TDDB68/TDDE47ting Systems

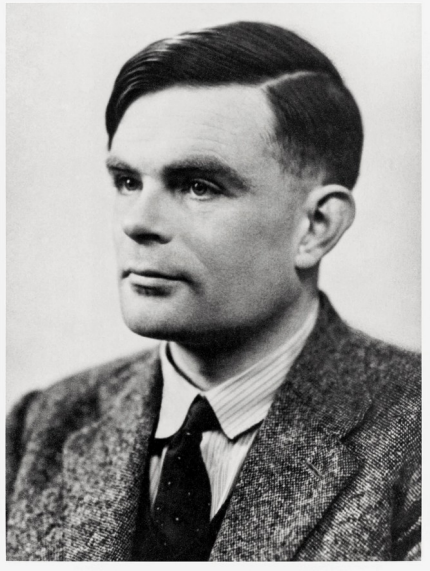
Lecture:
Virtualization + Synchronization II

Mikael Asplund, Senior Lecturer
Real-time Systems Laboratory
Department of Computer and Information Science

Thanks to Christoph Kessler for much of the material behind these slides.

”It is possible to invent a single machine which can be used to compute any computable sequence. If this machine U is supplied with a tape on the beginning of which is written the S.D of some computing machine M, then U will compute the same sequence as M.”

Alan Turing 1936



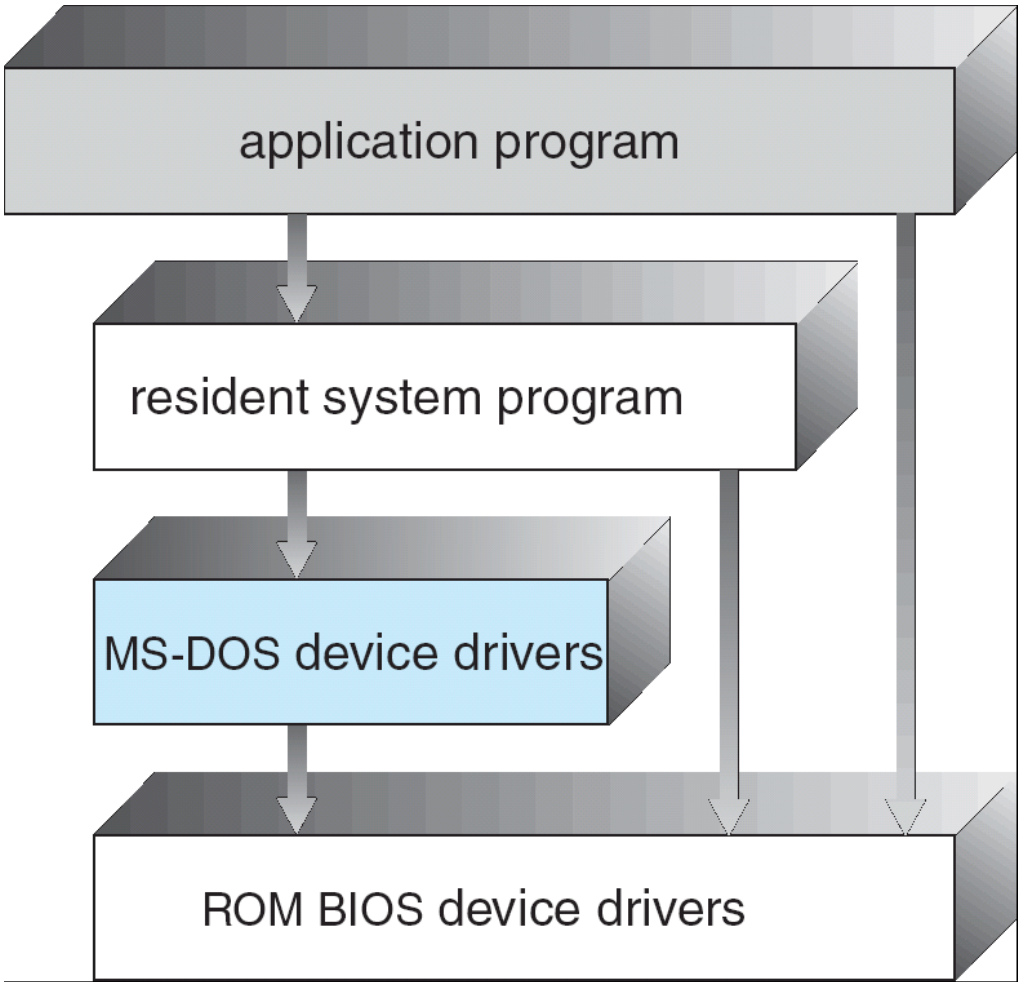
Functional
vs.
Non-functional

OS structures

Operating System Structures

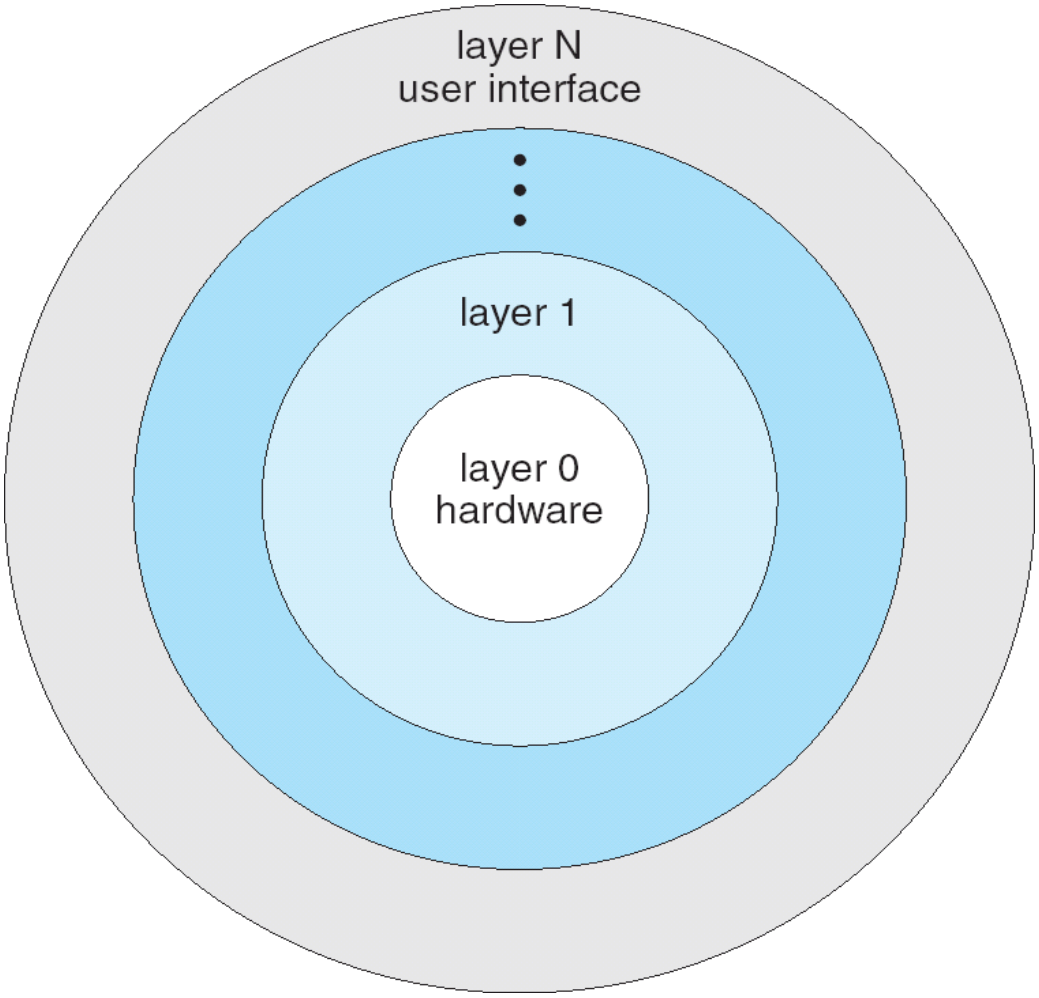
- How to manage OS complexity?
  - Divide-and-conquer!
  - Decompose into smaller components with well-defined interfaces and dependences
    - Layered Approach
    - Microkernels
    - Modules
    - Virtual Machines

