HARVARD
School of Engineering
and Applied Sciences

Virtualization & Abstractions for Concurrency

CS61, Lecture 24

Daniel Margo
Prof. Stephen Chong

November 29, 2011

Announcements

- Final Thursday Dec 1
- Assignment 6 (Shell) due Tuesday Dec 6, 11:59pm

HARVARD
School of Engineering
and Applied Sciences

Virtualization

CS61, Lecture 24

Daniel Margo

November 29, 2011

Virtualization

with Daniel Margo

Outline

- History: Why was virtualization developed?
- Purpose: What do we use it for today?
- How Virtualization Works
- Technical Challenges

A Brief Philosophical Statement

- **Resource sharing is a fundamental Systems problem.**

- To solve it, we develop abstractions:
  - File Systems (abstract storage)
  - Processes (abstract CPU)
  - Virtual Memory (abstract memory)

- In *ye olden times*, scarce computers were shared among many users. Many classic Systems abstractions were developed to solve this problem.

VM/370

- In the 1960's, IBM developed a family of machines (System/360, System/370) and an operating system called VM/370.
  - Users share a single machine by giving each his/her own "machine."
  - Real hardware partitioned by a transparent **virtual machine monitor** into per-machine virtual hardware.
  - Compared to operating systems, it's just a different sharing abstraction:

Conventional OS	**VM Monitor**
Schedules processor among processes	Shares processor across machines
Provides a global file namespace	Provides a "raw disk" to each machine
Virtual address space for each process	Partitions memory among machines

The VM Abstraction

- OS provide a programmatic abstraction (system calls, libraries, etc.) This makes it easier to write programs.
- Virtual machine monitors provide abstract hardware.

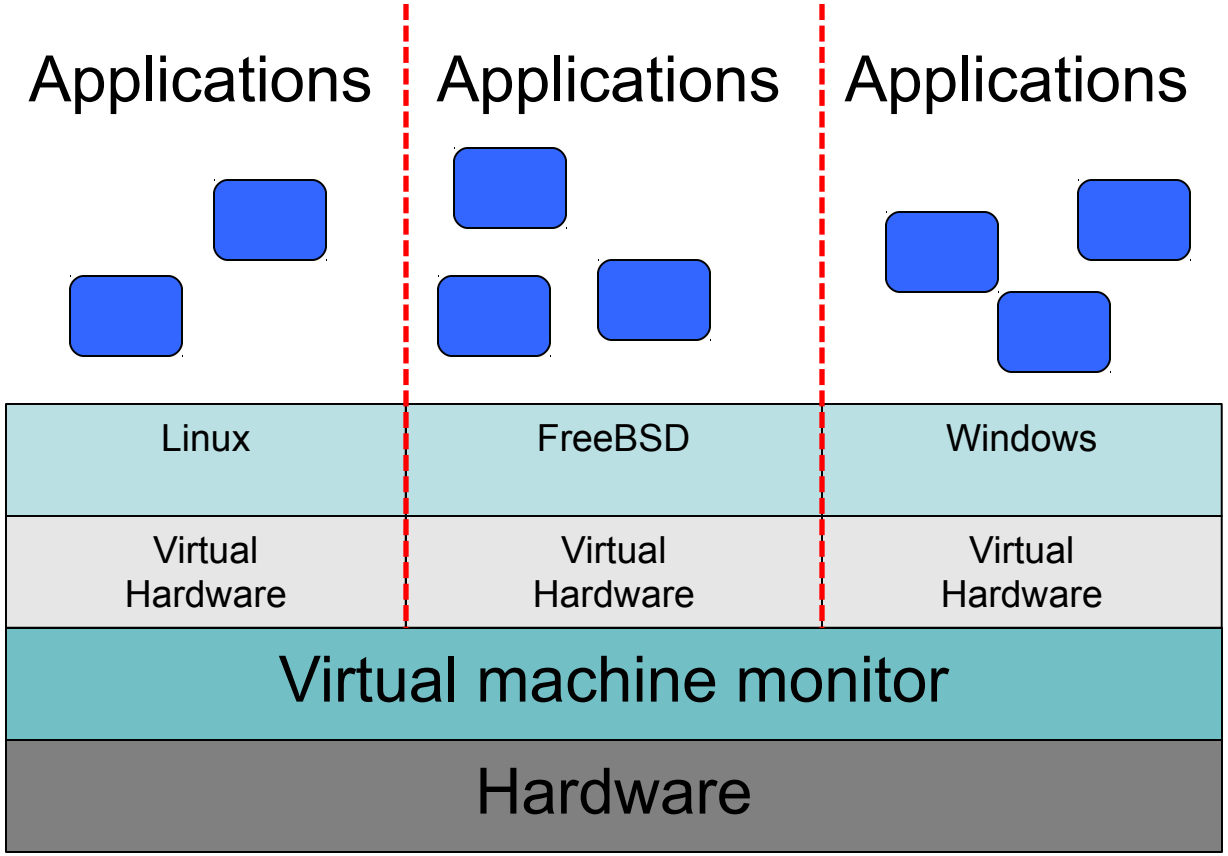
Think about this for a minute...

- Hardware is hard to program! In practice, we still want something like an OS to run our programs.

- Virtual machines largely died out in the 80’s...but made a major resurgence in the 90's and continue to do so.

WHY?

Virtual Machine Architecture



What is the VM Monitor?

- A layer of software, running on the real hardware, that multiplexes the hardware into virtual instances.
- An OS multiplexes among processes; a VMM multiplexes among virtual machines.
- From the point of view of an OS or application running on a VM, anything running on a different VM looks the same as if it were running on a different computer.
- This is a different relationship than that between processes running on the same OS!

Why are we doing this?

- Original VM/370: To provide multi-user sharing.
- Motivations when re-introduced:
  - Allow geeks like us to run UNIX/Linux and still play Starcraft.
  - Test and debug new installations (both OS and servers.)
  - Provide effective fault containment (on multiprocessors, etc.)
- Today's motivations:
  - **Software Delivery**: can package a working, optimized operating environment together with applications.
  - **Security**: can run dangerous/protected things in isolation.
  - **Resource Sharing**: Makes it easier for infrastructure providers (e.g. data centers) to efficiently use and provision

THAT CLOUD SLIDE

- So when you rent a server IN THE CLOUD, what are you really paying for?
  - **Computing resources**, packaged in the abstraction of “your own server.”
  - Comes with the software you need to run it.
  - Secure isolation from other “servers.”
  - Can be re-provisioned with extra CPU power, memory, etc. on demand (if you can pay for it.)

