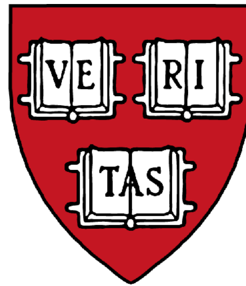
CS161: Operating Systems

Matt Welsh
mdw@eecs.harvard.edu



Lecture 21: Virtual Machines
April 24, 2007

© 2006 Matt Welsh – Harvard University

Introduction

This talk is an overview on modern virtual machines: VMWare and Xen.

Very cool stuff. Lots of exciting research happening in this area right now.

© 2006 Matt Welsh – Harvard University

Today: Virtual Machines

Operating systems abstract the physical hardware in certain ways.

- Thread abstraction: Each thread has its own "virtual CPU"
- Process and address space: Each process has its own "virtual memory"
- Filesystems: Each process has its own "virtual disk"

But the OS interface is very different than the physical machine

- Programs have to use system calls to request services from the OS
- OS interface is much higher-level than the physical machine interface.

Example: Global names

- All processes on a machine can see the same filesystem
  - *Though they may not have access to all of it.*
- This implies that processes are not truly *isolated* from each other.

© 2006 Matt Welsh – Harvard University

Virtual Machine Concept

Can we introduce a layer above the hardware that:

- Provides an illusion of a complete (virtual) physical machine?
- Allows multiple operating systems to run on the machine at once?
- Provides complete isolation and protection between VMs?

Linux

FreeBSD

Windows XP

Virtual machine (VM)

Virtual machine (VM)

Virtual machine (VM)

Virtual Machine Monitor (VMM)

© 2006 Matt Welsh – Harvard University

Why would we want to do this??

© 2006 Matt Welsh – Harvard University

Why would we want to do this??

Let Linux geeks like me run Windows without having to reboot.

- OK, kinda useful...

Big hosting centers want to host many different customers

- Some customers want Linux, others Windows, others have custom OS, some have specific kernel modifications that they need...
- Want to avoid buying separate machine for each customer application

Experiment with new OS features/code

- The virtual machine monitor (VMM) keeps the OS in a “sandbox”
- Used to teach OS at UW (students hack Linux running in VMWare)

Safely push complete OS+application environment “into the network”

- Akamai mainly caches static Web page content at many sites on the Internet
- What if they could run little “mini Amazons” or “mini Yahoos” for their customers?

Protection from viruses/worms/etc.

- Imagine: Every time you open an email attachment, fork a complete copy of your running OS and application stack in a virtual machine!!!

© 2006 Matt Welsh – Harvard University

Virtual Machine Goals

Must give each virtual machine the *complete* illusion of a real machine

- With I/O devices, interrupts, virtual memory hardware, protection levels, etc...

Must be 100% compatible with existing hardware

- Run existing, unmodified operating systems and applications directly on the VM!!

Must completely isolate VMs from each other

- Should not allow one VM to stomp on another's memory or CPU state

Must have high performance

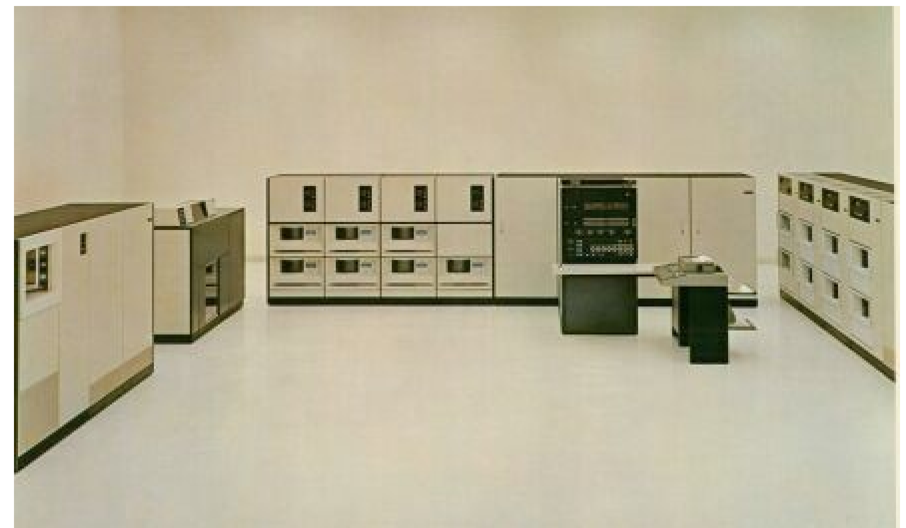
- Otherwise nobody will want to use it.

© 2006 Matt Welsh – Harvard University

IBM VM/370

One of the first true Virtual Machine Monitors

- Developed in 1960's for IBM System/360 and System/370 mainframes



© 2006 Matt Welsh – Harvard University

IBM VM/370

One of the first true Virtual Machine Monitors

- Developed in 1960's for IBM System/360 and System/370 mainframes

Allow one System/370 to be shared by several simultaneous users.

- Back then, timesharing was in its infancy: Standard S/370 OS did not allow multiple concurrent jobs

Why use VMMs rather than a “modern” process abstraction?

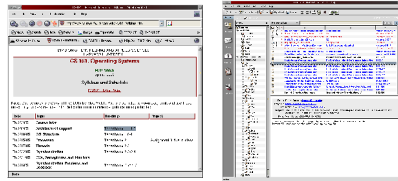
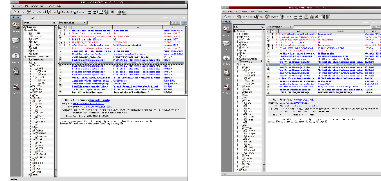
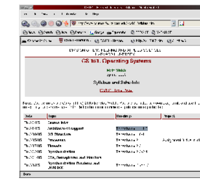
- Many user programs coded for the bare hardware
- Existing apps did not assume any protection: only one app running at a time.
- Different applications used different “Operating Systems” (more typically, libraries and routines for talking to hardware).
  - *No agreement on one universal OS for everyone.*
- So ... it was more natural to virtualize the *entire hardware* of the machine!!!

© 2006 Matt Welsh – Harvard University

VMWare

VMWare Workstation: Runs as a regular user app

- Not as a VMM running on bare hardware!

Linux	FreeBSD	Windows XP

Virtual machine (VM) | Virtual machine (VM) | Virtual machine (VM)

VMWare (Running as an application)

Host OS (e.g., Linux, Windows, etc.)

© 2006 Matt Welsh – Harvard University

Challenges: Why is this hard?

Guest OS needs to call privileged instructions

- e.g., to perform I/O, manipulate hardware state, etc.
- Should we let it do this directly??

Guest OS needs to manipulate page tables

- In order to set up virtual->physical mappings for its applications
- Why is this potentially a bad idea?

Guest OS needs to believe it's running on a real machine

- Must support complete instruction set of processor
- Must support all the weird virtual memory features (segments, paging, etc.)
- Must support (at least some) “real” I/O devices (disk, network, etc.)