Simple Structure (MSDOS)



Layered Approach

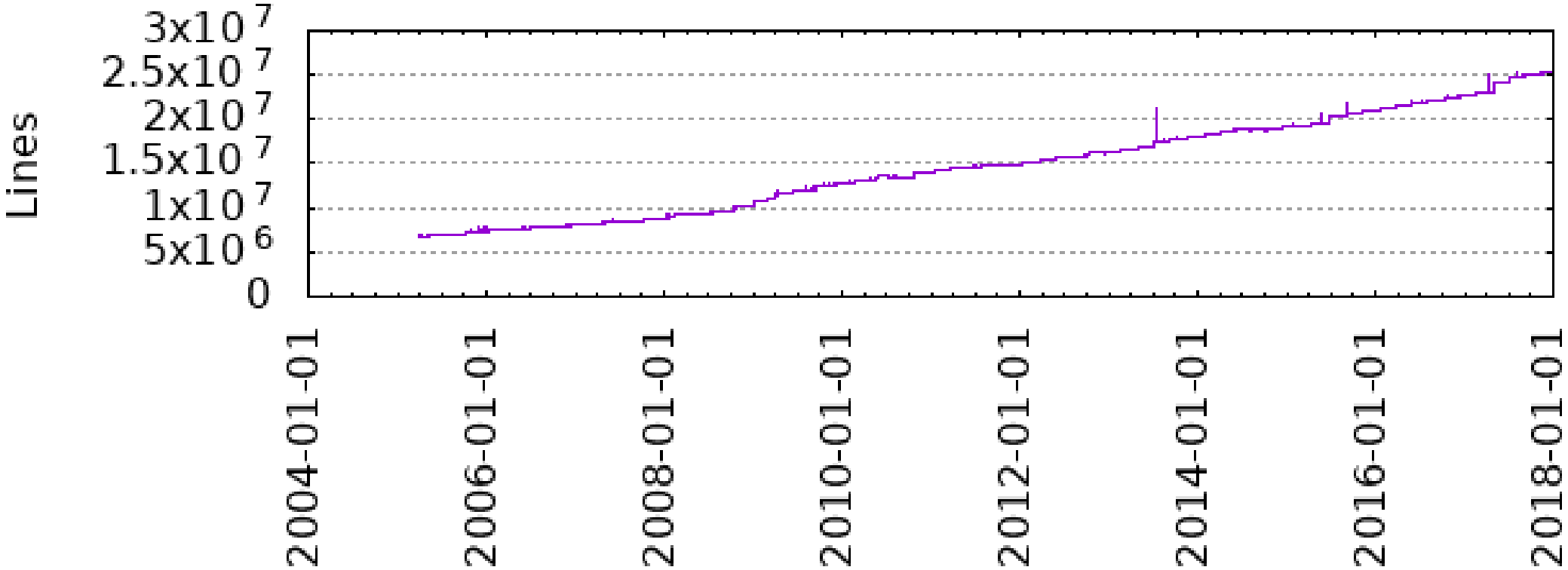
- The operating system is divided into a number of layers (levels, rings), each built on top of lower layers.
- Functions in layer *i* call only functions/services in layers ≤ *i* (**strict layering:** only in *i* or *i-1*)



Problems of the layered approach

- **Cyclic dependences** between different OS components
- **Less efficient**
  - Long call chains (e.g. I/O) down to system calls, possibly with parameter copying/modification at several levels
- Compromise solution: Have few layers

Linux kernel source code size



Microkernel System Structure

- “Lean kernel”: Moves as much service functionality as possible from the kernel into “*user*” space
  - Kernel: Minimal process and memory management; IPC
- Communication between user modules by message passing
- Example: Mach kernel, used e.g. in Tru64 Unix or Mac OS-X