- Physically, there's just a data center somewhere that its owner is trying to rent out efficiently. Virtualization is the multi-user abstraction of data center computing.

VMM Goals

- The VMM must give each virtual machine the *complete* illusion of a real machine, including:
  - I/O devices,
  - CPU interrupts,
  - CPU protection levels,
  - Virtual memory hardware,
  - etc...
- Must be 100% compatible with existing software
  - Run unmodified operating systems and applications.
- Must completely isolate VMs from each other
  - Can't allow one VM to stomp on another's memory or CPU state.
- Must have high performance
  - Or else nobody will want to use it.

Why is this hard?

- Guest OS needs to call privileged CPU instructions
  - To perform I/O, manipulate hardware, etc.
  - Can we let it do this directly?
- Guest OS needs to manipulate page tables
  - To set up virtual->physical mappings for its applications.
  - Why is this potentially dangerous?
- Guest OS needs to believe it's on a real machine
  - Must support **full** instruction set of processor...
  - ...including all the weird virtual memory features.
  - And (at least some) “real” I/O devices (disk, network, etc.)

Privileged Mode

- Typically an OS runs in a privileged CPU mode...
- But if we really want to prevent VMs from executing privileged instructions, then we cannot allow a VM to run in privileged mode!
  - Running the VM in unprivileged mode is the only way to ensure that a guest OS can't do dangerous stuff.
  - How do we get the guest OS to behave like an OS then?
- Option 1: 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.
  - This works, but …
    - It’s SLOW!
    - VMM is a full CPU emulator.

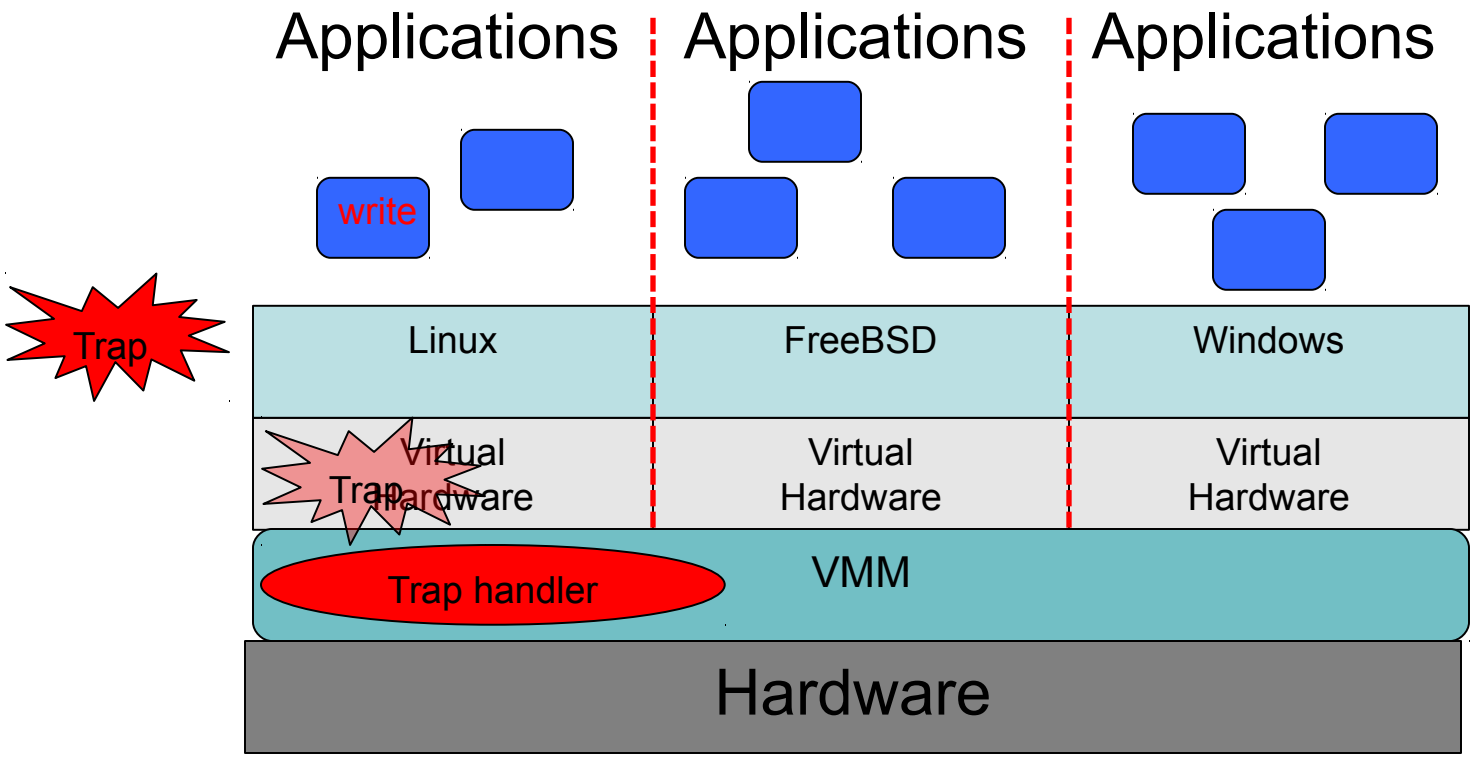
Option 2: Direct Execution

- Let the guest OS run directly on the machine, but in unprivileged mode. Much faster.
- What happens when a guest OS executes a privileged instruction?

Option 2: Direct Execution

- Let the guest OS run directly on the machine, but in unprivileged mode. Much faster.
- What happens when a guest OS executes a privileged instruction?
  - Interrupt!
  - Who catches this interrupt?
    - The software running in privileged mode, which is the VMM!
  - The VMM examines the privileged instruction and decides what to do:
    - Handle the fault (issue the privileged instruction for the guest.)
    - Disallow the operation (and kill the VM?)