© 2006 Matt Welsh – Harvard University

User vs. Supervisor mode

What does this all imply?

That the VM cannot run in the true “supervisor” mode of the machine!

- That is, we need to run the VM using the regular “user mode” of the processor.
- This is the only way to ensure that the Guest OS doesn't do anything nasty.

One implementation: Interpretation

- The VMM is simply a software layer that interprets every CPU instruction executed by the VM
- Can safely emulate privileged instructions
- All machine accesses (CPU execution, memory access, I/O) go through VMM
- So ... this approach works!!!

So why is this a bad idea?

Also ... VMM has to accurately emulate the entire CPU architecture (a real pain).

© 2006 Matt Welsh – Harvard University

Direct Execution

Another approach: Let the Guest OS run directly on the machine

- But ... in ***user mode*** (not supervisor mode).

This is a LOT faster than interpretation (obviously).

What about privileged instructions?

- Hmmm.... what if the Guest OS needs to do some I/O? Or mess with the page tables?

What happens if you try to run a privileged instruction in user mode?

© 2006 Matt Welsh – Harvard University

Direct Execution

Another approach: Let the Guest OS run directly on the machine

- But ... in ***user mode*** (not supervisor mode).

This is a LOT faster than interpretation (obviously).

What about privileged instructions?

- Hmmm.... what if the Guest OS needs to do some I/O?

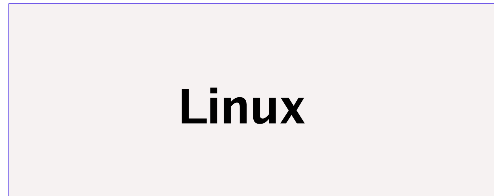
What happens if you try to run a privileged instruction in user mode?

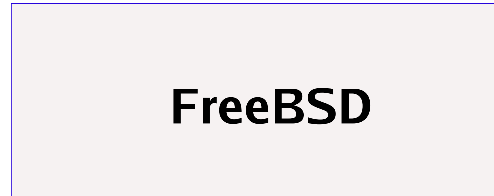
- The CPU causes a *trap* to the “real” OS (the VMM in this case).
- Trap includes the address of the faulting instruction
- VMM can inspect the instruction and decide what to do:
  - *Emulate the operation safely and restart the VMM*
  - *Disallow the operation (kill the VM)*

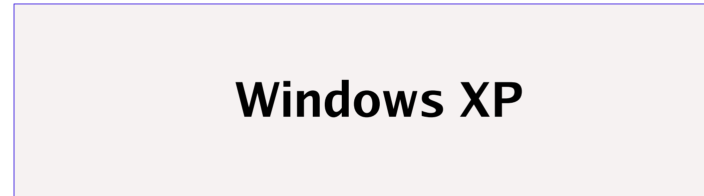
© 2006 Matt Welsh – Harvard University

VMM Trap Example

1) Linux calls OUT instruction to write to I/O device





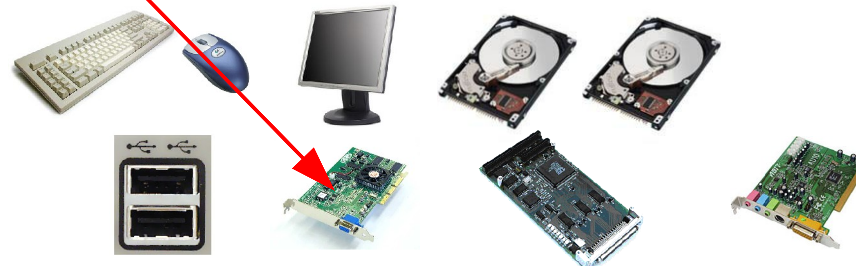



```
mov dx, ioport_id
out dx, al
```

3) VMM checks instruction (i.e., is the virtual I/O port accessible to Linux)

2) CPU issues protection fault

4) VMM issues real OUT instruction (with the correct hardware port ID)



```
mov dx, real_ioport_id
out dx, al
```

© 2006 Matt Welsh – Harvard University

Enforcing Protection

Now that OS and user apps are both running as “regular” programs, how do we protect them from each other??

- For example, how can we prevent the OS from stomping on the kernel's memory?
  - *Recall, in Linux, the OS is mapped into the user address space*

This is where multiple protection levels come in useful...

The x86 has four distinct protection levels (or “rings”)

- Idea: Run user code in Ring 3
- Run VMM in Ring 0
- Run Guest OS in Ring 1

© 2006 Matt Welsh – Harvard University

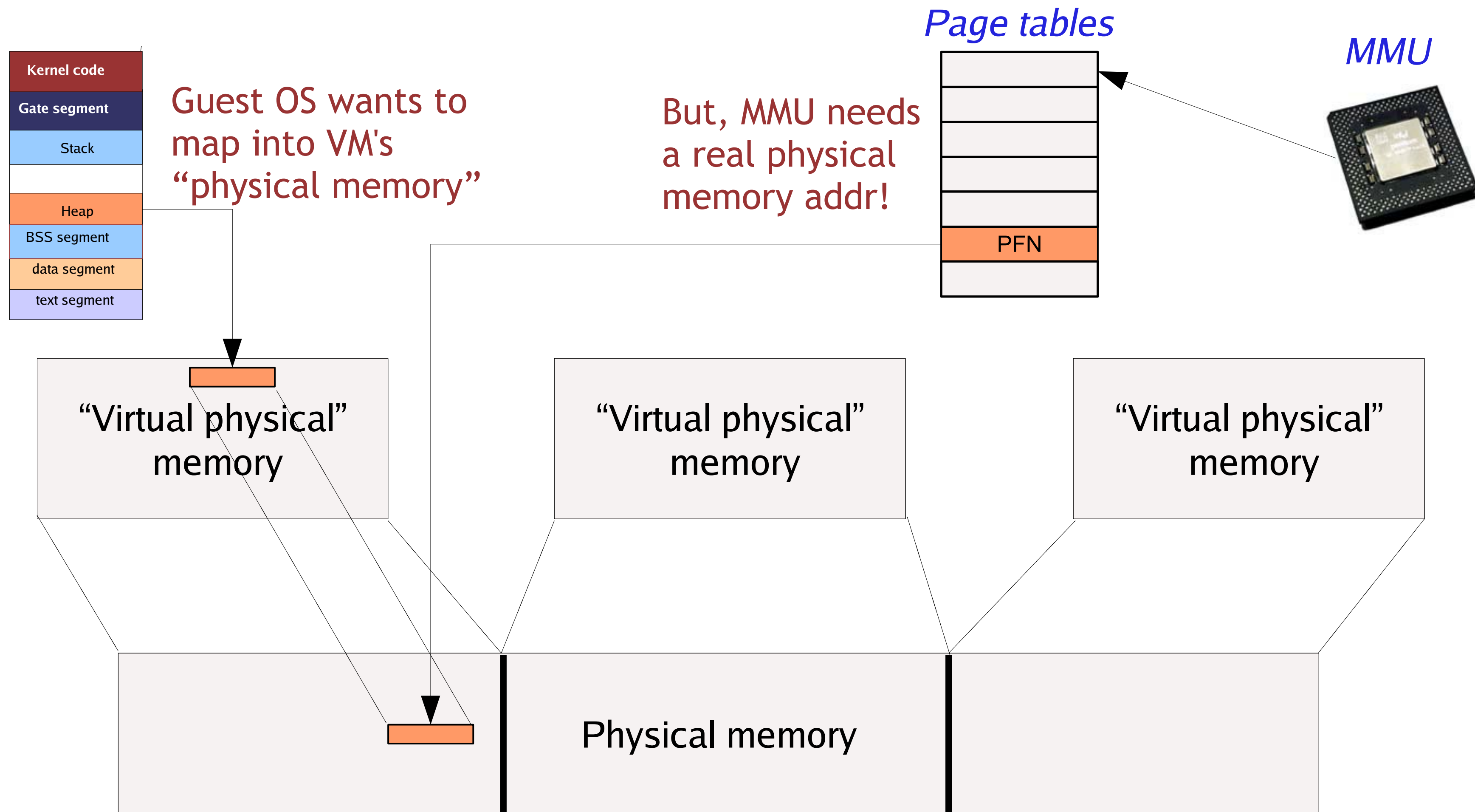
What about the MMU?

