Announcements

• Next five weeks
  - Week 10 (Apr 9): Proj4 (Buffer Overflow/calling convention) release, due week later
  - Week 11 (Apr 16): Proj5 (caches) release, due week and half later on Fri
  - Week 12 (Apr 23): C assignment 2 due Mon
  - Week 13 (Apr 30): Proj6 (multi-core/parallelism) release Prelim2 on Thur, May 3
  - Week 14 (May 7): Proj5 Tournament, Monday, May 7 Proj6 design doc due

• Final Project for class
  - Week 15 (May 15): Proj6 due Tue May 15 at 4:30pm
Where are we now and where are we going?

• How many programs do you run at once?
  
  • a) 1
  • b) 2
  • c) 3-5
  • d) 6-10
  • e) 11+
Big Picture: Multiple Processes

• Can we execute *more than one* program at a time with our current MIPS processor?
Big Picture: Multiple Processes

• How to run multiple processes?
• *Time-multiplex* a single CPU core (multi-tasking)
  ▪ Web browser, skype, office, ... all must co-exist

• Many cores per processor (multi-core) or many processors (multi-processor)
  ▪ Multiple programs run *simultaneously*
Processor & Memory

- CPU address/data bus...
- ... routed through caches
- ... to main memory
  - Simple, fast, but...

![Diagram showing CPU and memory regions: Text (0x000...0), Data (0x7ff...f), Stack (0xfff...f), and Heap (0x7ff...f)]
Multiple Processes

- Q: What happens when another program is executed concurrently on another processor?
Multiple Processes

• Q: Can we relocate second program?
Takeaway

- All problems in computer science can be solved by another level of indirection.
  - – David Wheeler
  - – or, Butler Lampson
  - – or, Leslie Lamport
  - – or, Steve Bellovin

Solution: Need a **MAP**

To map a **Virtual Address (generated by CPU)** to a **Physical Address (in memory)**
Big Picture: (Virtual) Memory

• How do we execute more than one program at a time?

• A: Abstraction – Virtual Memory
  ▪ Memory that appears to exist as main memory (although most of it is supported by data held in secondary storage, transfer between the two being made automatically as required—i.e. ”paging”)
  ▪ Abstraction that supports multi-tasking---the ability to run more than one process at a time
Next Goal

• How does Virtual Memory work?

• i.e. How do we create the “map” that maps a virtual address generated by the CPU to a physical address used by main memory?
Picture Memory as...?

Byte Array:

<table>
<thead>
<tr>
<th>addr</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffffffff</td>
<td>xaa</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>x00</td>
<td></td>
</tr>
<tr>
<td>x00</td>
<td></td>
</tr>
<tr>
<td>xef</td>
<td></td>
</tr>
<tr>
<td>xcd</td>
<td></td>
</tr>
<tr>
<td>xab</td>
<td></td>
</tr>
<tr>
<td>xff</td>
<td></td>
</tr>
<tr>
<td>0x00000000</td>
<td>x00</td>
</tr>
</tbody>
</table>

Segments:

- System reserved
- Stack
- Heap
- Data
- Text
- System reserved

Page Array:

- Page 0
- Page 1
- Page 2
- Page n

Each segment uses some # of pages

New!
A Little More About Pages

Suppose each page = 4KB

Anything in page 2 has address:
0x00002xxx

Lower 12 bits specify which byte you are in the page:
0x00002200 = 0010 0000 0000
= byte 512

upper bits = page number
lower bits = page offset

Sound familiar?
Data Granularity

ISA: instruction specific: LB, LH, LW (MIPS)
Registers: 32 bits (MIPS)
Caches: cache line/block
  Address bits divided into:
  index: which entry in the cache
tag: sanity check for address match
offset: which byte in the line
Memory: page
  Address bits divided into:
  page number: which page in memory
index: which byte in the page
Program’s View of Memory

32-bit machine:
0x00000000 – 0xffffffff to play with
(modulo system reserved)

2 Interesting/Dubious Assumptions:
The machine I’m running on has 4GB of DRAM.
I am the only one using this DRAM.

These assumptions are embedded in the executable!
If they are wrong, things will break!
Recompile? Relink?
Indirection* to the Rescue!

