Introduction to Linux Kernel

Prof. K. Gopinath
IISc

References:

- **Linux Kernel source!**
  - ***esp: /usr/src/linux/Documentation***
  - ***lxr.linux.no* is a nice web-based source browser**
- **Linux Kernel 2.4 Internals, *Tigran Aivazian***
- **Understanding the Linux Kernel, *Bovet/Cesati***
- **The Linux Kernel, *David A Rusling***
- **Tour of the Linux Kernel source, *Alessandro Rubini***
- **Linux Loadable Kernel Module HOWTO, *Bryan Henderson***

Objectives

- Review **development and history** of UNIX and Linux.
- Clarify the **nature and extent of the kernel** within the context of the operating system proper.
- Identify operating system **design goals and tradeoffs**.
- Review established **architectural approaches** for implementing an operating system.
- Introduce the notion of the **“core” Linux kernel**.
- Understand the Linux approach to **hardware (architecture) abstraction and independence**.
- Explore the layout of the Linux **source code tree**.

History

- UNIX: 1969 Thompson & Ritchie AT&T Bell Labs
- BSD: 1978 Berkeley Software Distribution
- Commercial Vendors: Sun, HP, IBM, SGI, DEC
- GNU: 1984 Richard Stallman, FSF
- POSIX: 1986 IEEE Portable Operating System unIX
- Minix: 1987 Andy Tannenbaum
- SVR4: 1989 AT&T and Sun
- Linux: 1991 Linus Torvalds Intel 386 (i386)
- Open Source: GPL, LGPL, Cathedral and the Bazaar

GNU/Linux Features

- "UNIX-like": Multi-user, multi-tasking, UNIX system
- Goals
  - Speed, efficiency
  - "aims at" standards compliance (e.g., POSIX)
- "all the features you would expect in a modern UNIX"
  - preemptive multitasking
  - virtual memory (protected memory, paging)
  - shared libraries
  - demand loading, dynamic kernel modules
  - shared copy-on-write executables
  - TCP/IP networking
- other features:
  - SMP support, large memory, large files
  - advanced networking, advanced filesystems
  - efficient, stable, highly portable, supports most device hardware
  - active development community, support, documentation, open source
  - GUIs, applications
- Components: kernel (VM, proc mgmt), libraries (syscalls, buf I/O), system utils (netw daemons)

What's a Kernel?

(Also: executive, system monitor, nucleus)

- controls and mediates access to hardware
- implements and supports fundamental abstractions
  - processes, files, devices, users, net, etc.
- schedules "fair" sharing of system resources
  - memory, cpu, disk, descriptors, etc.
- enforces security and protection
- responds to user requests for service (system calls)
- performs routine maintenance, system checks, etc.