The Guest OS needs to install its own page tables

- Otherwise the MMU wouldn't be able to translate virtual addresses for the VM!

However, it's not safe to let the guest OS install any PTEs it likes.

- Otherwise it could map any physical memory page it wants.
- This breaks isolation between VMs.

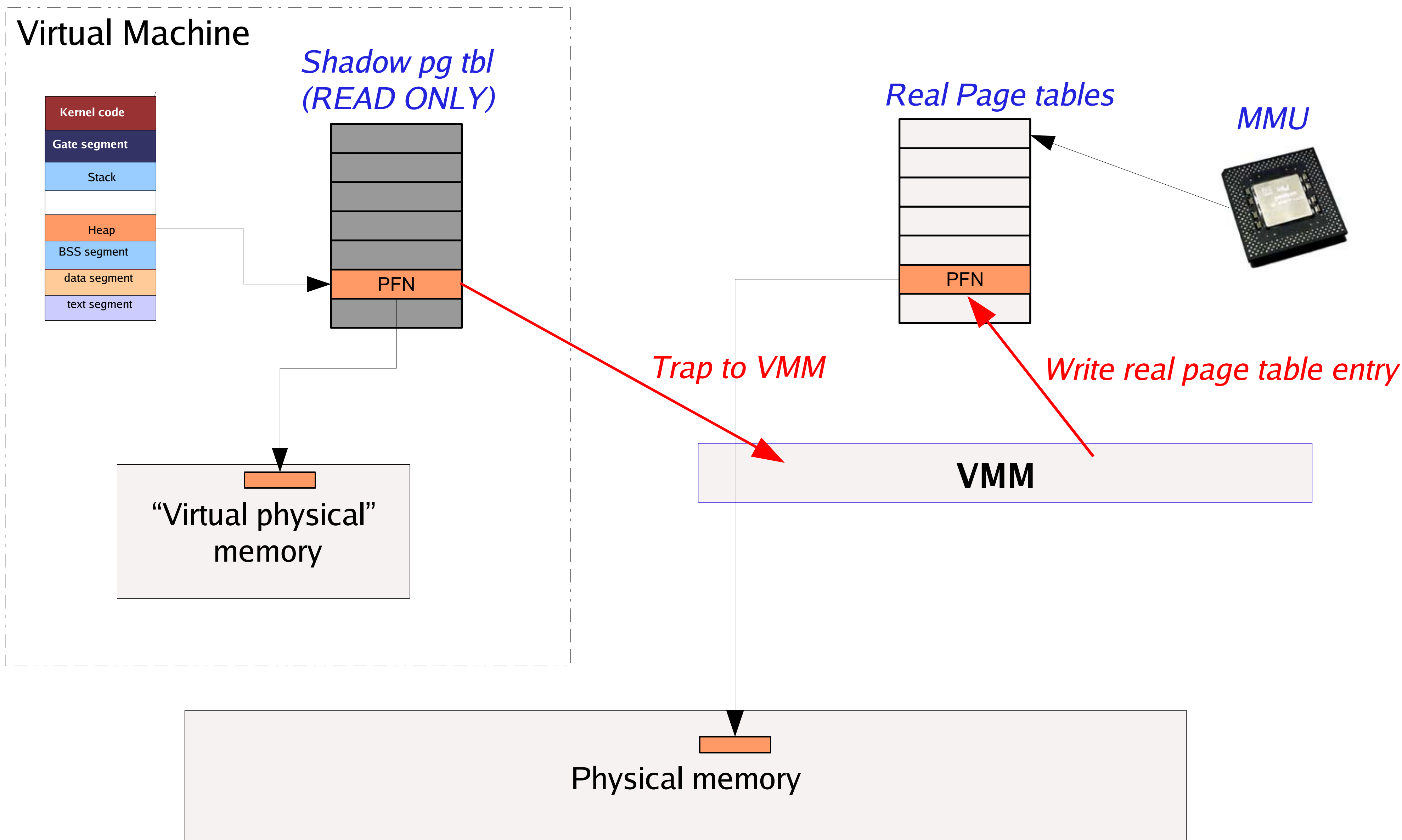


© 2006 Matt Welsh – Harvard University

Shadow Page Tables

Idea: VM modifies "shadow" page tables (not the real ones)

- VMM has to trap access to them and then update the real page tables accordingly.