**Virtual Memory:** a Solution for All Problems

- Each process has its own virtual address space
  - Program/CPU can access any address from 0...2^N
  - A process is a program being executed
  - Programmer can code as if they own all of memory

- On-the-fly at runtime, for each memory access
  - All accesses are *indirect* through a virtual address
    - Translate fake virtual address to a real physical address
    - Redirect load/store to the physical address

*google David Wheeler, Butler Lampson, Leslie Lamport, and Steve Bellovin
Virtual vs. Physical Address Spaces

Program #1’s Virtual Address Space

Program #2’s Virtual Address Space

Address Translation

Physical Address Space

Memory (DRAM)

DISK

- Not contiguous
- Page vs. Address?
Advantages of Virtual Memory

Easy relocation
• Loader puts code anywhere in physical memory
• Virtual mappings to give illusion of correct layout

Higher memory utilization
• Provide illusion of contiguous memory
• Use all physical memory, even physical address 0x0

Easy sharing
• Different mappings for different programs / cores

And more to come...
Takeaway

• All problems in computer science can be solved by another level of indirection.

• Need a map to translate a “fake” virtual address (generated by CPU) to a “real” physical Address (in memory)

• Virtual memory is implemented via a “Map”, a PageTable, that maps a **vaddr** (a virtual address) to a **paddr** (physical address):

• $paddr = PageTable[vaddr]$
Next Goal

• How do we implement that translation from a virtual address (vaddr) to a physical address (paddr)?

$\texttt{paddr = \text{PageTable}[vaddr]}$

• i.e. How do we implement the PageTable??
Virtual Memory Agenda

What is Virtual Memory?

How does Virtual memory Work?
  • Address Translation
  • Overhead
  • Paging
  • Performance
  • Virtual Memory & Caches
Address Translator: MMU

- Programs use virtual addresses
- Actual memory uses physical addresses

Memory Management Unit (MMU)
- HW structure
- Translates virtual → physical address on the fly
Address Translation: in Page Table

**OS-Managed** Mapping of Virtual $\rightarrow$ Physical Pages

```c
int page_table[2^{20}] = { 0, 5, 4, 1, ... };
```

```c
ppn = page_table[vpn];
```

**Remember:**

any address 0x00001234 is x234 bytes into Page C

both virtual & physical

VP 1 $\rightarrow$ PP 5

Assuming each page = 4KB
Page Table Basics

1 Page Table per process
Lives in Memory, i.e. in a page (or more...)
Location stored in Page Table Base Register

Assuming each page = 4KB
Simple Address Translation

Virtual Page Number  Page Offset

Lookup in Page Table

Physical Page Number  Page Offset

Assuming each page = 4KB
Simple Page Table Translation

Assuming each page = 4KB
General Address Translation

• What if the page size is not 4KB?
  → Page offset is no longer 12 bits

• What if Main Memory is not 4GB?
  → Physical page number is no longer 20 bits
Virtual Memory Agenda

What is Virtual Memory?
How does Virtual memory Work?
  • Address Translation
  • Overhead
  • Paging
  • Performance
  • Virtual Memory & Caches
Page Table Overhead

• How large is PageTable?
• Virtual address space (for each process):
  ▪ Given: total virtual memory: \(2^{32}\) bytes = 4GB
  ▪ Given: page size: \(2^{12}\) bytes = 4KB
  ▪ # entries in PageTable?
  ▪ size of PageTable?
  ▪ *This is one, big contiguous array, by the way!*
• Physical address space:
  ▪ Given: total physical memory: \(2^{29}\) bytes = 512MB
  ▪ overhead for 10 processes?
But Wait... There’s more!

• Page Table Entry won’t be just an integer
• Meta-Data
  ▪ Valid Bits
    • *What PPN means “not mapped”?* No such number...
    • **At first:** not all virtual pages will be in physical memory
    • **Later:** might not have enough physical memory to map all virtual pages
  ▪ Page Permissions
    • R/W/X permission bits for each PTE
    • **Code:** read-only, executable
    • **Data:** writeable, not executable
### Less Simple Page Table

<table>
<thead>
<tr>
<th>V</th>
<th>R</th>
<th>W</th>
<th>X</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0xC20A3</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0xC20A3</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>0x4123B</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>0x10044</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Physical Page Number**

Aliasing: mapping several virtual addresses → same physical page

---

Process tries to access a page without proper permissions

**Segmentation Fault**

Example:

Write to read-only? → process killed
Now how big is this Page Table?

struct pte_t page_table[2^{20}]
Each PTE = 8 bytes
How many pages in memory will the page table take up?

