CS 152
Computer Architecture and Engineering
Lecture 12 -- Virtual Memory

2014-2-27
John Lazzaro
(not a prof - “John” is always OK)

TA: Eric Love

www-inst.eecs.berkeley.edu/~cs152/
Virtual address spaces
Page table layout
TLB design options
Virtual machines
Exceptions and interrupts
The Limits of Physical Addressing

“Physical addresses” of memory locations

CPU
A0-A31
D0-D31

Memory
A0-A31
D0-D31

Where we are in CS 152 ...

All programs share one address space:
The physical address space

Machine language programs must be aware of the machine organization

No way to prevent a program from accessing any machine resource
Apple II: A physically addressed machine

Apple ][ (1977)
CPU: 1000 ns
DRAM: 400 ns

Steve Jobs
Steve Wozniak

RAM Complement | Apple II System
---|---
4K | $1,298.00
48K | $2,638.00
Apple II: A physically addressed machine

Apple ][ (1977)

<table>
<thead>
<tr>
<th>RAM Complement</th>
<th>Apple II System</th>
</tr>
</thead>
<tbody>
<tr>
<td>4K</td>
<td>$1,298.00</td>
</tr>
<tr>
<td>48K</td>
<td>2,638.00</td>
</tr>
</tbody>
</table>

**RAM Organization and Usage**

<table>
<thead>
<tr>
<th>Page #</th>
<th>Dec</th>
<th>Hex</th>
<th>Used for</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>$00</td>
<td>System Programs</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$01</td>
<td>System Stack</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>$02</td>
<td>Input Buffer</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>$03</td>
<td>Monitor Vector Locations</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>$04</td>
<td>Text/Lo-Res Graphics</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>$05</td>
<td>Text/Lo-Res Graphics</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>$06</td>
<td>Primary Page Storage</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>$07</td>
<td>Secondary Page Storage</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>$08</td>
<td>Text/Lo-Res Graphics</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>$09</td>
<td>Text/Lo-Res Graphics</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>$0A</td>
<td>Secondary Page Storage</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>$0B</td>
<td>FREE RAM</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>$0C  through $1F</td>
<td>FREE RAM</td>
</tr>
<tr>
<td>31</td>
<td>31</td>
<td>$1F</td>
<td>FREE RAM</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>$20  through $3F</td>
<td>FREE RAM</td>
</tr>
<tr>
<td>63</td>
<td>63</td>
<td>$3F</td>
<td>Hi-Res Graphics Primary Page Storage</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>$40  through $5F</td>
<td>FREE RAM</td>
</tr>
<tr>
<td>95</td>
<td>95</td>
<td>$5F</td>
<td>Hi-Res Graphics Secondary Page Storage</td>
</tr>
<tr>
<td>96</td>
<td>96</td>
<td>$60  through $B7F</td>
<td>FREE RAM</td>
</tr>
<tr>
<td>191</td>
<td>191</td>
<td>$B7F</td>
<td>FREE RAM</td>
</tr>
<tr>
<td>192</td>
<td>192</td>
<td>$C0</td>
<td>I/O and soft switches</td>
</tr>
<tr>
<td>193</td>
<td>193</td>
<td>$C1  through $C7</td>
<td>I/O shared ROM space</td>
</tr>
<tr>
<td>199</td>
<td>199</td>
<td>$C7</td>
<td></td>
</tr>
</tbody>
</table>
The Limits of Physical Addressing

"Physical addresses" of memory locations

CPU
A0-A31
D0-D31

Programming the Apple ][ ...

Memory
A0-A31
D0-D31

Data

All programs share one address space:
The **physical** address space

Machine language programs must be aware of the machine organization

No way to prevent a program from accessing any machine resource
Solution: Add a Layer of Indirection

User programs run in an standardized virtual address space

Address Translation hardware managed by the operating system (OS)
maps virtual address to physical memory

Hardware supports “modern” OS features: Protection, Translation, Sharing
MIPS R4000: Address Space Model

Process A

ASID = 12

Process A and B have independent address spaces

All address spaces use a standard memory map

May only be accessed by kernel/supervisor

When Process A writes its address 9, it writes to a different physical memory location than Process B’s address 9

To let Process A and B share memory, OS maps parts of ASID 12 and ASID 13 to the same physical memory locations.

Still works (slowly!) if a process accesses more virtual memory than the machine has physical memory
System Control Registers

Status (12): Indicates user, supervisor, or kernel mode

EntryLo0 (2): 8-bit ASID field codes virtual address space ID.