Microkernel Pros and Cons

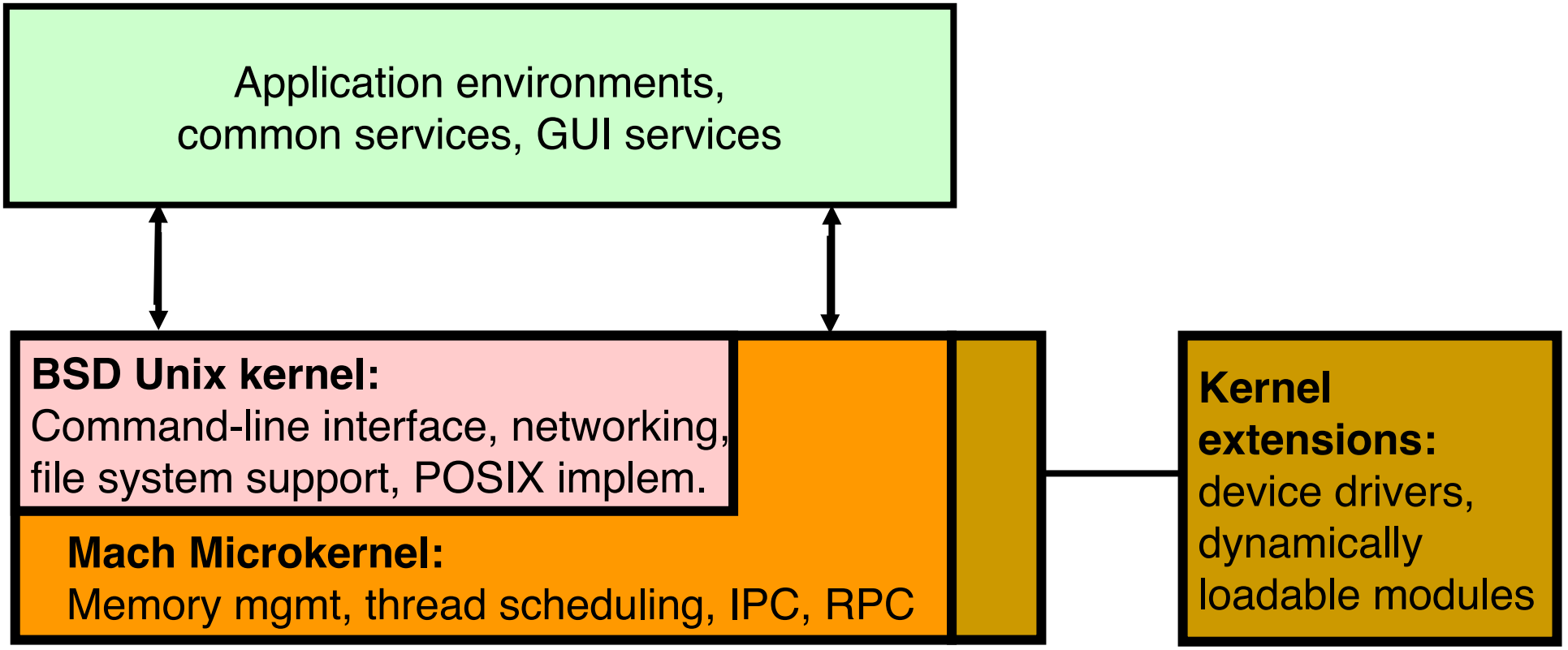
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication
  - More complicated synchronization

Modules

- Most modern operating systems implement kernel modules
- Component-based approach:
  - Each core component is separate
  - Each talks to the others over known interfaces
  - Each is loadable as needed within the kernel
- Overall, similar to layers
  but more flexible

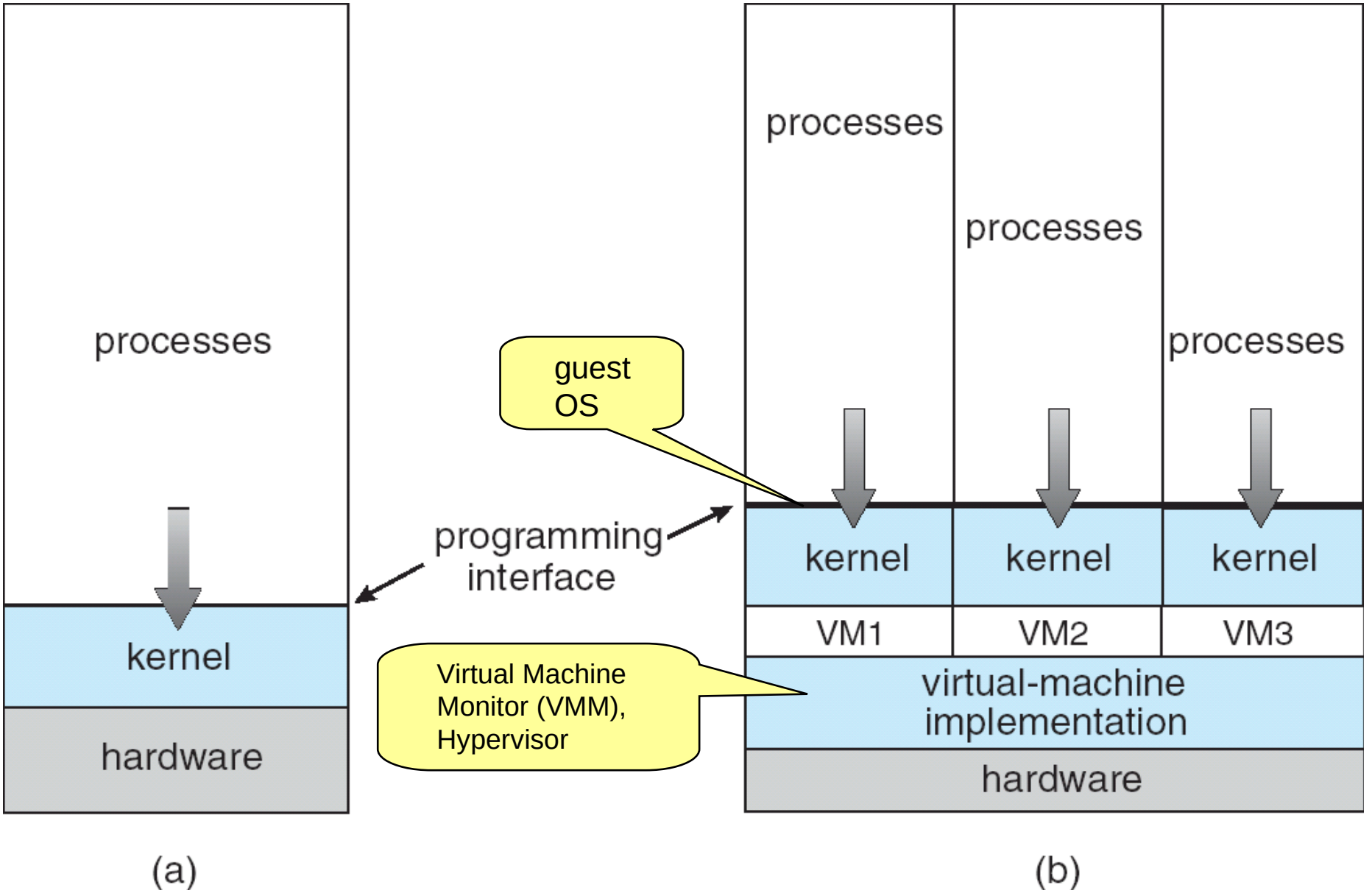
Example: Mac-OS X ”Darwin”

- Hybrid structure: Layering + Microkernel + Modules



Virtual Machines

Virtual Machines



(a) Nonvirtual machine

(b) Virtual machines

Why?