Example


Applications
Applications
Applications
write
Trap
Linux
FreeBSD
Windows
Virtual Hardware
Virtual Hardware
Virtual Hardware
Trap
Trap handler
VMM
Hardware

Protection Levels

- The OS relies on its privilege to protect its memory from users. How can we protect it without privilege?
- Turns out, the x86 architecture supports multiple protection levels (four to be exact).
- VMMs take advantage of them:
  - Run user code in Ring 3
  - Run the guest OS in Ring 1
  - Run the VMM in Ring 0

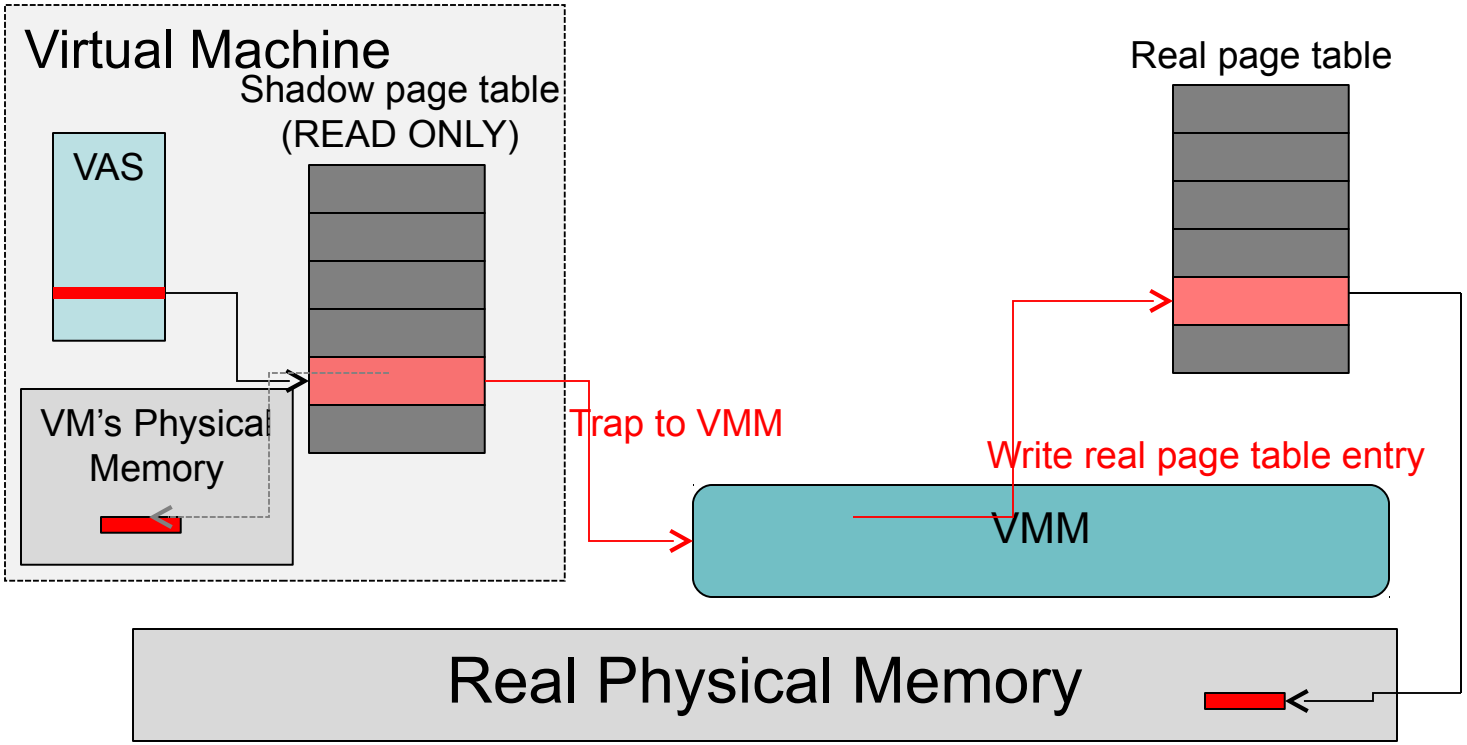
Memory Management

- The guest OS thinks that it has its own page tables (because managing virtual memory is one thing OSs do, and when it does its job it needs to be able to translate virtual addresses correctly).

- But … we can't allow the guest OS to actually manipulate hardware PTEs.
  - If we let it directly manipulate PTEs, it could map *any* page.
  - Including pages from other VMs!
  - And that would *destroy the world.*

Shadow Page Tables

- Give each virtual machine make-believe software page tables (called *shadow page tables*).
  - The guest OS will fault to the VMM when it manipulates page tables, allowing the VMM to update the real page tables appropriately.
  - So rather than virtual → physical, we now have virtual → shadow → physical memory!



Physical Memory Management

- How does the VMM allocate memory across VMs?
  - Each VM/OS needs enough physical memory.
  - How do you reclaim space from a VM?