User cannot write supervisor/kernel bits.
Supervisor cannot write kernel bit.

User cannot change address translation configuration
or run other privileged instructions...
MIPS Address Translation: How it works

Translation Look-Aside Buffer (TLB)
A small fully-associative cache of mappings from virtual to physical addresses

TLB also contains ASID and kernel/supervisor bits for virtual address

Fast common case: Virtual address is in TLB, process has permission to read/write it.
A virtual address space is divided into blocks of memory called **pages**.

A machine usually supports pages of a few sizes (MIPS R4000):

<table>
<thead>
<tr>
<th>Page Size</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4 Kbytes</td>
<td></td>
</tr>
<tr>
<td>16 Kbytes</td>
<td></td>
</tr>
<tr>
<td>64 Kbytes</td>
<td></td>
</tr>
<tr>
<td>256 Kbytes</td>
<td></td>
</tr>
<tr>
<td>1 Mbyte</td>
<td></td>
</tr>
<tr>
<td>4 Mbytes</td>
<td></td>
</tr>
<tr>
<td>16 Mbytes</td>
<td></td>
</tr>
</tbody>
</table>

A page table is indexed by a virtual address.

A valid page table entry codes physical memory “frame” address for the page.
The TLB caches page table entries

TLB caches page table entries.

Virtual Address

V page no.  offset

Page Table

Page Table for ASID

index into page table

physical address

V

Access Rights

PA

Physical Address

P page no.  offset

Virtual Memory

Physical frame address

In this example, physical and virtual pages must be the same size!

V=0 pages either reside on disk or have not yet been allocated. OS handles V=0 "Page fault"

MIPS handles TLB misses in software (random replacement). Other machines use hardware.
Page tables may not fit in memory!

A table for 4KB pages for a 32-bit address space has 1M entries

Each process needs its own address space!

Two-level Page Tables

32 bit virtual address

Top-level table wired in main memory

Subset of 1024 second-level tables in main memory; rest are on disk or unallocated
What if a page resides on disk?

Virtual Address

V page no. offset

Page Table

Page Table

index into page table

V Access Rights PA

table located in physical memory

Physical frame address

P page no. offset

Physical Address

V=0 pages either reside on disk or have not yet been allocated. OS handles V=0 “Page fault”

Question: What to do when a TLB miss causes an access to a page table entry with V=0?
VM and Disk: Page replacement policy

Set of all pages in Memory

Head pointer
Place pages on free list if used bit is still clear. Schedule pages with dirty bit set to be written to disk.

Tail pointer: Clear the used bit in the page table

Dirty bit: page has been written.
Used bit: set to 1 on any reference

On page fault: deallocate page table entry of a page on the free list.

Freelist
Free Pages

Architect’s role: support setting dirty and used bits
TLB Design Concepts
MIPS R4000 TLB: A closer look ...

"Virtual Addresses"

CPU

A0-A31

D0-D31

"Physical Addresses"

Memory System

A0-A31

D0-D31

Translation Look-Aside Buffer (TLB)

Virtual

Physical

Data

Checked against CPU ASID

Physical space larger than virtual space!

Virtual Address with 1M \(2^{20}\) 4-Kbyte pages

39 32 31 29 28 20 bits = 1M pages 12 11 0

ASID

VPN

Offset

Bits 31, 30 and 29 of the virtual address select user, supervisor, or kernel address spaces.

Virtual-to-physical translation in TLB

Offset passed unchanged to physical memory

36-bit Physical Address

35

PFN

Offset

UC Regents Fall 2006 © UCB

CS 152 L15: Virtual Memory
Can TLB and caching be overlapped?

This works, but ...

Q. What is the downside?

A. Inflexibility. VPN size locked to cache tag size.
Can we cache virtual addresses?

Only use TLB on a cache miss!

Downside: A subtle problem. What is it?

A. Synonym problem. If two address spaces share a physical frame, data may be in cache twice. Maintaining consistency is tricky.

Virtual Memory Recap

VM: Uniform memory models, protection, sharing.

A TLB acts as a fast cache for recent address translations.

Operating systems manage the page table and (often) the TLB.
Break
Running Windows on a Mac

2006 edition ...

Depends on the meaning of the word “run” ...
Method #1: Boot Camp

Basic Idea: New Macs use
Intel CPU and support chips.
So, set up boot ROM to let
you choose Win or OS X.