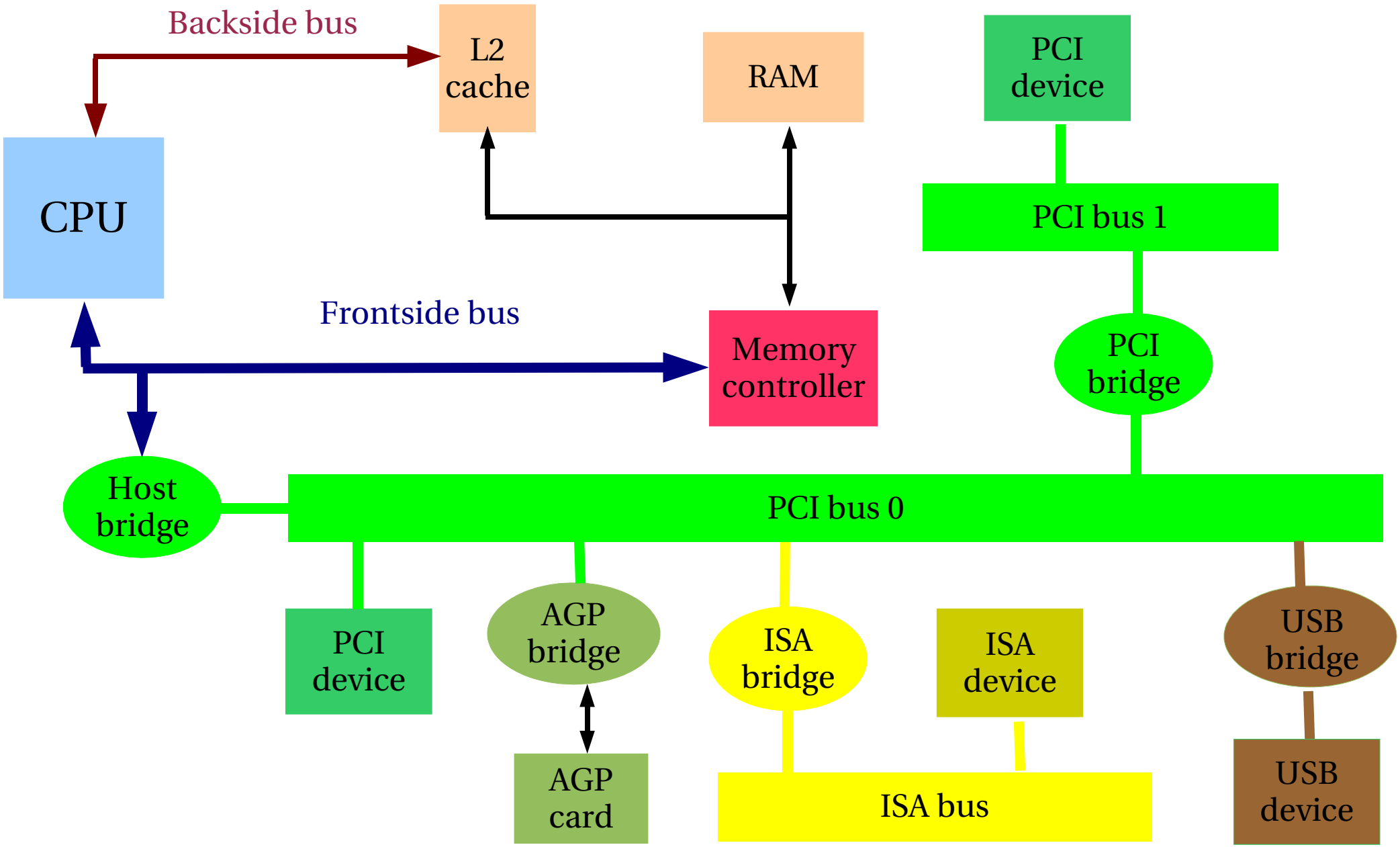
Highly concurrent!

Non-stop if possible! Even modifications on a live kernel

Highly extensible!

Needs to be designed to survive for a few decades (1 or 2) atleast!
Software arch/engg critical to success

Typical Modern PC Arch

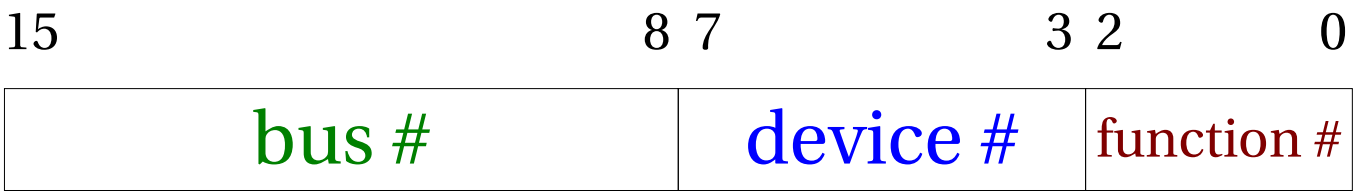


PCI: Peripheral Component Interconnect

- PCI is a standard for I/O buses and device interfacing
- Introduced in 1991 by Intel and many others
- Used on architectures IA-32, Alpha, PowerPC, SPARC64, ...
- 32-bit internal bus (64 bit in PCI 2.1)
- Clock speed: 25/33MHz
  (66MHz in PCI 2.1, 133MHz in PCI-X)
- It allows transfers in *burst mode*
- It allows *bus mastering*
- PCI devices are *plug and play*
- PCI devices may be *hot-pluggable (i.e.: CardBus)*

Buses, devices and functions (1)

Each PCI peripheral is identified by a 16-bit number:



- *bus number* : a PCI bus in the system (0-255)
- *device number* : a device (slot) on the bus (0-31)
- *function number* : a function of the device (0-7)