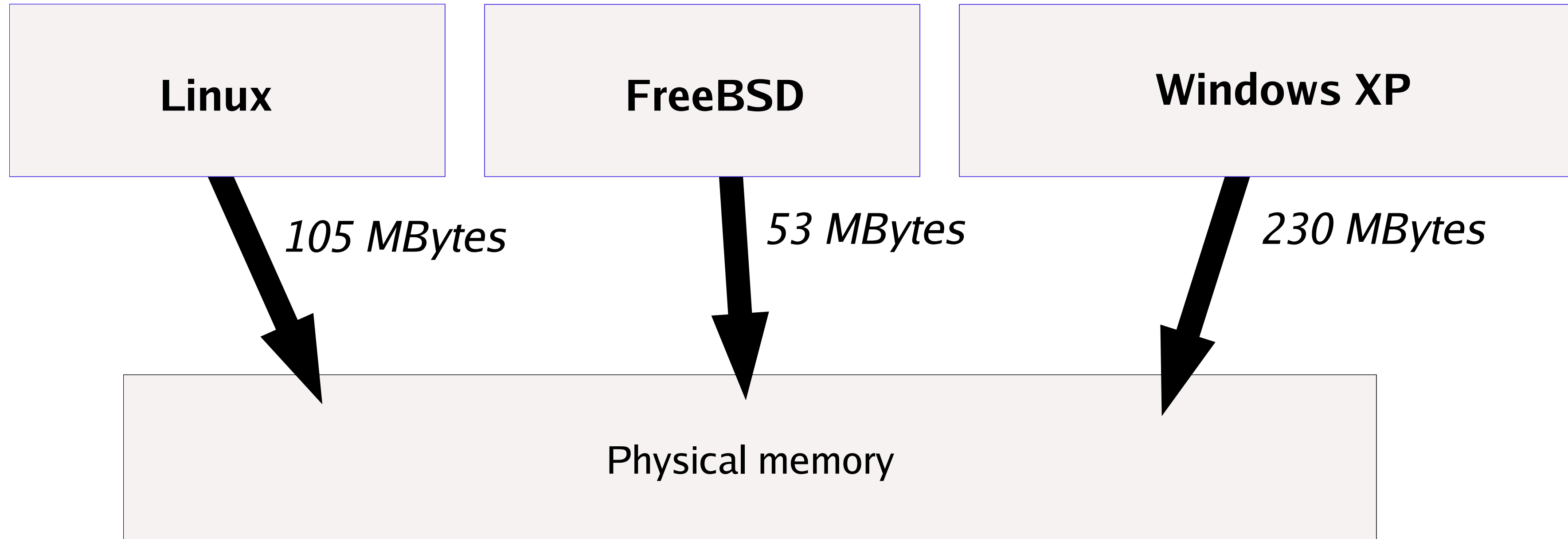


© 2006 Matt Welsh – Harvard University

Physical Memory Management

How do we allocate memory across Virtual Machines?

- Just like with processes, want to ensure every VM has enough physical RAM.

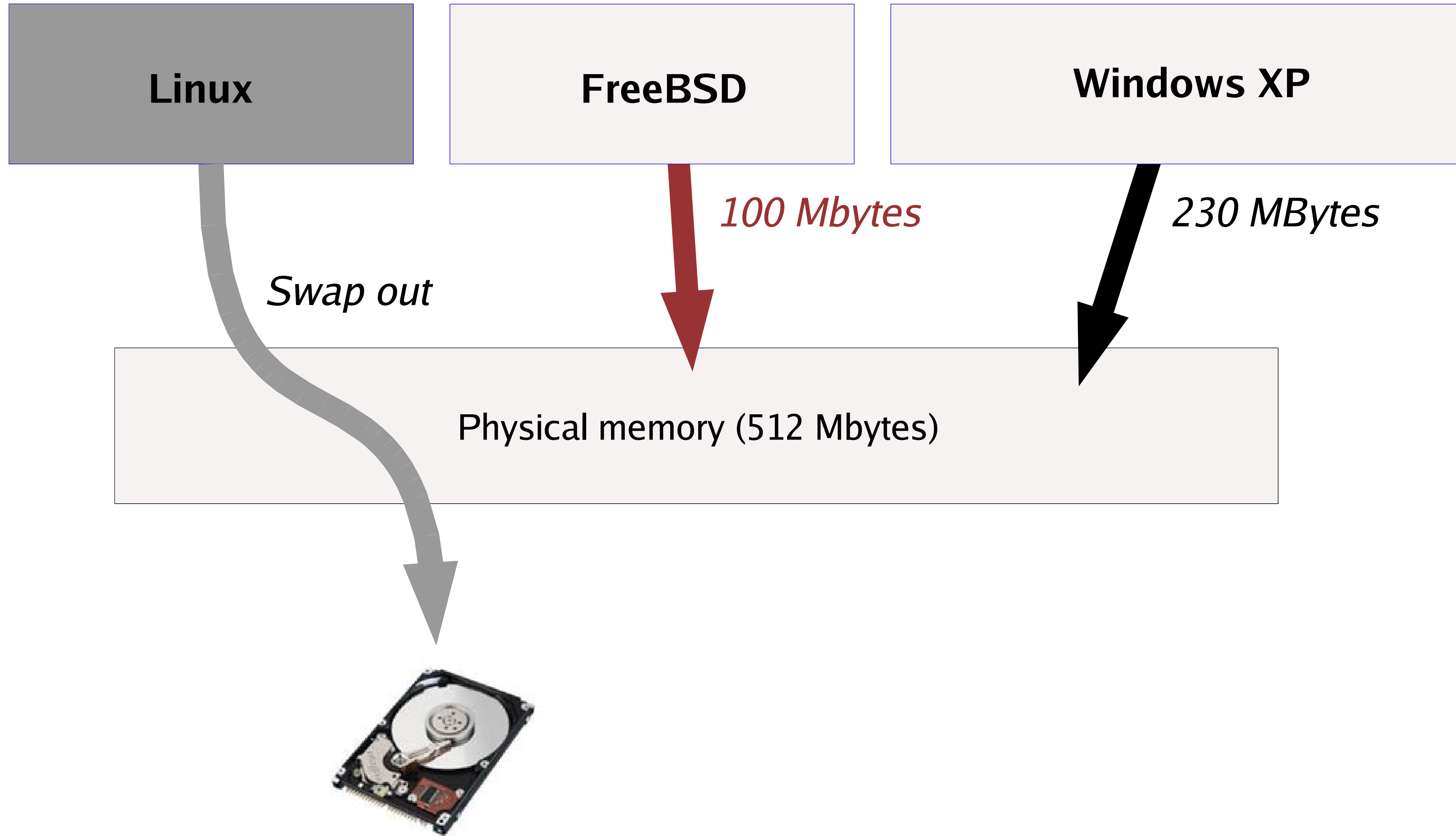


How does the VMM reclaim memory from a Guest OS?

© 2006 Matt Welsh – Harvard University

Reclaiming Memory

One way to manage memory: Swap entire VMs.