Assuming each page = 4KB
Takeaway

• All problems in computer science can be solved by another level of indirection.
• Need a map to translate a “fake” virtual address (generated by CPU) to a “real” physical Address (in memory)

• Virtual memory is implemented via a “Map”, a PageTage, that maps a vaddr (a virtual address) to a paddr (physical address):
  • \( paddr = \text{PageTable}[vaddr] \)

• A page is constant size block of virtual memory. Often, the page size will be around 4kB to reduce the number of entries in a PageTable.

• We can use the PageTable to set Read/Write/Execute permission on a per page basis. Can allocate memory on a per page basis. Need a valid bit, as well as Read/Write/Execute and other bits.
• But, overhead due to PageTable is significant.
Next Goal

• How do we reduce the size (overhead) of the PageTable?
Next Goal

• How do we reduce the size (overhead) of the PageTable?

• A: Another level of indirection!!
Multi-Level Page Table

*Indirection to the Rescue, AGAIN!*
Multi-Level Page Table

*Doesn’t this take up more memory than before?*

**Benefits**
- Don’t need 4MB contiguous physical memory
- Don’t need to allocate every PageTable, only those containing valid PTEs

**Drawbacks**
- Performance: Longer lookups
Virtual Memory Agenda

What is Virtual Memory?
How does Virtual memory Work?
  • Address Translation
  • Overhead
  • Paging
  • Performance
  • Virtual Memory & Caches
Paging

What if process requirements > physical memory?

*Virtual starts earning its name*

Memory acts as a cache for secondary storage (disk)

- Swap memory pages out to disk when not in use
- Page them back in when needed

Courtesy of Temporal & Spatial Locality (again!)

- Pages used recently mostly likely to be used again

More Meta-Data:

- Dirty Bit, Recently Used, *etc.*
- OS may access this meta-data to choose a victim
Example: accessing address beginning with 0x00003 (PageTable[3]) results in a Page Fault which will page the data in from disk sector 200
Page Fault

Valid bit in Page Table = 0
→ means page is not in memory

OS takes over:
• Choose a physical page to replace
  ▪ “Working set”: refined LRU, tracks page usage
• If dirty, write to disk
• Read missing page from disk
  ▪ Takes so long (~10ms), OS schedules another task

Performance-wise page faults are really bad!
Virtual Memory Agenda

What is Virtual Memory?
How does Virtual memory Work?
  • Address Translation
  • Overhead
  • Paging
  • Performance
  • Virtual Memory & Caches
Watch Your Performance Tank!

For every instruction:

• MMU translates address (virtual $\rightarrow$ physical)
  ▪ Uses PTBR to find Page Table in memory
  ▪ Looks up entry for that virtual page

• Fetch the instruction using physical address
  ▪ Access Memory Hierarchy (I$ \rightarrow$ L2 $\rightarrow$ Memory)

• Repeat at Memory stage for load/store insns
  ▪ Translate address
  ▪ **Now** you perform the load/store
Performance

• Virtual Memory Summary
• PageTable for each process:
  ▪ Page
    ▪ Single-level (e.g. 4MB contiguous in physical memory)
    ▪ or multi-level (e.g. less mem overhead due to page table),
    ▪ ...
  ▪ every load/store translated to physical addresses
  ▪ page table miss: load a swapped-out page and retry instruction, or kill program

• Performance?

• Solution?
Next Goal

• How do we speed up address translation?
Translation Lookaside Buffer (TLB)

- Small, fast cache
- Holds VPN → PPN translations
- Exploits temporal locality in pagetable
- TLB Hit: huge performance savings
- TLB Miss: invoke TLB miss handler
  - *Put translation in TLB for later*
TLB Parameters

Typical

• very small (64 – 256 entries) → *very fast*
• fully associative, or at least set associative
• tiny block size: why?

Example: Intel Nehalem TLB

• 128-entry L1 Instruction TLB, 4-way LRU
• 64-entry L1 Data TLB, 4-way LRU
• 512-entry L2 Unified TLB, 4-way LRU
TLB to the Rescue!

For every instruction:
• Translate the address (virtual → physical)
  ▪ CPU checks TLB
  ▪ That failing, walk the Page Table
    • Use PTBR to find Page Table in memory
    • Look up entry for that virtual page
    • Cache the result in the TLB

• Fetch the instruction using physical address
  ▪ Access Memory Hierarchy (I$ → L2 → Memory)

• Repeat at Memory stage for load/store insns
  ▪ CPU checks TLB, translate if necessary
  ▪ **Now** perform load/store
Virtual Memory Agenda

What is Virtual Memory?
How does Virtual memory Work?
  • Address Translation
  • Overhead
  • Paging
  • Performance
  • Virtual Memory & Caches
    • Caches use physical addresses
    • Prevents sharing except when intended
    • Works beautifully!
Recall TLB in the Memory Hierarchy

TLB is passing a physical address so we can load from memory.