NF properties to consider

- Resource efficiency
- Security and fault tolerance
  - (through protection)
- Flexibility
- Responiveness for an individual unit

Virtual Machine Examples

- Multitasking OS creates the illusion that each process executes on its own (virtual) processor with its own (virtual) memory.
- qemu (used in Pintos labs) simulates x86 hardware
- The Java VM simulates an abstract computer that executes Java bytecode.

Implementation of virtualization

- Full virtualization
  - Emulation
- Hardware-assisted virtualization
  - Uses hypervisors
- Paravirtualization
  - Requires modification of guest OS
- Application containment

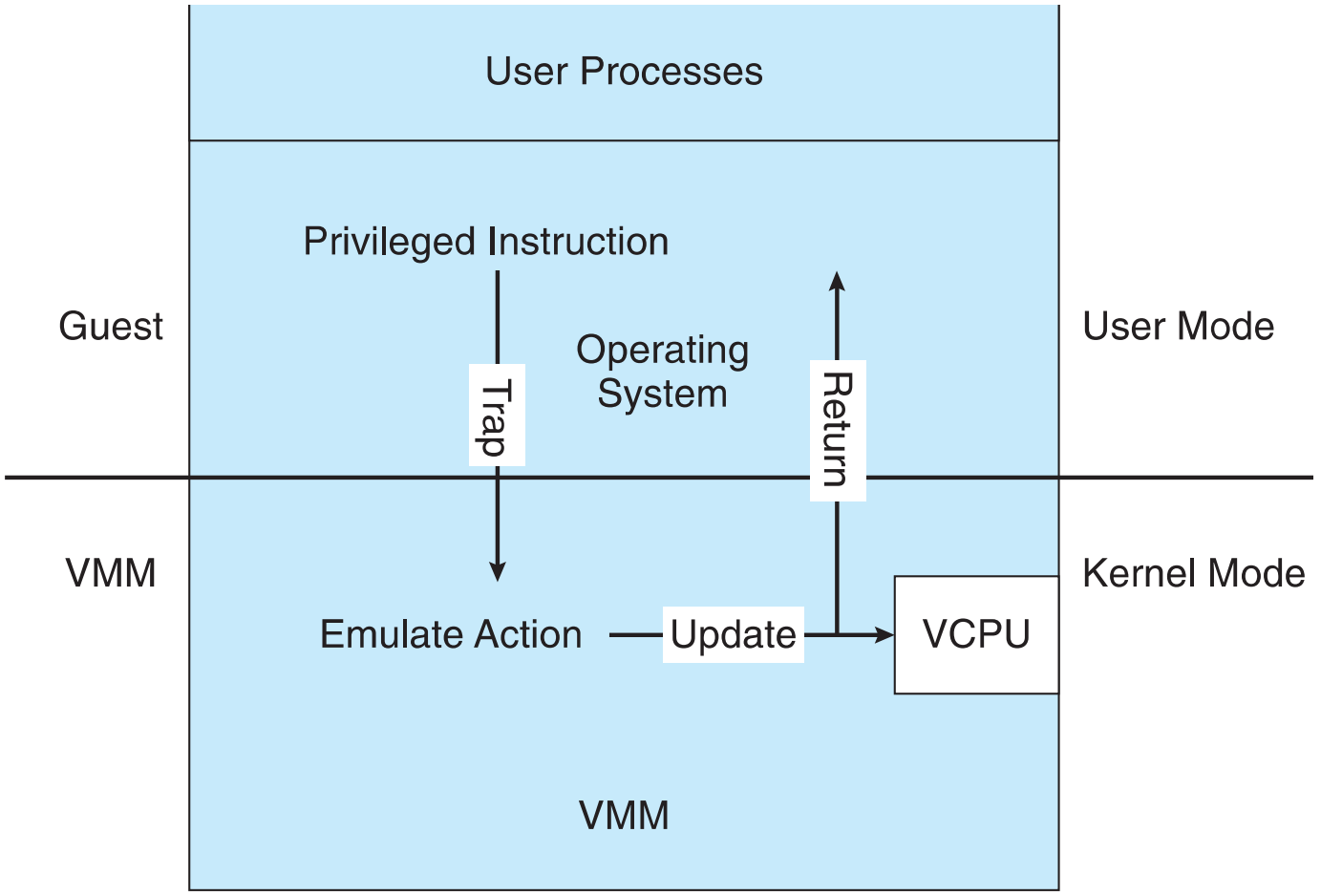
Building blocks

- Trap and emulate
- Binary translation
- Nested page tables
- More HW assistance

Trap and Emulate

- Dual mode CPU means guest executes in user mode
  - Kernel runs in kernel mode
  - Not safe to let guest kernel run in kernel mode too
  - So VM needs two modes – virtual user mode and virtual kernel mode
    - Both of which run in real user mode
  - Actions in guest that usually cause switch to kernel mode must cause switch to virtual kernel mode