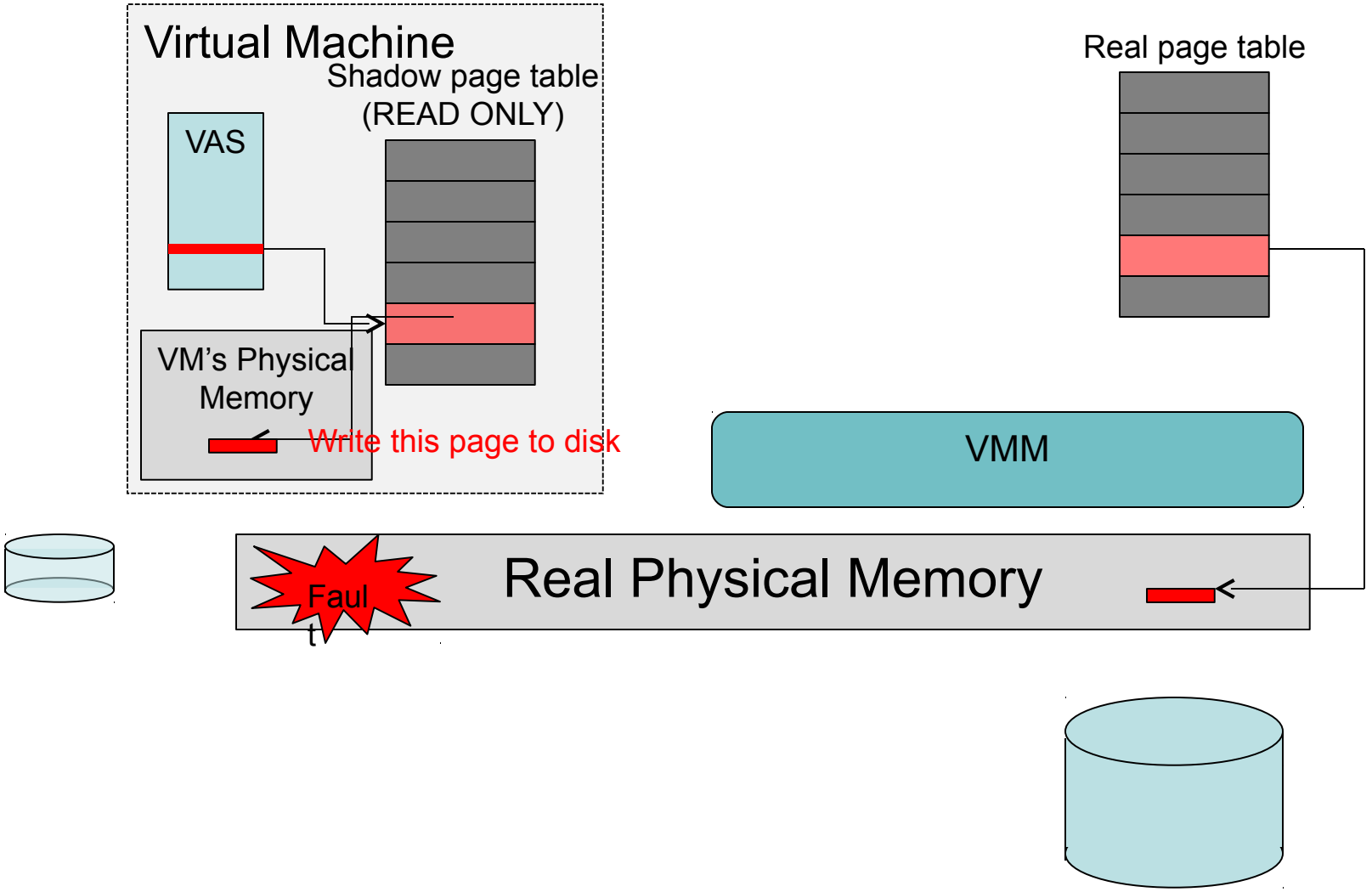
- Option 1: Swap the entire VM
  (just like the OS swaps a process)
- Option 2: VMM-managed paging
  (just like the OS pages processes).

- Is the problem any different? Can't we just use the same techniques in the VMM that we use in the OS?

VMM-Managed Paging

- VMM has no idea how pages are being used.
  - In particular, can't distinguish between OS and app pages.
  - If the VMM is solely responsible for paging, bad things can happen.
- Double Paging:
  - VMM picks a page from the guest OS and evicts it
    - This just means that a page that appears to be in the virtual machine's physical memory is really on disk.
  - Now imagine that the guest OS is under memory pressure and it decides to evict a page from its "physical" memory.
    - It could choose a page that the VMM has already evicted...
    - ...and the VMM will interrupt and page it back in...
    - ...so that the guest OS can write it out again a second time!
    - This is very silly!

