Virtual Memory

Prof. Hakim Weatherspoon
CS 3410, Spring 2015
Computer Science
Cornell University

P & H Chapter 5.7 (up to TLBs)
Welcome back from Spring Break!
Where did you go?

a) Home
b) Caribbean, Hawaii, Florida, California, South America, etc
c) Europe
d) Africa
e) Asia (also middle east)
Announcements

Lab3: Available today, and due by next Wednesday

HW2: Do up to Problem5 this week. Do it now.
    Do Problem9, coding a hashtable in C, now.
Announcements

Next five weeks

• Week 10 (Apr 7): Lab3 (calling convention) release
• Week 11 (Apr 14): Proj3 (caches) release, Lab3 due Wed
• Week 12 (Apr 21): Lab4 (virtual memory) release, due in-class, Proj3 due Fri, HW2 due Sat
• Week 13 (Apr 28): Proj4 (multi-core/parallelism) release, Lab4 due in-class, Prelim2
• Week 14 (May 5): Proj3 Tournament, Proj4 design doc due

Final Project for class

• Week 15 (May 12): Proj4 due Wed
Where are we now and where are we going?
Where are we now and where are we going?

C

```c
int x = 10;
x = 2 * x + 15;
```

MIPS assemble

```assembly
addi r5, r0, 10
muli r5, r5, 2
addi r5, r5, 15
```

C compiler

```c
addi r5, r0, 10
muli r5, r5, 2
addi r5, r5, 15
```

MIPS machine code

```
00100000000001010000000000001010
00000000000001010010100001000000
001000010100101000000000001111
```

Assembler

```
op = addi r0         r5                                    10
op = addi r5         r5 15
```

CPU Circuits

Machine code

```
op = r-type               r5       r5 shamt=1     func=sll
```

Silicon Transistors Gates

```
or = r-type               r5       r5 shamt=1     func=sll
```

```
or = addi r5, r5, 15
or = r-type               r5       r5 shamt=1     func=sll
```
Where are we now and where are we going?

C

```c
int x = 10;
x = 2 * x + 15;
```

MIPS

```
addi r5, r0, 10
muli r5, r5, 2
addi r5, r5, 15
```

assembly

```
00100000000001010000000000001010
00000000000001010010100001000000
00100000101001010000000000001111
```

machine code

High Level Languages

Instruction Set Architecture (ISA)
Where are we now and where are we going?

Code Stored in Memory
(also, data and stack)

$0$ (zero)
$1$ ($at$)
$29$ ($sp$)
$31$ ($ra$)

Stack, Data, Code
Stored in Memory

Write-Back

Instruction Fetch

Instruction Decode

Forward Unit

Execute

Memory

New pc

PC

Inst

new pc

D

control

compute jump/branch targets

extend

detect hazard

PC

instruction

PC +4
Where are we now and where are we going?

Memory: big & slow  vs Caches: small & fast

Processor

| LB | $1 ← M[ 1 ] |
| LB | $2 ← M[ 5 ] |
| LB | $3 ← M[ 1 ] |
| LB | $3 ← M[ 4 ] |
| LB | $2 ← M[ 0 ] |
| LB | $2 ← M[ 12 ] |
| LB | $2 ← M[ 5 ] |
| LB | $2 ← M[ 12 ] |
| LB | $2 ← M[ 5 ] |
| LB | $2 ← M[ 12 ] |

Cache

<table>
<thead>
<tr>
<th>tag</th>
<th>data</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>tag</th>
<th>data</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>tag</th>
<th>data</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>tag</th>
<th>data</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>tag</th>
<th>data</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>tag</th>
<th>data</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>tag</th>
<th>data</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>tag</th>
<th>data</th>
</tr>
</thead>
</table>

Memory

Memory: big & slow  vs Caches: small & fast

- Stack
- Heap
- Data
- Text

Processor Memory

Misses:

Hits:

Memory: big & slow  vs Caches: small & fast
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?

a) Yes, no problem at all
b) No, because memory addresses will conflict
c) Yes, caches can avoid memory address conflicts
d) Yes, our modified Harvard architecture avoids memory address conflicts
e) Yes, because we have multiple processors (multiple cores)
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

Processor

Memory

Cache

Misses:

Hits:

tag
data

$0

$1

$2

$3

0x000...0

0x7ff...f

0xfff...f

Stack

Heap

Data

Text
Processor & Memory

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

```
<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text</td>
</tr>
<tr>
<td>Data</td>
</tr>
<tr>
<td>Heap</td>
</tr>
<tr>
<td>Stack</td>
</tr>
<tr>
<td>0x000...0</td>
</tr>
<tr>
<td>0x7ff...f</td>
</tr>
<tr>
<td>0xffff...f</td>
</tr>
</tbody>
</table>
```

CPU address/data bus... routed through caches to main memory.
Q: What happens when another program is executed concurrently on another processor?