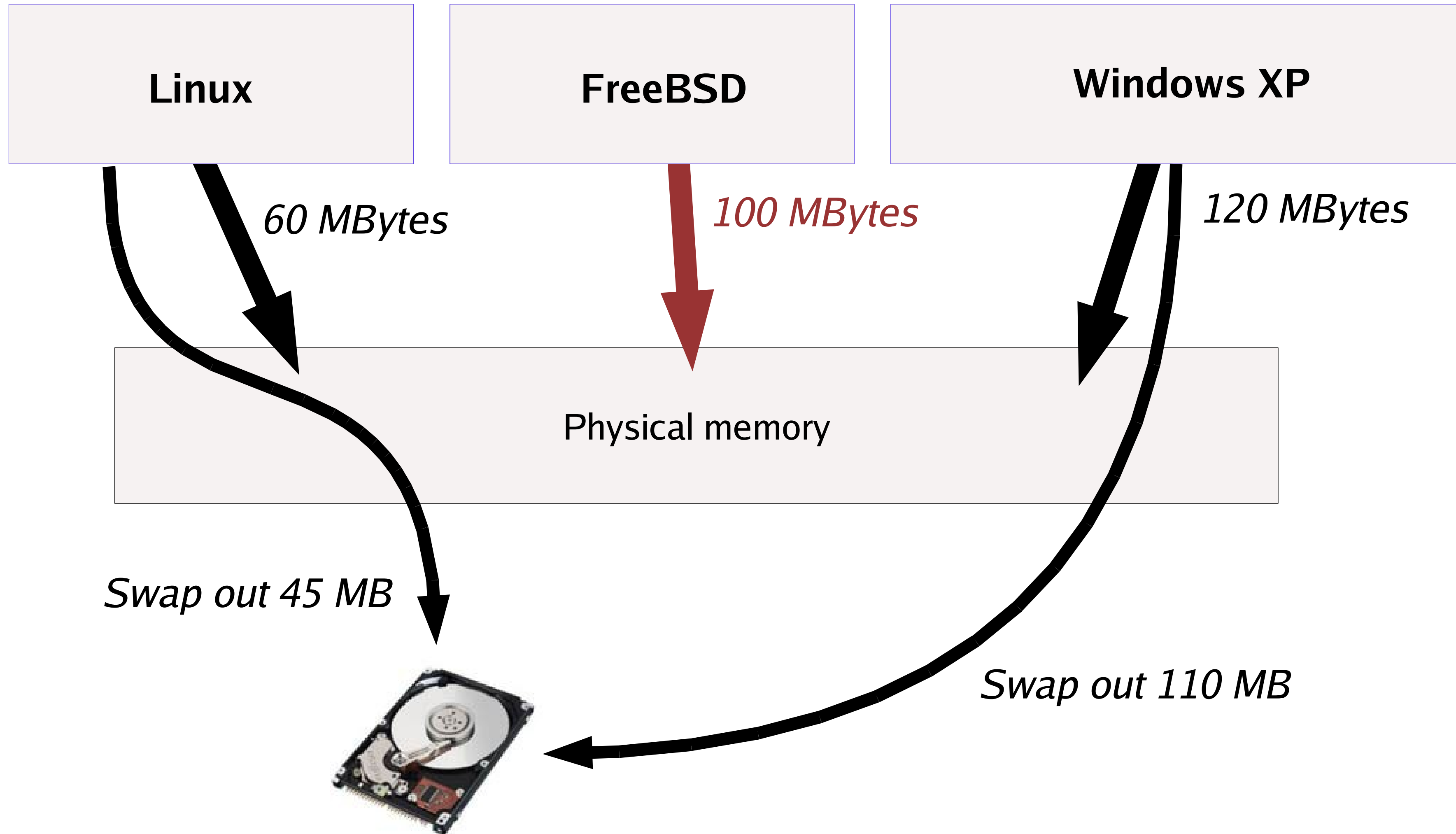


© 2006 Matt Welsh – Harvard University

Reclaiming Memory

Another way: "Meta-paging"

- VMM swaps out individual pages to disk, just like the OS does.



© 2006 Matt Welsh – Harvard University

Problems with memory reclamation

What's wrong with the VMM “stealing” memory from a guest OS?

© 2006 Matt Welsh – Harvard University

Problems with memory reclamation

What's wrong with the VMM “stealing” memory from a guest OS?

VMM has no idea what those pages are being used for!

- Could be something really important, i.e., the Guest OS's main memory map
- It's just generally a bad idea for the VMM to make uninformed decisions.

Can lead to *double paging*:

- VMM picks a page from the Guest OS and swaps it out.
  - *(Of course, the Guest OS has no idea ... it just thinks it's a physical page.)*
- Guest OS is under memory pressure, so it picks the same page to swap out.
- Voila: Page has been swapped to disk *twice!*
  - *And you gotta swap it back in before you swap it out the second time - DOH!*

Solution?

- Get the Guest OS to pick which pages to swap out itself!
- So how do we induce the Guest OS to start swapping??

© 2006 Matt Welsh – Harvard University

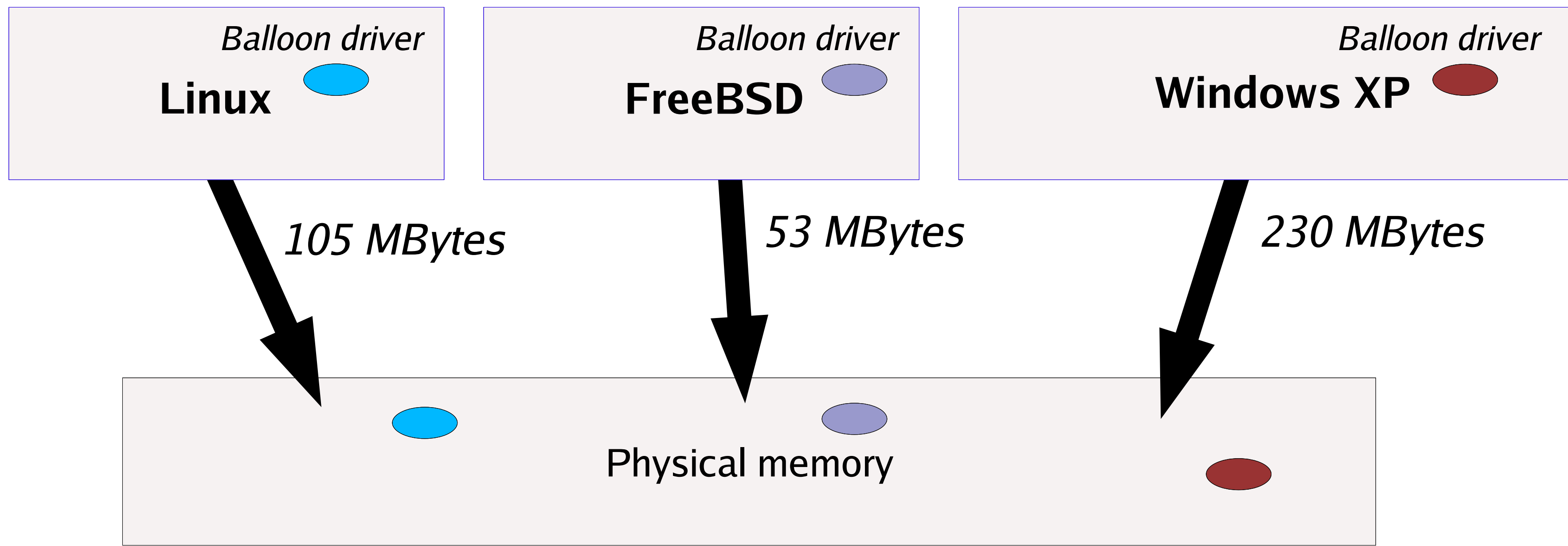
Ballooning

Idea: Add a little driver to the Guest OS.

- Driver can allocate (pinned) physical pages by inflating its “balloon”

When VMM wants the Guest OS to give up memory, asks driver to inflate the balloon

- Causes Guest OS to feel physical memory pressure.
- Causes its own paging mechanisms to kick in!