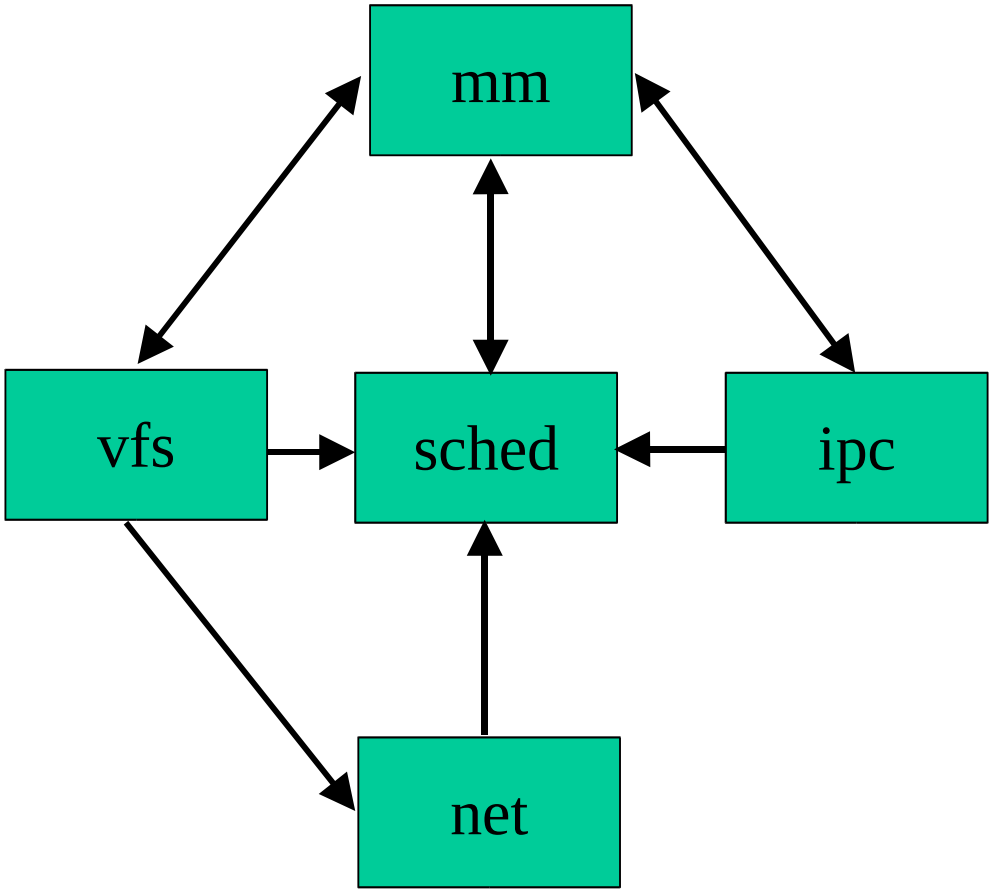
Kernel Design Goals

- performance: efficiency, speed
  - utilize resources to capacity, low overhead, code size
- stability: robustness, resilience
  - uptime, graceful degradation
- capability: features, flexibility, compatibility
- security, protection
  - protect users from each other, secure system from bad guys
- portability
- clarity
- extensibility

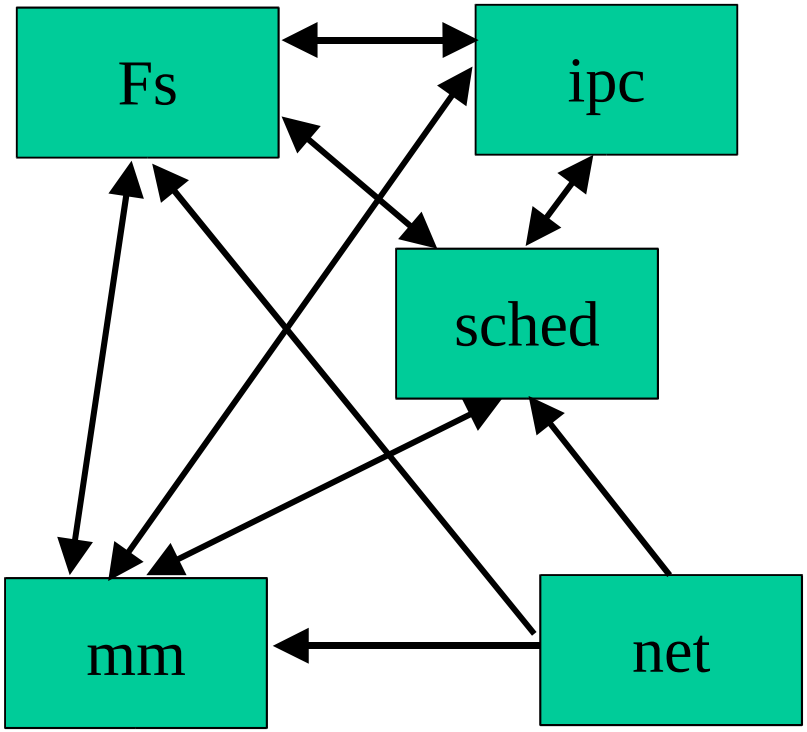
Design Tradeoffs

- Butler Lampson: “choose any three design goals”
- efficiency vs. protection
  - more checks, more overhead
- clarity vs. compatibility
  - ugly implementation of “broken” standards (e.g. signals)
- flexibility vs. security
  - the more you can do, the more potential security holes!
- not all are antagonistic
  - portability tends to enhance code clarity