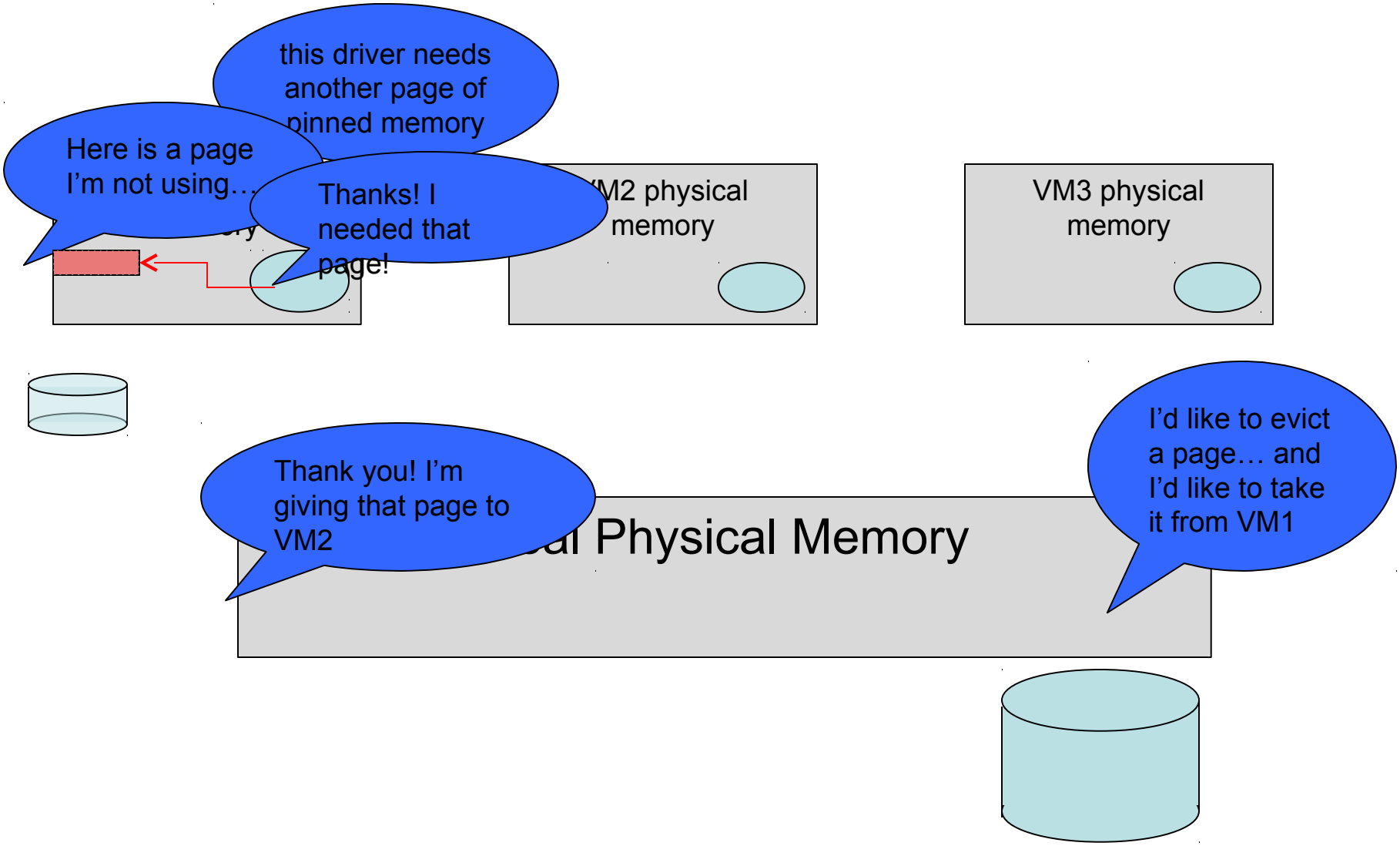
Double Paging Example



Avoiding Double Paging: Ballooning

- What the VMM really wants is to trick the guest OS into evicting pages so that the VMM doesn't have to.
- This suggests that we need the guest OS to feel pressured for memory.
- How do we induce memory pressure on the guest?
- Add a “balloon” hardware driver to each guest OS
  - The driver allocates (pinned) physical pages.
  - When the VMM “inflates the balloon” it pins more pages.
  - These pages aren’t actually needed; all they do is consume part of the virtual machine’s address space; thus they are perfect candidates for the VMM to take away from the guest!

Ballooning in action



Optimizing Memory Usage

- In many cases, you may run many copies of the same guest OS.
  - These copies use many of the same pages.
  - Just like an OS would like to avoid multiple copies of the same text page, the VMM would like to avoid having multiple copies of the same (OS) text page.
- How can you identify redundant pages?
  - Borrow a page from the file system folks who do deduplication.
  - Maintain a table of the hash values of all the contents of all the pages in memory.
  - When you are about to allocate a new chunk of real memory for a page, lookup its hash.
  - If the hash is already in the table, compare contents to be sure.
  - If they really are the same, reuse the physical page marking it copy-on-write.
- This technique is called “content-based page sharing.”

I/O Device Interfaces

- All access to I/O devices must go through VMM
  - This can add a lot overhead!
- Many real I/O devices have complex interfaces.
  - It takes many I/O operations to do simple things.
  - E.g., reading a disk block or sending an Ethernet frame require a lot of interaction with the VMM.
- Observation: The VMM provides “virtual” devices
  - Those devices can be dirt simple!
  - Example:
    - Simple network driver with two I/O interfaces.
    - One transmits a packet, another receives a packet.
- Lesson: Design simple hardware!

Hardware Realities

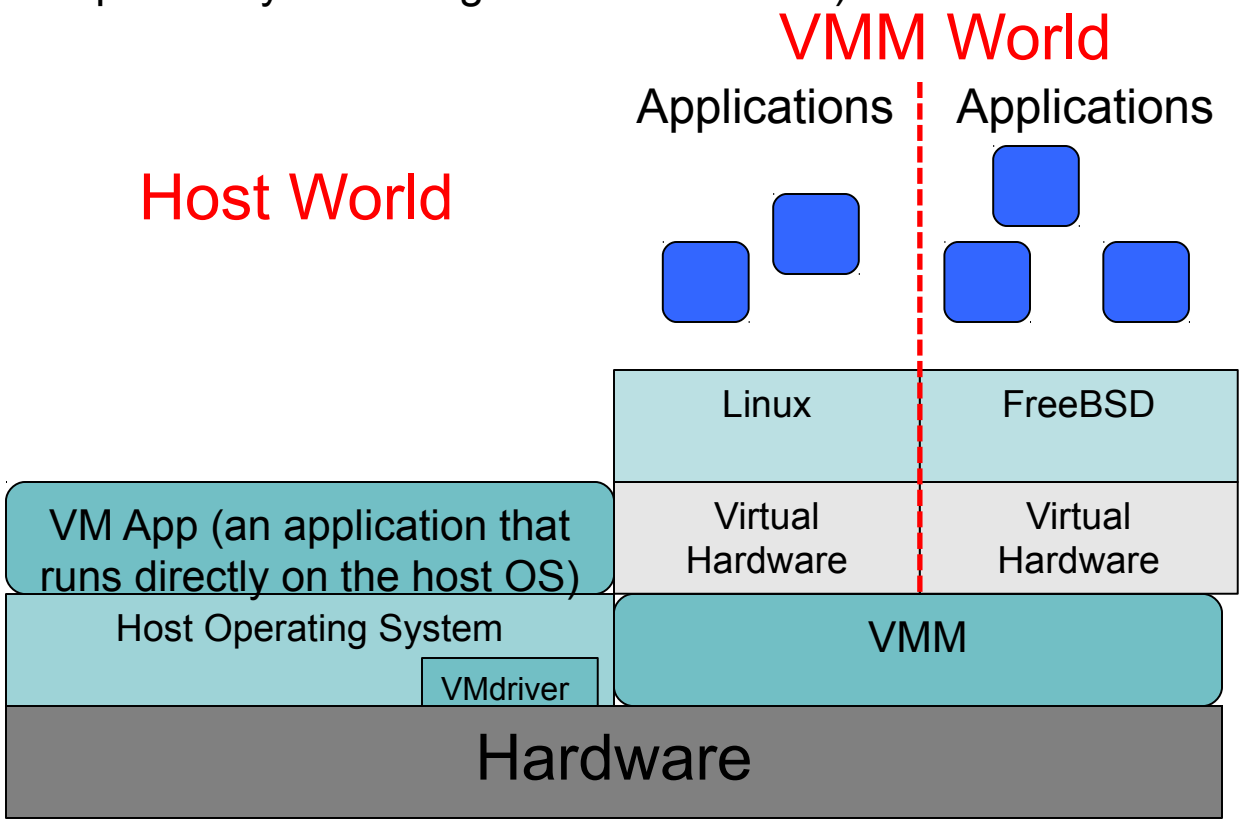
- So far, we've pretended that it's actually possible to virtualize the CPU, but some CPU architectures are not actually fully virtualizable:
  - Every privileged instruction must trap to the OS when executed in user mode.
  - Except for protection levels, instructions in privileged and unprivileged mode must operate identically.
- Guess what: the x86 architecture is ***not*** virtualizable!
- Example: The POPF instruction
  - Restores the CPU state from the stack, into the EFLAGS register.
  - Some bits in the EFLAGS be modified only in privileged mode.
    - E.g., Interrupt enabled bit
  - When execute in user mode, POPF will ***not*** modify the interrupt enabled bit.
    - This leads to different behavior in different modes.
- This means a guest OS is not going to observe the behavior it expects if we execute it in user mode.