Trap-and-Emulate Virtualization Implementation



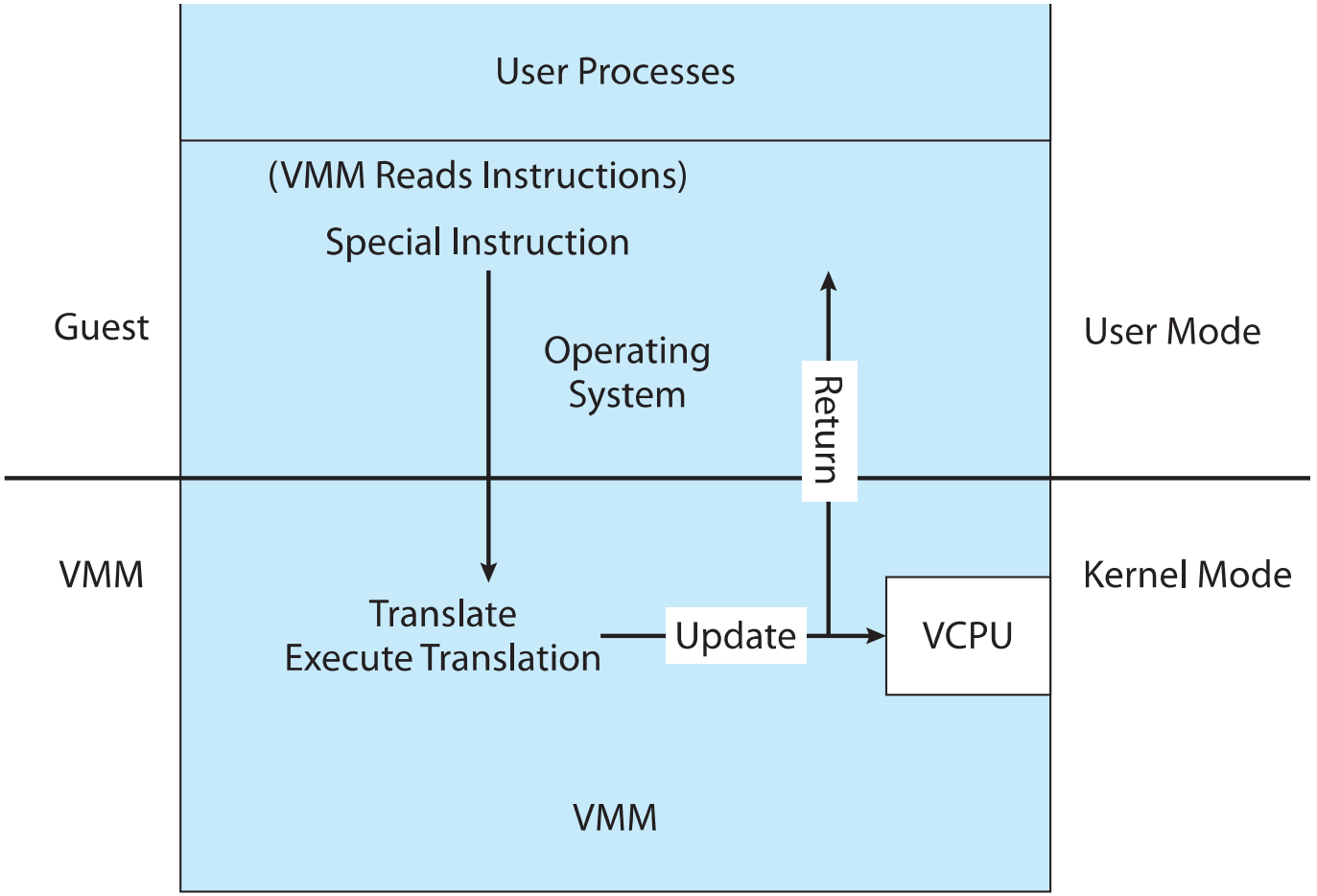
Binary Translation

- Some CPUs don't have clean separation between privileged and nonprivileged instructions
  - Earlier Intel x86 CPUs are among them
    - Earliest Intel CPU designed for a calculator
  - Backward compatibility means difficult to improve
  - Consider Intel x86 `popf` instruction
    - Loads CPU flags register from contents of the stack
    - If CPU in privileged mode -> all flags replaced
    - If CPU in user mode -> on some flags replaced
      - No trap is generated

Binary Translation (Cont.)

- Other similar problem instructions we will call ***special instructions***
  - Caused trap-and-emulate method considered impossible until 1998
- Binary translation solves the problem
  1. Basics are simple, but implementation very complex
  2. If guest VCPU is in user mode, guest can run instructions natively
  3. If guest VCPU in kernel mode (guest believes it is in kernel mode)
     a) VMM examines every instruction guest is about to execute by reading a few instructions ahead of program counter
     b) Non-special-instructions run natively
     c) Special instructions translated into new set of instructions that perform equivalent task (for example changing the flags in the VCPU)

Binary Translation Virtualization Implementation



Nested Page Tables

- Memory management is a challenge
  - Who controls the page tables?

- Common method (for trap-and-emulate and binary translation) is **nested page tables** (**NPTs**)
  - Each guest maintains page tables to translate virtual to physical addresses
  - VMM maintains per guest NPTs to represent guest's page-table state
    - Just as VCPU stores guest CPU state
  - Page table modifications must be propagated
  - Can cause many more TLB misses -> much slower performance