+++ Great compatibility.
   Just add device drivers.

+++ No performance hit:
   full-speed, use all RAM, etc.

--- Must reboot to change OS.

--- Sharing files between OS
    partitions securely is tricky.
Method #2: Run WINE on OS X

Basic Idea: Emulate the Windows API in software running under OS X. Lets you run Windows apps in a “compatibility box” without running Windows.

+++ Do not need to buy Windows.
+++ No reboot to run Win-app.
--- Slow.
--- Chances are, the app you want to run has compatibility woes.
Method 3: Virtual PC

Basic Idea: Make a software emulation of PC hardware. Runs as a user process under OS X. Boot Windows and run apps on the emulator.

+++ Runs on PowerPC Macs.
+++ Good compatibility. Easier to emulate CPU than Win API.
--- Must buy Windows.
--- Slow
Emulating a PC --> emulating everything!

Windows expects to see raw disks, so VirtualPC has **Virtual Disks**.

Windows expects to set up a graphics card, to VirtualPC has a **Virtual GPU**.

Windows expects to manipulate page tables, so VirtualPC has **Virtual TLB**.

Windows expects to configure network: **Virtual Ethernet Card**.

Like the movie “The Truman Show” ... no wonder it's slow.
Method 4: Parallels, a Virtual Machine

Basic Idea: Like emulating a PC, but different. Use an Intel-based Mac, runs on top of OS X. Uses hardware support to create a fast virtual PC that boots Windows.

+++ Reasonable performance.

Virtual CPU runs 33% slower than running on physical CPU.

(2006 data)

2 GB physical memory for a 512 MB virtual PC to run w/o disk swaps.

Source: http://www.atpm.com/12.10/parallels.shtml
“Hardware assist?” What do we mean?

In an emulator, we run Windows code by *simulating* the CPU in software.

In a virtual machine, we let *safe* instructions (e.g., ADD R3 R2 R1) run on the actual hardware in user mode.

We use hardware features to prevent direct execution of *unsafe* instructions (e.g., change a page table entry).

We *trap* each attempt, and emulate the instruction in software, in a safe way.

*A “trap” is one type of “exception”*...
Exceptions and Interrupts

Exception: An unusual event happens to an instruction during its execution. Examples: divide by zero, undefined opcode.

Interrupt: Hardware signal to switch the processor to a new instruction stream. Example: a sound card interrupts when it needs more audio output samples (an audio “click” happens if it is left waiting).
Challenge: Precise Interrupt / Exception

**Definition:**

*It must appear as if an interrupt is taken between two instructions* (say $I_i$ and $I_{i+1}$)

- the effect of all instructions up to and including $I_i$ is totally complete
- no effect of any instruction after $I_i$ has taken place

The interrupt handler either aborts the program or restarts it at $I_{i+1}$.

*Follows from the contract between the architect and the programmer* ...
Key observation: architected state only changes in memory and register write stages.
Adding trap support to pipelines ...

Detect @ decode, set an Exc E bit

Select Handler PC

PC Address Exceptions

PC

Inst. Mem

D

Decode

E

+ M

Data Mem

W

Illegal Opcode

Overflow

Data Addr Except

Kill Writeback

Commit Point

Cause

EPC

Select Handler PC

PC Address Exceptions

PC

Inst. Mem

D

Illegal Opcode

Overflow

Data Addr Except

Kill Writeback

Commit Point

Cause

EPC

Detect @ decode, set an Exc E bit

PC

Inst. Mem

D

Illegal Opcode

Overflow

Data Addr Except

Kill Writeback

Commit Point

Cause

EPC

Call the code to be run on this trap type. Pass along Cause and EPC as arguments.

PC

Inst. Mem

D

Illegal Opcode

Overflow

Data Addr Except

Kill Writeback

Commit Point

Cause

EPC

Call the code to be run on this trap type. Pass along Cause and EPC as arguments.
Virtual Machines: Better as servers than clients

Run large-scale workloads on virtual machines hosted on Google's infrastructure. Choose a VM that fits your needs and gain the performance of Google's worldwide fiber network.

Get Started
Google runs Linux servers running the KVM module (Kernel Virtual Machine Monitor) and spins up VMs with varying specs on demand.