VMware Approach

- Binary translation
  - Code running in each VM is scanned on-the-fly for “non-virtualizable” instructions.
  - Code is dynamically rewritten into a *safe* form.
- This allows you to play some games.
  - When you translate code in the guest OS, you can incorporate some of the functionality from the virtual machine monitor.
  - Example: Fast, safe versions of I/O processing code
- Binary translation can be slow, so much of the secret is in making it fast.
  - Once you’ve translated a page once, you probably don’t want to do it again…
  - Maintain a *translation cache* of recently-rewritten code pages.
  - If you observe the same page again, use its already translated version.
  - Must trap writes to any code page to invalidate the cache
    - This only matters if you allow writable code pages
    - But even if you do, you can get this to work.
- This should sound a lot like a dynamic compiler! That's because it kinda is.

VMware Workstation

- Virtual machine monitor runs as an application
  (not a special layer running on the hardware!)



Paravirtualization

- Binary translation is fairly complex and slow.
- Paravirtualization is an alternative:
  - Define a virtualizable subset of the x86 instruction set.
  - Port the guest OS to the new instruction set architecture.
  - Violates the “no modifying the guest OS” rule.
- Xen uses this approach.
- Advantages
  - Simple
  - Fast
  - Allows you to simplify OS/VMM interaction.
- Disadvantages
  - Requires that you port the guest OS.
  - Requires that guest doesn't depend on any non-virtualizable features.

References

- The Origin of the VM/370 Time-Sharing System
  - R. J. Creasy, IBM J. Res. Develop, 25:5, Sep. '81
- The double paging anomaly
  - R. P. Goldberg, R. Hassinger, AFIPS 1974
- DISCO: Running Commodity Operating Systems on Scalable Multiprocessors
  - E. Bugnion, S. Devince, M. Rosenblum, SOSP 1997
- Virtualizing I/O Devices on VMware Workstation's hosted Virtual Machine Monitor
  - J. Sugerman, G. Venkitachalam, B. Lim, USENIX ATC, 2001
- Memory Resource Management in the VMWare ESX Server
  - C. Waldspurger, OSDI'02
- Xen and the Art of Virtualization
  - P. Barham *et al.*, SOSP'03

HARVARD
School of Engineering
and Applied Sciences

Abstractions for Concurrency

CS61, Lecture 24

Prof. Stephen Chong

November 29, 2011

Synchronization is difficult

- Our systems have increasing parallelism
  - Multi-core, distributed systems (cloud computing, web apps, …)
- We need to be able to write concurrent programs
  - To take advantage of parallelism
  - To correctly implement complex systems
- But concurrent programming is difficult!

Why is concurrent programming hard?

- One potential reason: mismatch between concepts in programmer's head, and concepts expressed in code

Locks are bad

- Standard problems:
- Taking too few locks
  - Not holding the right lock when accessing a resource
- Taking too many locks
  - Reduce concurrency, or cause deadlock
- Taking wrong locks
  - Connection between resource and lock protecting it exists only in mind of programmer
- Taking locks in wrong order
- Error recovery
  - When an error happens, need to make sure that state is consistent, and locks are not held forever
- Lost wakeups erroneous retries
  - Easy to forget to signal a CV, or to re-test condition upon waking

Example

```
#include <stdio.h>

void transfer(account *acct_from,
              account *acct_to,
              int amt) {

    acct_from.balance -= amt;



    acct_to.balance += amt;

}
```

Potential data race!
Unsynchronized accesses
to shared resource

Example

```
#include <stdio.h>

void transfer(account *acct_from,
              account *acct_to,
              int amt) {
    acquire(global_lock);
    acct_from.balance -= amt;
    release(global_lock);

    acquire(global_lock);
    acct_to.balance += amt;
    release(global_lock);
}
```

No data races!
But is it correct yet?

Example

```
#include <stdio.h>

void transfer(account *acct_from,
              account *acct_to,
              int amt) {
    acquire(global_lock);
    acct_from.balance -= amt;



    acct_to.balance += amt;
    release(global_lock);
}
```

Programming abstractions

- Idea: expose a higher-level abstraction to programmers to let them express synchronization
- Use **atomic** construct to indicate code that should run as if it were the only code running on the system

```
void transfer(account *acct_from,
              account *acct_to,
              int amt) {
    atomic {
        acct_from.balance -= amt;
        acct_to.balance += amt;
    }
}
```

`atomic` construct

- High-level construct for coordinating mutation of shared data
- `atomic { S }` executes `S` atomically and in isolation
  - Isolation: from `S`'s perspective, no other threads are interleaved with it
  - Atomically: from other threads' perspective, either `S` has not yet executed or has finished execution; never see intermediate state
- Makes it easy to reason about invariants

How to implement `atomic`?

- Several possibilities:
- Use locks
  - Have the compiler figure out which locks to acquire for an atomic section
  - One global lock
    - easy to figure out, limits concurrency
  - One lock per variable
    - overhead of lock management
    - for dynamic structures, must approximate, or have some way of associating locks with dynamically malloced locations
- Transactional memory

Transactional memory

- Support for atomic access to memory
- Can be implemented in
  - Hardware
    - Needs modifications to processor, cache, and bus protocols
  - Software
    - Programming language support and/or runtime libraries, and some hardware support (e.g., atomic compare and swap)

Sketch of an STM implementation

- Software Transactional Memory (STM)
- On entry to an atomic section, start a log of memory accesses
- When we write a value, record the new value in the log
  - (Don't write the value to memory yet)
- When we read a value, check the log to see if we've written it within this transaction
  - Otherwise, read the value from memory, and record the value read in the log

Sketch of an STM implementation

- On exit from an atomic section, **validate** the log
  - Check that all values read during the transaction match the current value in memory
- If the log is valid, **commit** it
  - Update memory with the writes that were recorded in the log
- If the log is not valid, abort and rety
  - What does it mean for log to not be valid?
  - It read a value from a location that was then updated
  - Another thread modified the location!
  - Throw away the log, and try to re-execute the atomic section
  - Is this safe?