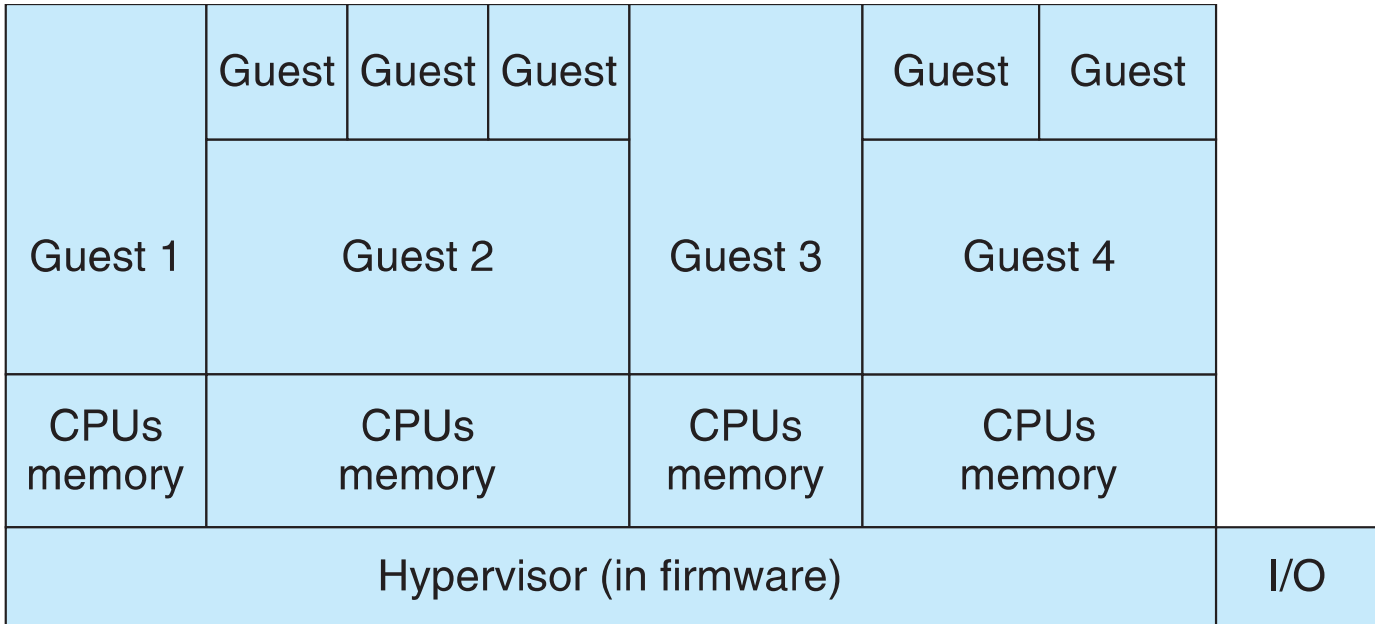
More Hardware Assistance

- More support -> more feature rich, stable, better performance of guests
- Intel added new **VT-x** instructions in 2005 and AMD the **AMD-V** instructions in 2006
  - CPUs with these instructions remove need for binary translation
  - Generally define more CPU modes – “guest” and “host”
  - VMM can enable host mode, define characteristics of each guest VM, switch to guest mode and guest(s) on CPU(s)
  - In guest mode, guest OS thinks it is running natively, sees devices (as defined by VMM for that guest)
    - Access to virtualized device, priv. instructions cause trap to VMM
    - CPU maintains VCPU, context switches it as needed
- New examples and variants appear over time

Types of VMs – Emulation

- Another (older) way for running one operating system on a different operating system
  - Most current virtualization techniques requires host CPU = guest CPU
  - Emulation allows guest to run on different CPU
- Necessary to translate all guest instructions from guest CPU to native CPU
- Useful when host system has one architecture, guest compiled for other architecture
  - Company replacing outdated servers with new servers containing different CPU architecture, but still want to run old applications
- Performance challenge – order of magnitude slower than native code
- Very popular – especially in gaming where old consoles emulated on new

Type 0 Hypervisor



Types of VMs – Type 1 Hypervisor

- Commonly found in company datacenters
  - In a sense becoming “datacenter operating systems”
    - Datacenter managers control and manage OSes in new, sophisticated ways by controlling the Type 1 hypervisor
    - Consolidation of multiple OSes and apps onto less HW
    - Move guests between systems to balance performance
    - Snapshots and cloning

Types of VMs – Type 1 Hypervisor (Cont.)

- Special purpose operating systems that run natively on HW
  - Rather than providing system call interface, create run and manage guest OSes
  - Can run on Type 0 hypervisors but not on other Type 1s
  - Run in kernel mode
  - Guests generally don't know they are running in a VM
  - Implement device drivers for host HW because no other component can
  - Also provide other traditional OS services like CPU and memory management

Types of VMs – Type 1 Hypervisor (Cont.)

- Another variation is a general purpose OS that also provides VMM functionality
  - RedHat Enterprise Linux with KVM, Windows with Hyper-V, Oracle Solaris
  - Perform normal duties as well as VMM duties
  - Typically less feature rich than dedicated Type 1 hypervisors
- In many ways, treat guests OSes as just another process
  - Albeit with special handling when guest tries to execute special instructions

Types of VMs – Type 2 Hypervisor

- Very little OS involvement in virtualization
- VMM is simply another process, run and managed by host
  - Even the host doesn't know they are a VMM running guests
- Tend to have poorer overall performance because can't take advantage of some HW features
- But also a benefit because require no changes to host OS
  - Student could have Type 2 hypervisor on native host, run multiple guests, all on standard host OS such as Windows, Linux, MacOS

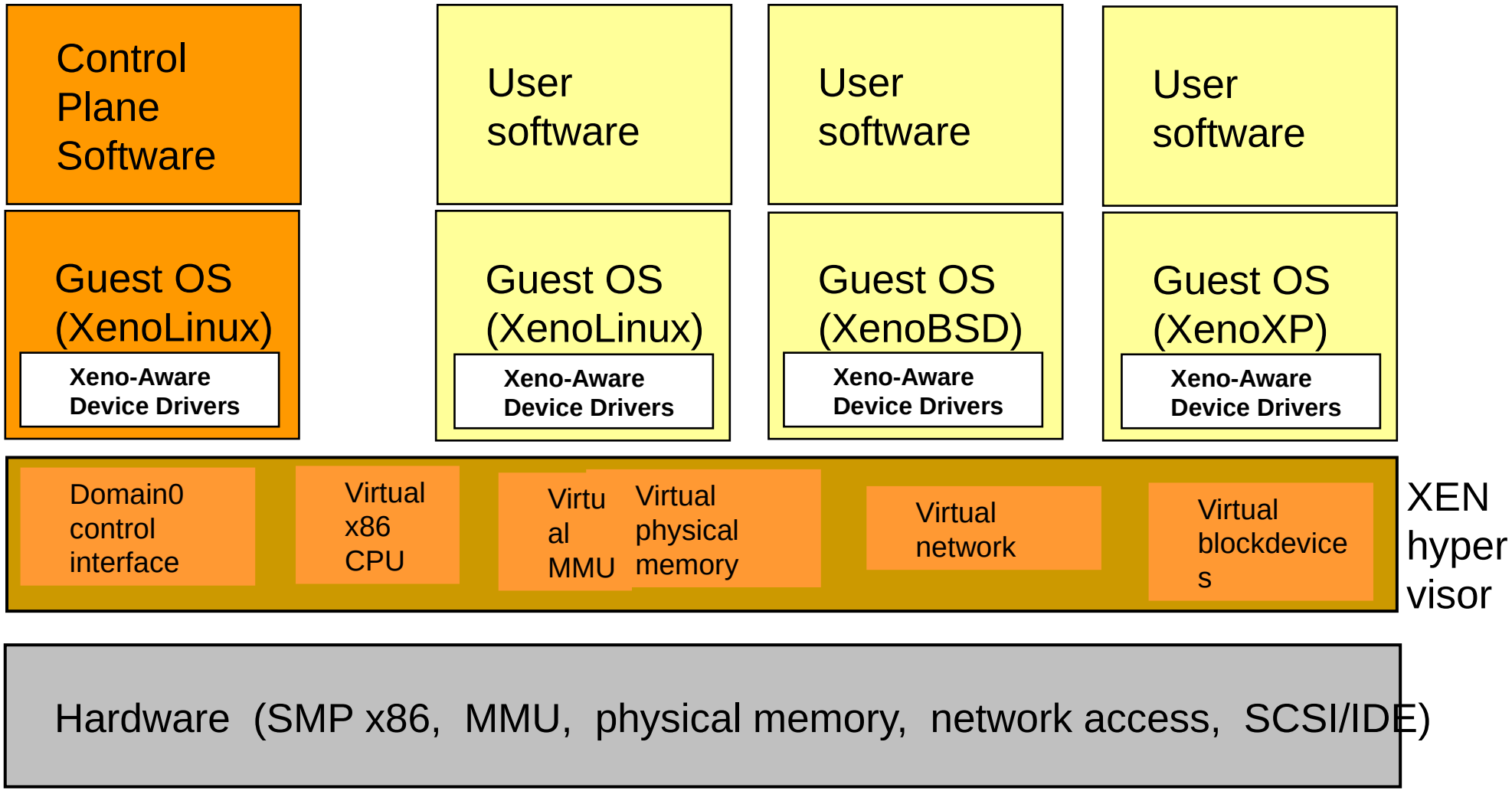
Examples - VMware

- VMware Workstation runs on x86, provides VMM for guests
- Runs as application on other native, installed host operating system -> Type 2
- Lots of guests possible, including Windows, Linux, etc. all runnable concurrently (as resources allow)
- Virtualization layer abstracts underlying HW, providing guest with is own virtual CPUs, memory, disk drives, network interfaces, etc.
- Physical disks can be provided to guests, or virtual physical disks (just files within host file system)