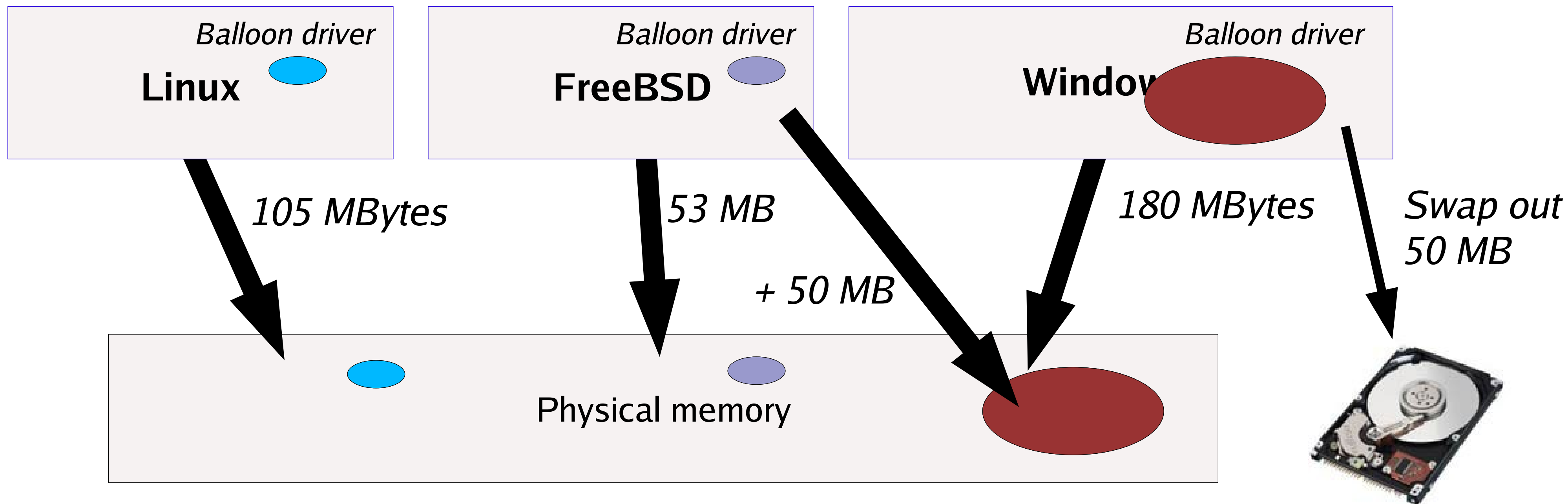


© 2006 Matt Welsh – Harvard University

Ballooning

Idea: Add a little driver to the Guest OS.

- Driver can allocate (pinned) physical pages by inflating its "balloon"
- Causes Guest OS to feel physical memory pressure.
- Causes the Guest OS's paging mechanisms to kick in!
  - *Critical: The balloon stays inflated ... but the VMM recycles the memory for another Guest OS!*

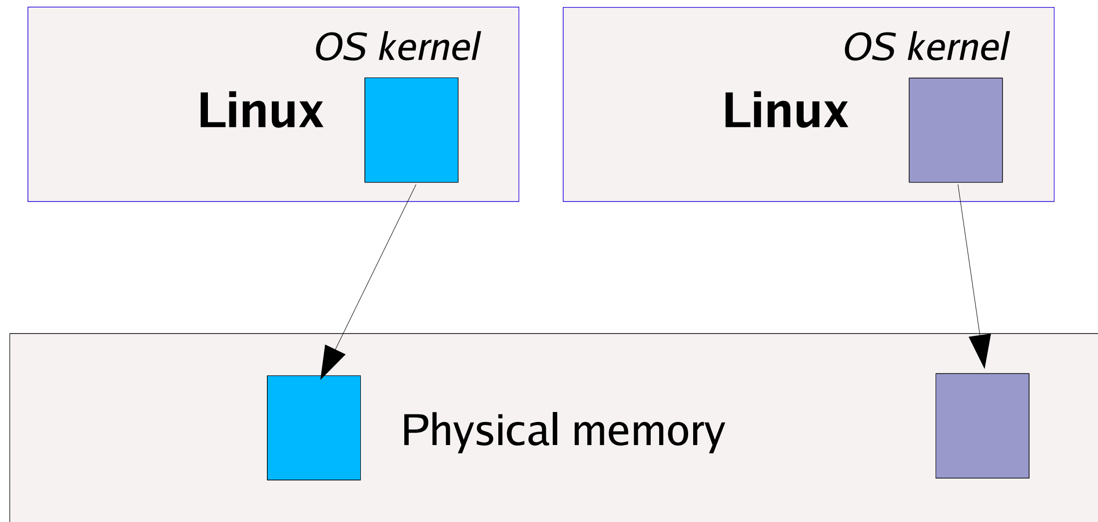


© 2006 Matt Welsh – Harvard University

Page Sharing

What if you are running many copies of the same OS?

- Lots of pages with the same stuff in memory.
- Not unlike the problem of many copies of the same process...



How can we find these redundant pages?

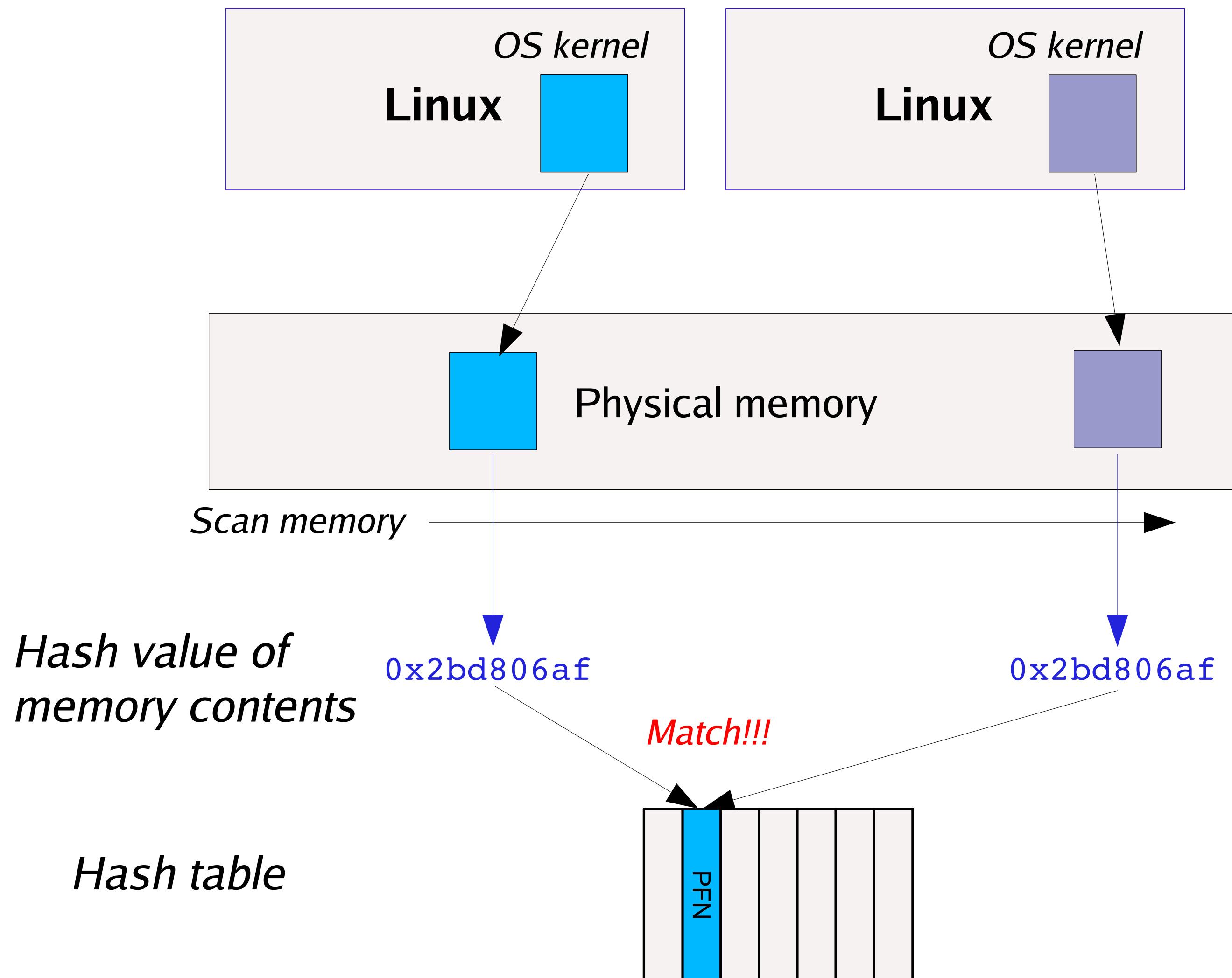
- Unlike forking a process or running two copies of the same binary, VMM has no *a priori* notion that two pages are "the same"