STM issues

- Only effects of atomic section must be reading and writing memory

```
struct node list;

atomic {
    int x = list->val;
    int y = list->next->val;
    if (x > y) launch_missile();
}
```

- How to block until a condition is true?
  - `retry` primitive
  - `atomic {`
    `if (amt > acct_from->balance) retry;`
    `…`
    `}`
- See "Beautiful concurrency" by Simon Peyton Jones for an introduction to STM in the Haskell programming language.

Nondeterminism

- Appropriate abstractions can make concurrent programming easier
- But concurrent programs can still be difficult to reason about, implement, and debug
- One reason: nondeterminism
- Example:

```
acquire(lock);
foo = 7;
release(lock);
```

```
acquire(lock);
foo = 42;
release(lock);
```

  - No data race for foo: access is protected by a lock
  - But there *is* a race for which thread acquires the lock first
  - Behavior of program is **nondeterministic**: sometimes foo=7, sometimes foo=42

Deterministic by default

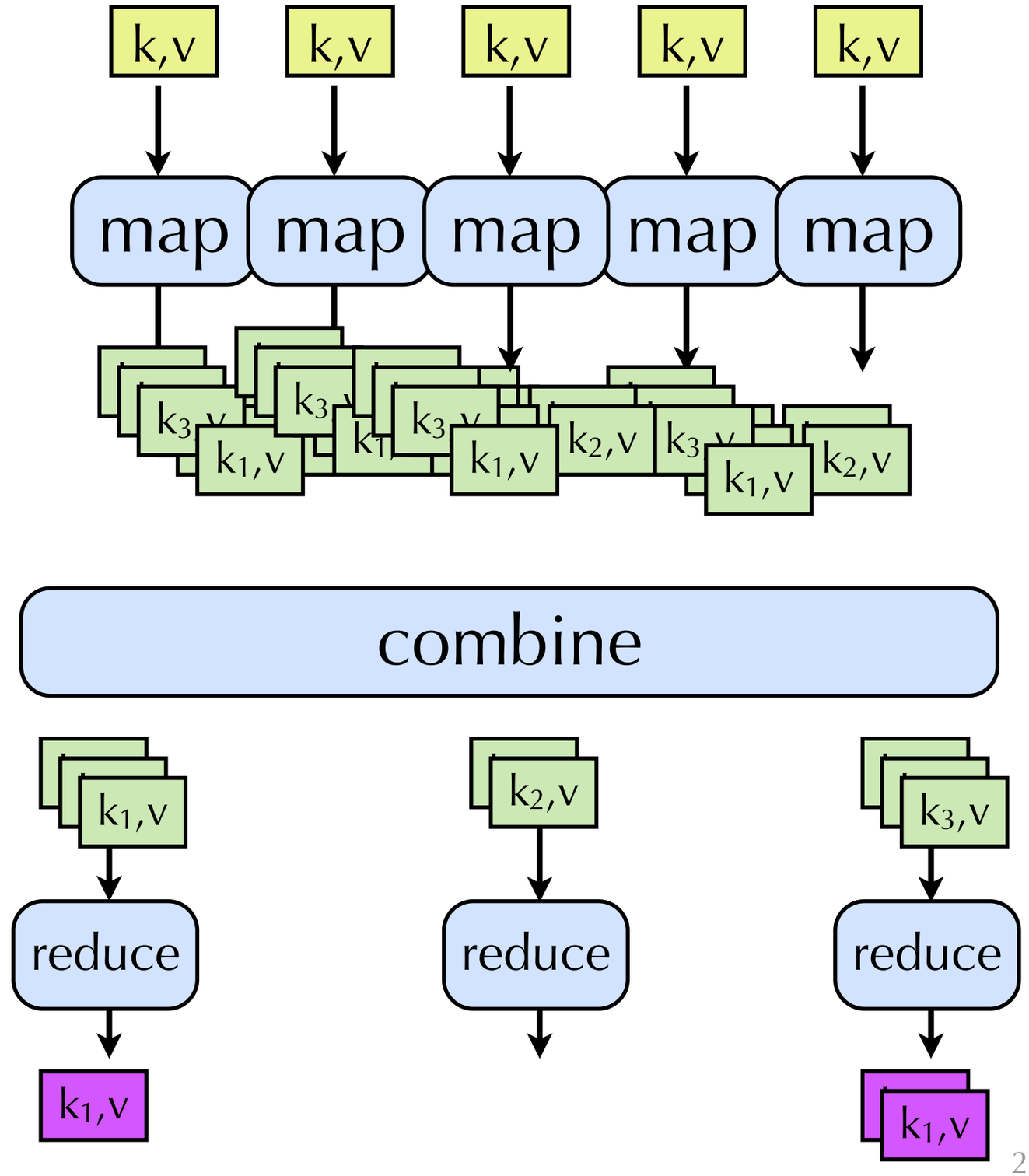
- Deterministic programs
  - Predictable, repeatable behavior even for concurrent programs
  - Enables programmers to reason easily about code, debug code predictably, and diagnose error messages
  - Test cases test functionality, regardless of particular interleavings
  - Program equivalent to its sequential (i.e., non-concurrent) version
- But nondeterminism important for some algorithms
  - E.g., clustering algorithms, optimization algorithms
- Idea: programs should be deterministic by default
  - Non-determinism must be explicitly requested

Map-reduce

- Framework for massively scaled computation
  - Petabytes of data on thousands of machines
  - Widely used at Google
  - Several implementations/variants
    - Hadoop (open source)
    - Dryad (Microsoft)
- Define computation using two functions: **map** and **reduce**
  - Inspired by map and reduce functions used in functional programming (c.f. CS 51!)