Types of VMs – Paravirtualization

- Does not fit the definition of virtualization – VMM not presenting an exact duplication of underlying hardware
  - But still useful!
  - VMM provides services that guest must be modified to use
  - Leads to increased performance
  - Less needed as hardware support for VMs grows
- Xen, leader in paravirtualized space, adds several techniques
  - For example, clean and simple device abstractions
    - Efficient I/O
    - Good communication between guest and VMM about device I/O
    - Each device has circular buffer shared by guest and VMM via shared memory

Paravirtualization Example: Xen

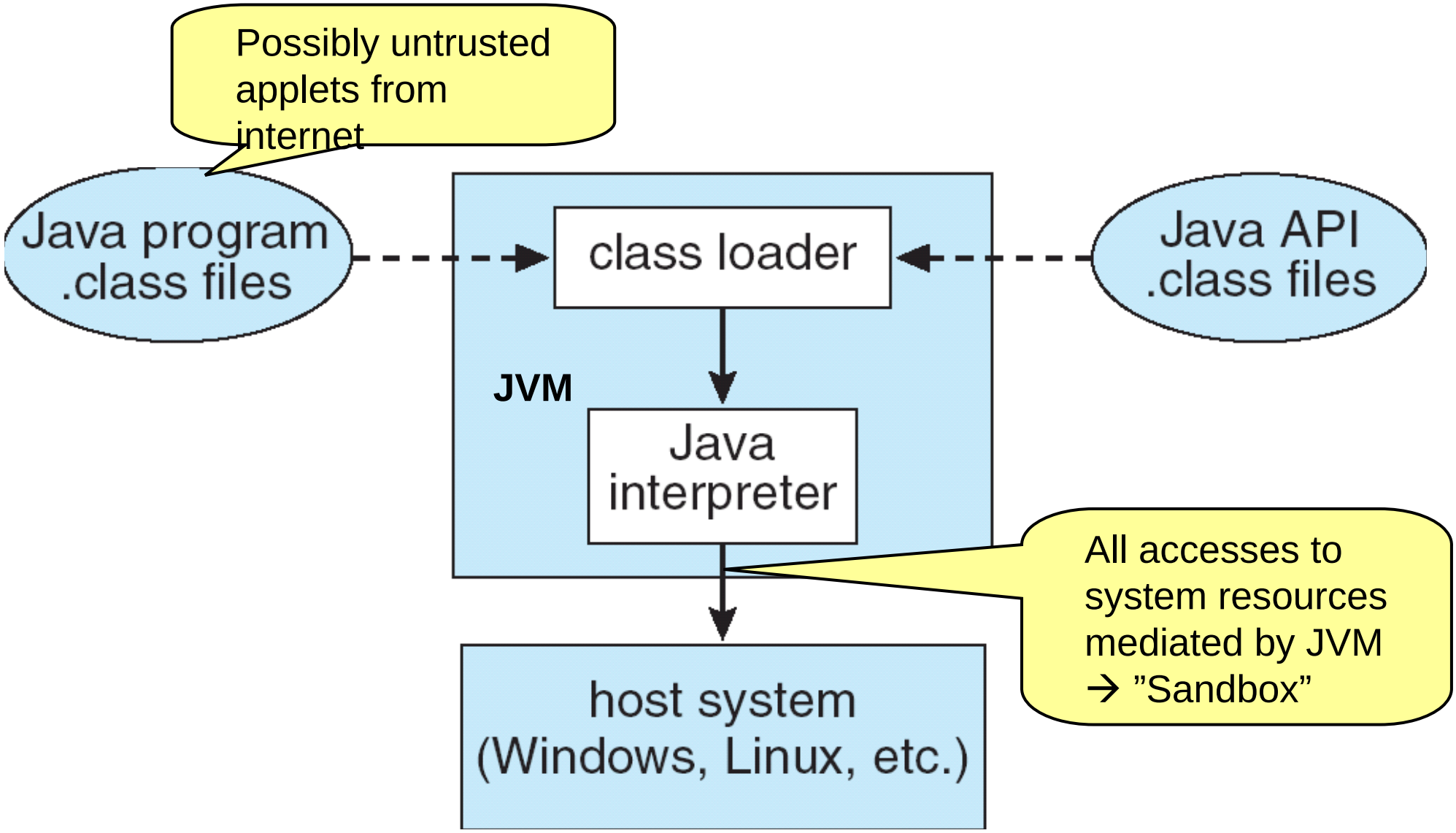


Adapted from: P. Barham et al.: Xen and the Art of Virtualization. Proc. SOSP 2003

Types of VMs – Programming Environment Virtualization

- Programming language is designed to run within custom-built virtualized environment
  - For example Oracle Java has many features that depend on running in **Java Virtual Machine** (**JVM**)
- In this case virtualization is defined as providing APIs that define a set of features made available to a language and programs written in that language to provide an improved execution environment
- JVM compiled to run on many systems (including some smart phones even)
- Programs written in Java run in the JVM no matter the underlying system
- Similar to **interpreted languages**