<table>
<thead>
<tr>
<th>Instance type</th>
<th>Virtual Cores</th>
<th>Memory</th>
<th>Price (US$)/Hour (US hosted)</th>
</tr>
</thead>
<tbody>
<tr>
<td>n1-standard-1</td>
<td>1</td>
<td>3.75GB</td>
<td>$0.104</td>
</tr>
<tr>
<td>n1-standard-2</td>
<td>2</td>
<td>7.5GB</td>
<td>$0.207</td>
</tr>
<tr>
<td>n1-standard-4</td>
<td>4</td>
<td>15GB</td>
<td>$0.415</td>
</tr>
<tr>
<td>n1-standard-8</td>
<td>8</td>
<td>30GB</td>
<td>$0.829</td>
</tr>
<tr>
<td>n1-standard-16</td>
<td>16</td>
<td>60GB</td>
<td>$1.659</td>
</tr>
</tbody>
</table>
$2/hour for a 16-virtual-core 104GB machine ...

### High Memory
Machines for tasks that require more memory relative to virtual cores

<table>
<thead>
<tr>
<th>Instance type</th>
<th>Virtual Cores</th>
<th>Memory</th>
<th>Price (US$)/Hour (US hosted)</th>
</tr>
</thead>
<tbody>
<tr>
<td>n1-highmem-2</td>
<td>2</td>
<td>13GB</td>
<td>$0.244</td>
</tr>
<tr>
<td>n1-highmem-4</td>
<td>4</td>
<td>26GB</td>
<td>$0.488</td>
</tr>
<tr>
<td>n1-highmem-8</td>
<td>8</td>
<td>52GB</td>
<td>$0.975</td>
</tr>
<tr>
<td>n1-highmem-16</td>
<td>16</td>
<td>104GB</td>
<td>$1.951</td>
</tr>
</tbody>
</table>

### High CPU
Machines for tasks that require more virtual cores relative to memory

<table>
<thead>
<tr>
<th>Instance type</th>
<th>Virtual Cores</th>
<th>Memory</th>
<th>Price (US$)/Hour (US hosted)</th>
</tr>
</thead>
<tbody>
<tr>
<td>n1-highcpu-2</td>
<td>2</td>
<td>1.80GB</td>
<td>$0.131</td>
</tr>
<tr>
<td>n1-highcpu-4</td>
<td>4</td>
<td>3.60GB</td>
<td>$0.261</td>
</tr>
<tr>
<td>n1-highcpu-8</td>
<td>8</td>
<td>7.20GB</td>
<td>$0.522</td>
</tr>
<tr>
<td>n1-highcpu-16</td>
<td>16</td>
<td>14.40GB</td>
<td>$1.044</td>
</tr>
</tbody>
</table>
Shell command for managing VMs ...

Command-line arguments set RAM, disk, network ... and the OS to install.

VMs can be monitored and reconfigured on the fly.

# virt-install
  --name kvm1 \
  --ram 500 \n  --disk path=/var/lib/libvirt/images/kvm1.img,size=5 \n  --network network:default \n  --accelerate \n  --vnc \n  -c /tmp/SLES11-x86_64-DVD.iso

# virsh nodeinfo
CPU model: x86_64
CPU(s): 8
CPU frequency: 1596 MHz
CPU socket(s): 2
Core(s) per socket: 4
Thread(s) per core: 1
NUMA cell(s): 1
Memory size: 8150212 kB

- To set the running memory of a KVM:
  # virsh setmem <Name of KVM> <Amount of memory in KB>
Under the hood

What Google customers “think” is happening.

When the guest OS tries to privileged instructions, host kernel intercepts.

What actually happens.

Where Google runs “make new VM” commands
Example: Paging

Guest OS runs instructions that manipulate page tables, and as far as it can tell everything works OK.

Hypervisor watches every Guest OS move, and updates the “shadow” page tables and TLB to work correctly.

Exists to “fake out” guest OS.
Hardware Support

Intel VT-x adds a new level of privilege where hypervisors can run (VMX root).

Guest OS’s run in “kernel mode” (ring 0) like they do on bare metal. Hypervisor relieved from faking guest privilege levels.
Hardware Support

Intel VT-x supports Extended Page Tables (EPT).

The TLB and the page replacement hardware tracks the shadow/guests page mappings.
A work in progress

Each generation of Intel server adds new features for virtualization, and improves the latency for getting in and out of VMX root mode for critical operations.
On Tuesday

Memory semantics for multi-core ...

<table>
<thead>
<tr>
<th>T</th>
<th>Synchronization + Sequential Consistency</th>
<th>Chapter 5.5-6.</th>
</tr>
</thead>
<tbody>
<tr>
<td>3/4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Have a good weekend!