A: The addresses will conflict
  • Even though, CPUs may take turns using memory bus
Q: Can we relocate second program?
Q: Can we relocate second program?
A: Yes, but...

- What if they don’t fit?
- What if not contiguous?
- Need to recompile/relink?
- ...

Solution? Multiple processes/processors
Q: Can we relocate second program?
A: Yes, but...

- What if they don’t fit?
- What if not contiguous?
- Need to recompile/relink?
- ...

Solution: Need a **MAP**
To map a **Virtual Address** (generated by CPU) to a **Physical Address** (in memory)
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)
Goals for Today: Virtual Memory

What is Virtual Memory?

How does Virtual memory Work?

- Address Translation
  - Pages, page tables, and memory mgmt unit
- Paging
- Role of Operating System
  - Context switches, working set, shared memory
- Performance
  - How slow is it
  - Making virtual memory fast
  - Translation lookaside buffer (TLB)
- Virtual Memory Meets Caching
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?
Virtual Memory

Virtual Memory: A Solution for All Problems

- Program/CPU can access any address from $0...2^N$ (e.g. $N=32$) 
  (so addresses 0 to $2^{32}$; 0x0000 0000 to 0xFFFF FFFF)

Each process has its own virtual address space

- 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

map
- all accesses are *indirect* through a virtual address
- translate fake virtual address to a real physical address
- redirect load/store to the physical address
Virtual Address Space

Physical Address Space

Address Translation
Programs load/store to virtual addresses

Actual memory uses physical addresses

**Memory Management Unit (MMU)**
- Responsible for translating on the fly
- Essentially, just a big array of integers:
  \[
  \text{paddr} = \text{PageTable}[	ext{vaddr}];
  \]
Virtual Memory Advantages

Advantages

Easy relocation

• Loader puts code anywhere in physical memory
• Creates *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...
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):

\[ paddr = \text{PageTable}[\text{vaddr}] \]
Next Goal

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

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

i.e. How do we implement the PageTable??
Address Translation

Pages, Page Tables, and the Memory Management Unit (MMU)
How large should a PageTable be for a MMU?

\[
paddr = \text{PageTable}[\text{vaddr}]\;
\]

Granularity?

- Per word... \(2^{32} / 4 = 2^{30} = 1 \text{ billion pagetable entries}\)
- Per block...
- Variable.....

Typical:

- 4KB – 16KB pages
- 4MB – 256MB jumbo pages
Attempt #1: Address Translation

31  12  11  0
Virtual page number                  Page Offset

lookup in PageTable

12  11  0
Physical page number                Page offset

CPU generated

Main Memory
e.g. Page size

4 kB = 2^{12}

Attempt #1: For any access to virtual address:

- Calculate **virtual page number** and **page offset**
- Lookup **physical page number** at PageTable[vpn]
- Calculate physical address as ppn:offset
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):

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

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.
Example

How to translate a vaddr (virtual address) generated by the CPU to a paddr (physical address) used by main memory using the PageTable managed by the memory management unit (MMU).
Next Goal

Example

How to translate a `vaddr` (virtual address) generated by the CPU to a `paddr` (physical address) used by main memory using the PageTable managed by the memory management unit (MMU).

Q: Where is the PageTable stored??
Q: Where to store page tables?
a) In memory
b) In cache
c) In registers
d) On disk
e) None of the above
Q: Where to store page tables?
A: In memory, of course...