Map-reduce overview

- **map** takes a key-value pair as input, and produces a set of key-value pairs
- **reduce** takes a key and a set of values, and produces a set of key-value pairs



Map-reduce examples

- Count number of occurrences of each work in a large collection of documents
  - map(String key, String value) {
    // key is doc name, value is doc contents
    for each word w in value: emit(w, 1)
    }
  - reduce(String key, Set<int> values) {
    // key is word, values is set of counts
    result := 0;
    for each i in values: result += i;
    emit(key, result)
    }

Map-reduce examples

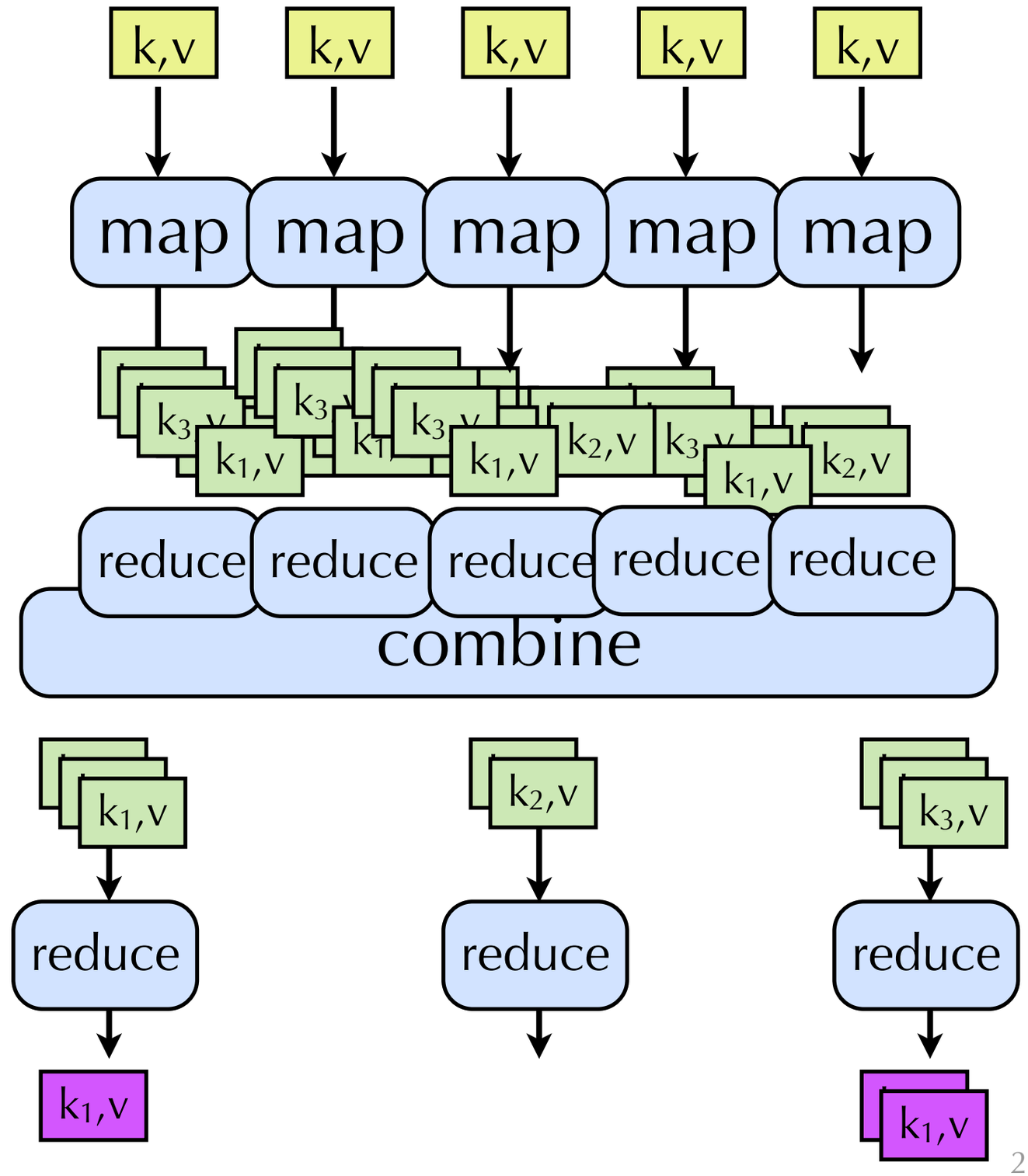
- Reverse web-link graph
  - map takes (URL, webpage) pairs, and emits (target,source) pairs for each link
  - reduce just emits (target, set(sources))
- Count URL access frequency
  - map takes logs of web page requests as values, and outputs (URL, 1) pairs
  - reduce sums up count values

Why is map-reduce effective?

- Shields the programmer from the (massive) parallelism
  - And other issues, such as failure of machines, transfer of data, creating threads, processes, etc.
- If map and reduce are deterministic then result is equivalent to sequential execution
  - Makes it easy to reason about program!

Map reduce

- Skimmed over many details, extensions
  - See "MapReduce: Simplified Data Processing on Large Clusters" by Dean and Ghemawat, OSDI 2004, for an introduction
- If reduce is commutative and associative (which it often is) then opportunity to improve performance without changing result



HARVARD
School of Engineering
and Applied Sciences

The end of CS 61

What we've covered

- Systems Programming and Machine Organization
- An introduction to computer systems
  - Machine representation of data and programs
  - Compilation, optimization, linking, loading
  - Memory, storage, caching, virtual memory, dynamic memory management
  - Systems programming
    - I/O, sockets, processes
  - Concurrency
    - Threads, synchronization mechanisms, synchronization problems
- Answered (to some extent!) the questions
  - What happens when I run a program?
  - How do computers work?
  - What affects the performance and reliability of my programs?

Where to from here?

- We've just scratched the surface!
- Some courses
  - CS 51: more about abstraction
  - CS 141: Computing Hardware
  - CS 148: Design of VLSI Circuits and Systems
  - CS 143: Computer Networks
  - CS 152: Programming languages
  - CS 153: Compilers
  - CS 161: Operating systems
  - CS 164: Mobile Software Engineering
  - CS 165: Information Management (databases)
  - CS 171: Visualization
  - CS 175: Computer graphics
  - CS 179: Design of Usable Interactive Systems
  - CS 189r: Autonomous multi-robot systems
  - Other 100 level courses and many 200 level courses...

- Harvard Computer Society http://www.hcs.harvard.edu
- Systems Research at Harvard http://www.eecs.harvard.edu/~syrah/
- Programming Languages at Harvard http://www.cs.harvard.edu/pl/