What if the data is in the cache?
Virtually Addressed Caching

- Q: Can we remove the TLB from the critical path?
- A: Virtually-Addressed Caches
Virtual vs. Physical Caches

Q: What happens on context switch?
Q: What about virtual memory aliasing?
Q: So what’s wrong with physically addressed caches?
Indexing vs. Tagging

• Physically-Addressed Cache
  ▪ slow: requires TLB (and maybe PageTable) lookup first

• Virtually-Addressed Cache
  ▪ fast: start TLB lookup before cache lookup finishes
  ▪ PageTable changes (paging, context switch, etc.)
    → need to purge stale cache lines (how?)
  ▪ Synonyms (two virtual mappings for one physical page)
    → could end up in cache twice (very bad!)

• Virtually-Indexed, Physically Tagged Cache
  ▪ ~fast: TLB lookup in parallel with cache lookup
  ▪ PageTable changes → no problem: phys. tag mismatch
  ▪ Synonyms → search and evict lines with same phys. tag
Typical L1: On-chip virtually addressed, physically tagged
Typical L2: On-chip physically addressed
Typical L3: On-chip ...
Design Decisions of Caches/TLBs/VM

• Caches, Virtual Memory, & TLBs
• Where can block be placed?
  ▪ Direct, n-way, fully associative

• What block is replaced on miss?
  ▪ LRU, Random, LFU, ...

• How are writes handled?
  ▪ No-write (w/ or w/o automatic invalidation)
  ▪ Write-back (fast, block at time)
  ▪ Write-through (simple, reason about consistency)
Summary of Caches/TLBs/VM

• Caches, Virtual Memory, & TLBs
• Where can block be placed?
  ▪ Caches: direct/n-way/fully associative (fa)
  ▪ VM: fa, but with a table of contents to eliminate searches
  ▪ TLB: fa
• What block is replaced on miss?
  ▪ varied
• How are writes handled?
  ▪ Caches: usually write-back, or maybe write-through, or maybe no-write w/ invalidation
  ▪ VM: write-back
  ▪ TLB: usually no-write
## Summary of Cache Design Parameters

<table>
<thead>
<tr>
<th></th>
<th>L1</th>
<th>Paged Memory</th>
<th>TLB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Size (blocks)</strong></td>
<td>1/4k to 4k</td>
<td>16k to 1M</td>
<td>64 to 4k</td>
</tr>
<tr>
<td><strong>Size (kB)</strong></td>
<td>16 to 64</td>
<td>1M to 4G</td>
<td>2 to 16</td>
</tr>
<tr>
<td><strong>Block size (B)</strong></td>
<td>16-64</td>
<td>4k to 64k</td>
<td>4-32</td>
</tr>
<tr>
<td><strong>Miss rates</strong></td>
<td>2%-5%</td>
<td>10^{-4} to 10^{-5}%</td>
<td>0.01% to 2%</td>
</tr>
<tr>
<td><strong>Miss penalty</strong></td>
<td>10-25</td>
<td>10M-100M</td>
<td>100-1000</td>
</tr>
</tbody>
</table>
Translation in Action

Virtual Address

TLB Access

TLB Hit?

no

no

TLB miss handler (HW or OS)

yes

Physical Address

$ Access

$ Hit?

no

no

DRAM Access

DRAM Hit?

yes

deliver Data back to CPU

Next Topic:
Exceptional Control Flow
Takeaways

Need a map to translate a “fake” virtual address (from process) to a “real” physical Address (in memory).

The map is a **Page Table**: \( \text{ppn} = \text{PageTable}[\text{vpn}] \)

A page is constant size block of virtual memory. Often ~4KB to reduce the number of entries in a PageTable.

Page Table can enforce Read/Write/Execute permissions on a per page basis. Can allocate memory on a per page basis. Also need a valid bit, and a few others.

Space overhead due to Page Table is significant.
Solution: another level of indirection!
**Two-level of Page Table** significantly reduces overhead.

Time overhead due to Address Translations also significant.
Solution: caching! **Translation Lookaside Buffer (TLB)** acts as a cache for the Page Table and significantly improves performance.