Special page table base register
(CR3:PTBR on x86)
(Cop0:ContextRegister on MIPS)
Simple Page Table

<table>
<thead>
<tr>
<th>Physical Page Number</th>
<th>vaddr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10045</td>
<td>0xC20A3000</td>
</tr>
<tr>
<td>0xC20A3</td>
<td>0x90000000</td>
</tr>
<tr>
<td>0x4123B</td>
<td>0x4123B000</td>
</tr>
<tr>
<td>0x10044</td>
<td>0x10045000</td>
</tr>
</tbody>
</table>

vpn | pgoff | vaddr
---|-------|-------
0x10045 | 0xC20A3 | 0xC20A3000
0xC20A3 | 0x90000000 | 0x90000000
0x4123B | 0x4123B000 | 0x4123B000
0x10044 | 0x10045000 | 0x10045000

PTBR
### Simple PageTable

<table>
<thead>
<tr>
<th>Physical Page Number</th>
<th>paddr</th>
<th>vaddr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10045</td>
<td>0x4123B</td>
<td>0x00002</td>
</tr>
<tr>
<td>0xC20A3</td>
<td>0x4123B</td>
<td>0xAABC</td>
</tr>
<tr>
<td>0x4123B</td>
<td>0x4123B</td>
<td>0xAABC</td>
</tr>
<tr>
<td>0x10044</td>
<td>0xC20A3</td>
<td>0xAABC</td>
</tr>
</tbody>
</table>

**PTBR**

- 0x00000000
- 0x90000000
- 0xC20A3000
- 0x10045000
- 0x10044000
- 0x4123B000

**vaddr**

- 0x00002
- 0xAABC

**paddr**

- 0x4123B
- 0xAABC
**Cool Trick #1:** Don’t map all pages

Need **valid bit** for each page table entry

**Q:** Why?

**A:** e.g. access to NULL will fail

**A:** we might not have that much physical memory

---

<table>
<thead>
<tr>
<th>Physical Page Number</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10045</td>
<td>1</td>
</tr>
<tr>
<td>0xC20A3000</td>
<td></td>
</tr>
<tr>
<td>0x90000000</td>
<td></td>
</tr>
<tr>
<td>0x4123B000</td>
<td></td>
</tr>
<tr>
<td>0x10044000</td>
<td></td>
</tr>
<tr>
<td>0x00000000</td>
<td>0</td>
</tr>
</tbody>
</table>
Cool Trick #2: Page permissions!

Keep R, W, X permission bits for each page table entry

Q: Why?

A: can make code read-only, executable; make data read-write but not executable; etc.
Cool Trick #3: **Aliasing**

Map the same physical page at several virtual addresses

**Q**: Why?

**A**: can make different views of same data with different permissions
Overhead for VM Attempt #1: How large is PageTable?

Virtual address space (for each process):

- total virtual memory: $2^{32}$ bytes = 4GB
- page size: $2^{12}$ bytes = 4KB
- entries in PageTable? $2^{20} = 1$ million entries in PageTable
- size of PageTable? Size of PageTable Entry (PTE) size = 4 bytes
  So, PageTable size = $4 \times 2^{20} = 4$MB

Physical address space:

- total physical memory: $2^{29}$ bytes = 512MB
- overhead for 10 processes?
  $10 \times 4$MB = 40 MB of overhead!
  - 40 MB /512 MB = 7.8% overhead, space due to PageTable
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):

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

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

A: Another level of indirection!!
Beyond Flat Page Tables
Assume most of PageTable is empty

How to translate addresses? Multi-level PageTable

* x86 does exactly this
Beyond Flat Page Tables

Assume most of PageTable is empty

How to translate addresses? Multi-level PageTable

Q: Benefits?
A: Don’t need 4MB contiguous physical memory
A: Don’t need to allocate every PageTable, only those containing valid PTEs

Q: Drawbacks
A: Performance: Longer lookups
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):

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

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.

Another level of indirection, two levels of PageTables and significantly reduce the overhead due to PageTables.
Next Goal

Can we run process larger than physical memory?
Paging
Paging

Can we run process larger than physical memory?

- The “virtual” in “virtual memory”

View memory as a “cache” for secondary storage

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

Assumes Temporal/Spatial Locality

- Pages used recently most likely to be used again soon
Cool Trick #4: Paging/Swapping

Need more bits:
Dirty, RecentlyUsed, ...
Summary

Virtual Memory

• Address Translation
  • Pages, page tables, and memory mgmt unit
• Paging

Next time

• Role of Operating System
  • Context switches, working set, shared memory
• Performance
  • How slow is it
  • Making virtual memory fast
  • Translation lookaside buffer (TLB)
• Virtual Memory Meets Caching
MIPS Processor Milestone Celebration!