© 2006 Matt Welsh – Harvard University

Content-based Page Sharing

Idea: Scan physical memory and look for identical pages

- Keep only one copy of the identical page
- Use copy-on-write when it's modified

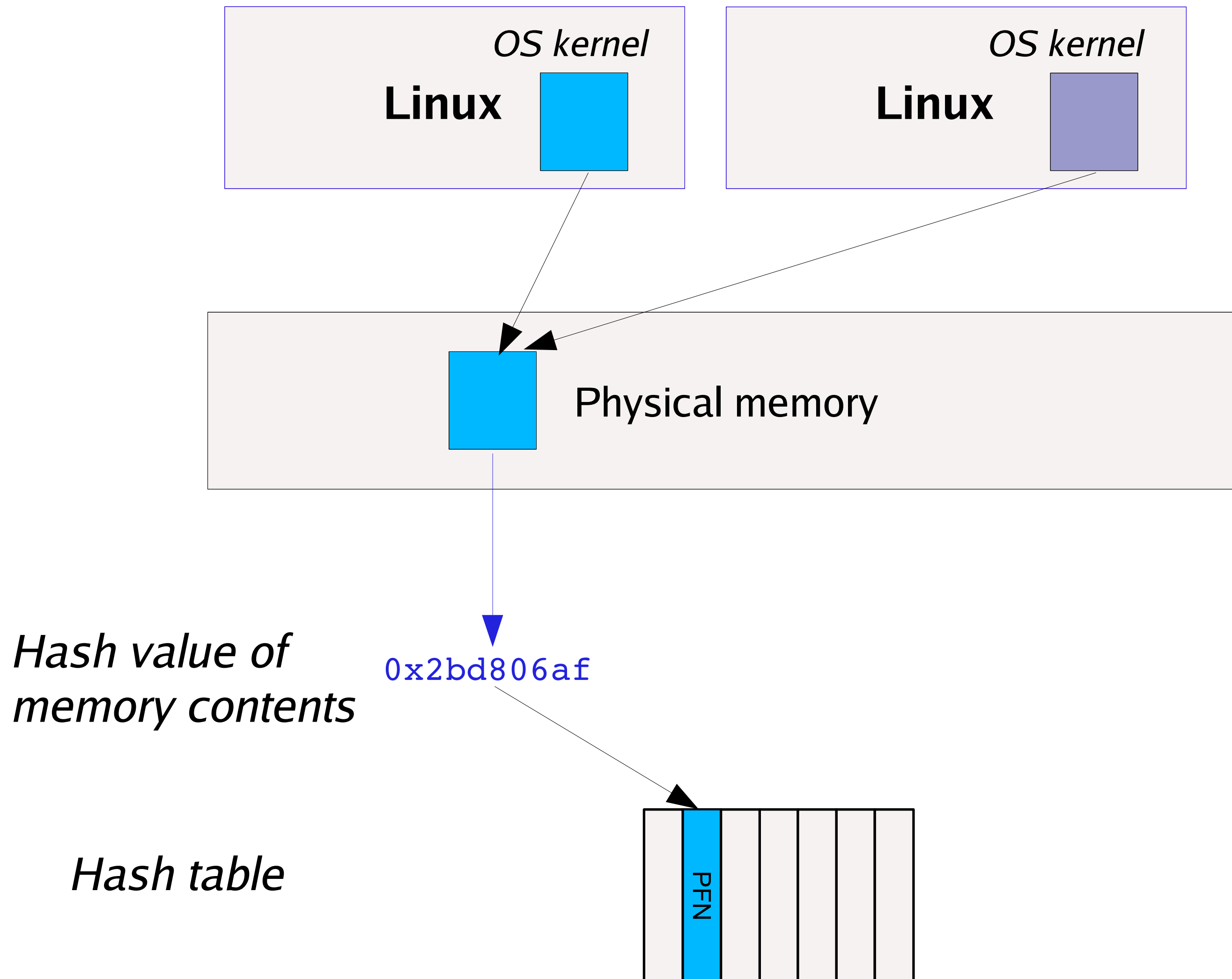


© 2006 Matt Welsh – Harvard University

Content-based Page Sharing

Idea: Scan physical memory and look for identical pages

- Keep only one copy of the identical page
- Use copy-on-write when it's modified



