
  • OS updates current ASID (e.g., by setting TLBHI::ASID on MIPS)
  • OS doesn’t need to flush TLBs
  • Even if OS occasionally has to evict entries, this is better than having to evict ALL entries during EVERY context switch (since this generally requires size(TLB) page table walks when a new task starts to warm TLB)
  • Scheduler can reduce invalidations with AS-to-core affinity

• If TLB entries are *not* tagged with ASIDs:
  • OS must invalidate all TLB entries during a context switch
  • x86: Writing to %cr3 on x86—this updates PDE pointer and invalidates all TLB entries
  • MIPS: OS can use constant value for all ASIDs, and manually invalidate all TLB entries during context switch
TLB Invalidations

- When OS changes a PTE, must also invalidate any matching TLB entry!
  - x86: “INVLPG virtAddr” invalidates individual TLB entry
  - MIPS: Use “TLBP” (the TLB probe instruction) to set %INDEX to that of the TLB entry to invalidate; then, use “TLBWI” to overwrite it

- On a multicore machine, PTEs from a single address space can be mapped into multiple per-core TLBs
  - If a core wants to modify a PTE entry, it must send cross-core interrupts to other cores
  - Once other cores are spin-waiting, first core modifies PTE then wakes up other cores
  - Other cores invalidate relevant TLB entries and resume execution
TLB Design Trade-offs

- Software-managed TLB
  - Good: OS has freedom to design page tables, page directories, and other arbitrarily interesting structures
  - Good: OS has freedom to design TLB eviction policy that might be too complex to implement in hardware
  - Bad: Performance overhead
    - Software is slower than hardware
    - OS lacks access to low-level hardware state, so handling TLB misses in software may require discarding work that’s already in the CPU pipeline
WHAT HAPPENS TO THIS STATE?

Fetch

Decode

Execute/Memory

Update registers

 mov %eax, [%esp]
 add %eax, 42
 sub %edi, %eax
...

 TLB MISS

 sub %edi, %eax

 add %eax, 42

 mov %eax, [%esp]
TLB Design Trade-offs

- Software-managed TLB
  - Good: OS has freedom to design page tables, page directories, and other arbitrarily interesting structures
  - Good: OS has freedom to design TLB eviction policy that might be too complex to implement in hardware
  - Bad: Performance overhead
    - Software is slower than hardware
    - OS lacks access to low-level hardware state, so handling TLB misses in software may require discarding work that’s already in the CPU pipeline
- Hardware-managed TLB
  - Good: TLB miss doesn’t cause exception that must be handled by OS
    - Hardware can just stall the current instruction . . .
    - . . . and let other instructions proceed!
  - Bad: Page table/page directory/etc format can’t be changed by OS