The Java Virtual Machine



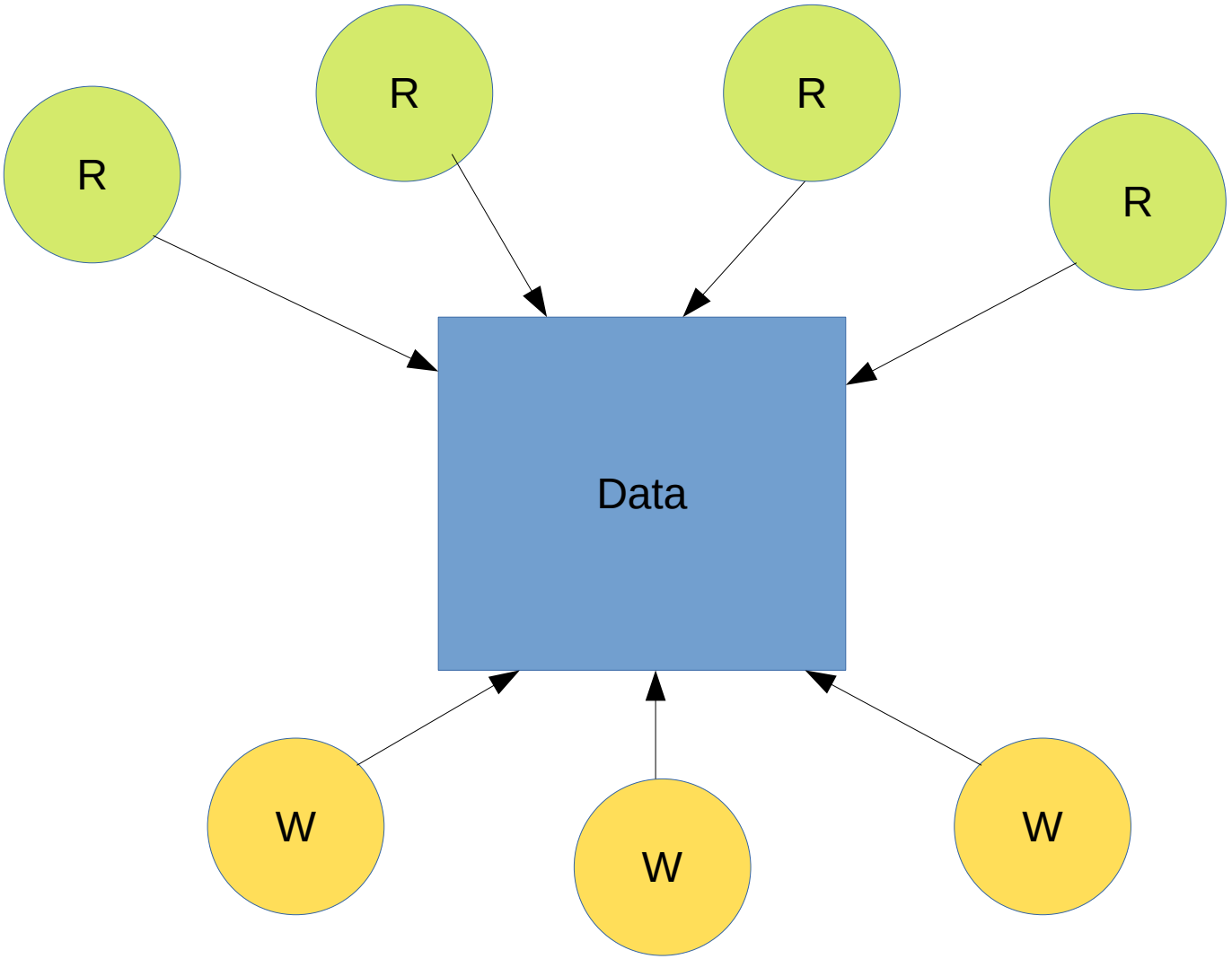
Types of VMs – Application Containment

- Virtualization still costly!
- Oracle **containers** / **zones** for example create virtual layer between OS and apps
  - Only one kernel running – host OS
  - OS and devices are virtualized, providing resources within zone with impression that they are only processes on system
  - Each zone has its own applications; networking stack, addresses, and ports; user accounts, etc
  - CPU and memory resources divided between zones
    - Zone can have its own scheduler to use those resources
- Popular today: Docker

Synchronization II

Readers-writer problem (relevant for lab 6)

Who gets access?



Shared data

- Data set
- Semaphore **`rw_mutex`** initialized to 1
- Semaphore **`mutex`** initialized to 1
- Integer **`read_count`** initialized to 0

Writer process

```
while (true) {
   wait(rw_mutex);
         ...
   /* writing is performed */
         ...
    signal(rw_mutex);
}
```

Reader process

```
while (true){
        wait(mutex);
        read_count++;
        if (read_count == 1) /* first reader */
            wait(rw_mutex);
        signal(mutex);
          ...
        /* reading is performed */
          ...
        wait(mutex);
        read count--;
        if (read_count == 0) /* last reader */
            signal(rw_mutex);
        signal(mutex);
}
```

Readers-Writers Problem Variations

- First reader-writer problem
  - Once a reader has access, readers will be prioritized over writers
  - Writers starve
- Second reader-writer problem
  - Once a writer is ready to write, no “newly arrived reader” is allowed to read.
  - Readers starve
- Third reader-writer problem
  - Implement a service queue
  - Complex
- Problem is solved on some systems by kernel providing reader-writer locks

Lock-free concurrent programming

Lock-based solution

```
Process P1
while true {
  acquire(lock)
  critical_section
  release(lock)
}
```

```
Process P2
while true {
  acquire(lock)
  critical_section
  release(lock)
}
```

Basic definitions

- A **lock-free** algorithm guarantees that at least one process can make progress within a finite time
  - Also called non-blocking
- A lock-free algorithm is **wait-free** if every process makes progress within some finite time

Primitive

- Compare and Swap (CAS):

```
boolean CAS (int *p, int old, int new) {
    if (*p != old) {
      return FALSE;
    } else {
      *p = new;
      return True;
    }
 }
```

Simple lock-free stack algorithm

- Due to Treiber 1986
- This presentation based on Michael and Scott 1998 (JPDC)

Data structures

- struct pointer_t {
  node_t* ptr;
  uint count;
  }
- struct node_t {
  int value;
  pointer_t next;
  }
- struct stack_t {
  pointer_t top;
  }

Push

- push(stack_t* S, int value) {
  node = new_node();
  node->value = value;
  node->next.ptr = NULL;
  repeat
    top = S->top;
    node->next.ptr = top.ptr;
  until CAS(&S->top, top, [node, top.count+1]);
  }

Pop

- pop(stack_t* S, int *pvalue) {

  repeat

  top = S->top;

  if top.ptr == NULL

  return False;

  until CAS(&S->top, top, [top.ptr->next.ptr, top.count+1]);

  *pvalue = top.ptr->value;

  free(top.ptr);

  return True;

  }

In general

- Some lock-free algorithms provide reasonable performance
- Wait-free algorithms have low performance
- Complex to create
- More library support is coming