© 2006 Matt Welsh – Harvard University

I/O Device Interfaces

All access to I/O devices must go through VMM

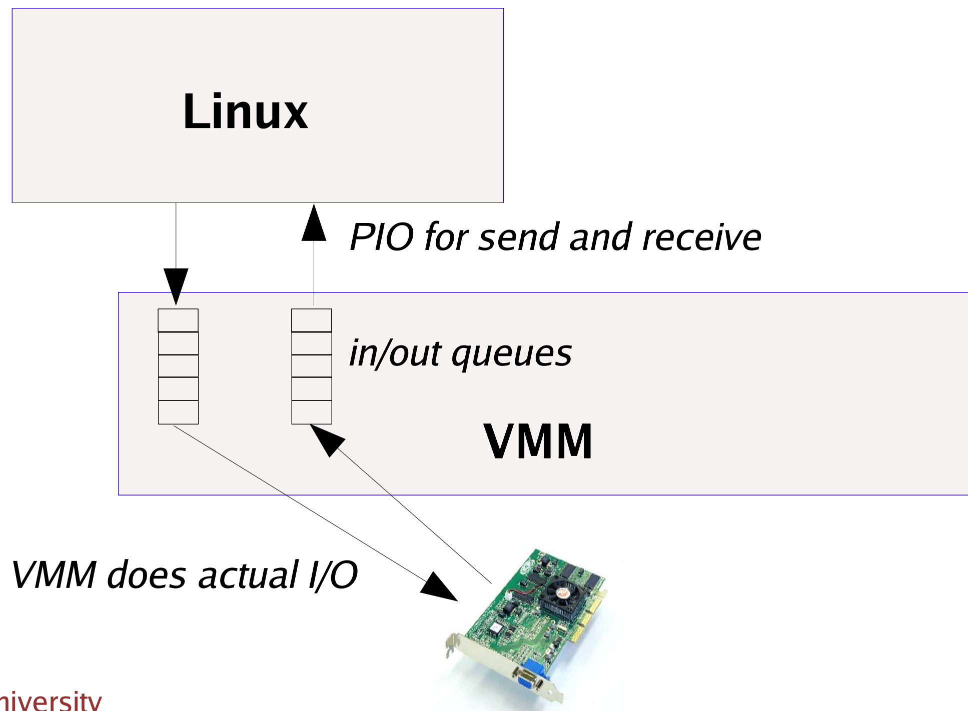
- This can be very expensive!

Many real I/O devices have a very “chatty” interface

- Many programmed I/O operations to do simple things
- To read a disk block or send an Ethernet frame, need a lot of interactions with VMM

Observation: The VMM can provide much simpler I/O interfaces

- Example: One PIO to transmit a packet, another to receive a packet



© 2006 Matt Welsh – Harvard University

Virtualizing the x86

Our discussion so far assumes that the CPU is “virtualizable”

- For example, that any privileged insn executed in user mode will trap to the OS
- Also that regular and privileged instructions operate identically (except for protection)

Problem: The Intel x86 is not virtualizable!

Example: POPF instruction

- Restores the CPU state from the stack, into the EFLAGS register
- Some bits in EFLAGS can only be modified when in supervisor mode
  - *e.g., Interrupt enabled bit*
- When executed in user mode, POPF will **not** modify the interrupt enabled bit!
  - *In other words, different behavior when running as user or supervisor.*

Bottom line: Guest OS will not see the correct behavior of the POPF instruction when running in user mode!

© 2006 Matt Welsh – Harvard University

VMWare Approach

VMWare uses *binary translation* to get around this problem

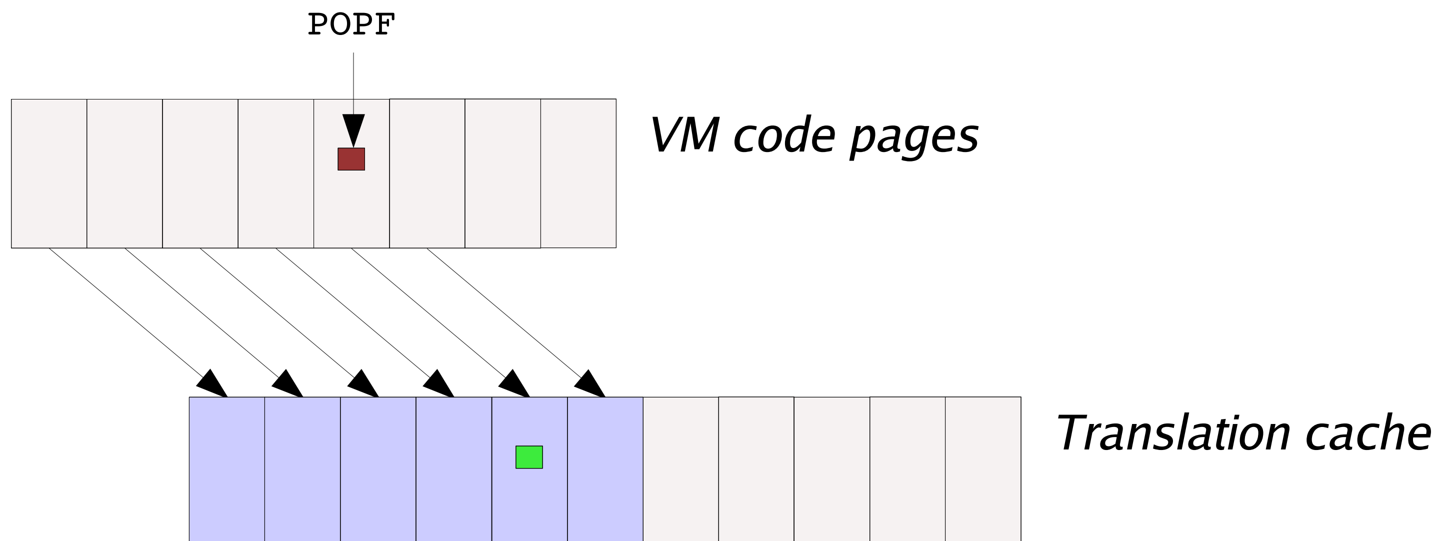
- Code running in each VM is scanned on-the-fly for “non-virtualizable” instructions
- Code is rewritten into a safe form

Can also directly inline VMM functionality into the VM code

- Eg., Fast versions of I/O processing code

VMM maintains a translation cache of recently-rewritten code pages

- Avoids high overhead for rescanning and rewriting on each execution pass
- Must trap writes to any code page to invalidate the cache
  - *(OS and apps can still execute code from writeable pages ... but not too common)*



© 2006 Matt Welsh – Harvard University

Paravirtualization

Binary translation is obviously fairly complex

- Not to mention the performance overhead.

Another approach: Paravirtualization

- Idea: Define a *subset* of the x86 instruction set that is virtualizable
- *Port* the Guest OS to this new instruction set architecture

Technique employed in two research VMMs: Xen and Denali

Paravirtualization is very simple and very fast

- However, requires a (non-trivial) porting effort
- Must ensure Guest OS doesn't depend on any “non-virtualizable” features
- Can also simplify other aspects of the VMM/OS interface, e.g., page table mgmt
- Xen project has ported Linux and NetBSD ... did a preliminary port of Windows XP

© 2006 Matt Welsh – Harvard University

For More Information

"The Origin of the VM/370 Time-Sharing System,"
- R. J. Creasy, IBM J. Res. Develop, 25:5, Sep. '81

"Memory Resource Management in the VMWare ESX Server,"
- Carl Waldspurger, OSDI'02

"Scale and Performance in the Denali Isolation Kernel,"
- A. Whitaker, M. Shaw, and S. Gribble, OSDI'02

"Xen and the Art of Virtualization,"
- P. Barham *et al.*, SOSP'03

"Virtualization system including a virtual machine monitor for a computer with a segmented architecture,"
- S. Devine *et al.*, US Patent #6,397,242

© 2006 Matt Welsh – Harvard University