Dependency Diagrams

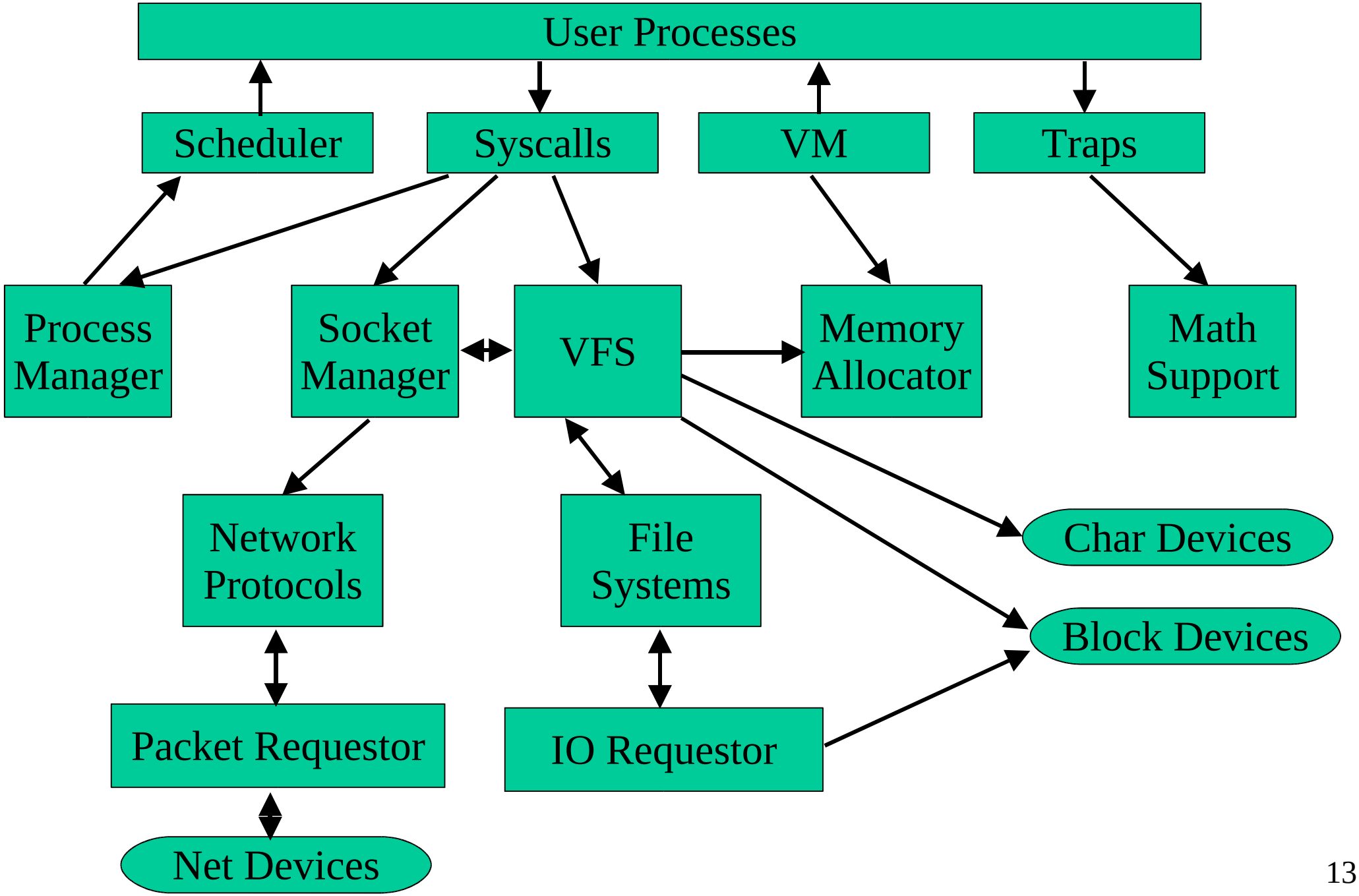


Conceptual



Concrete

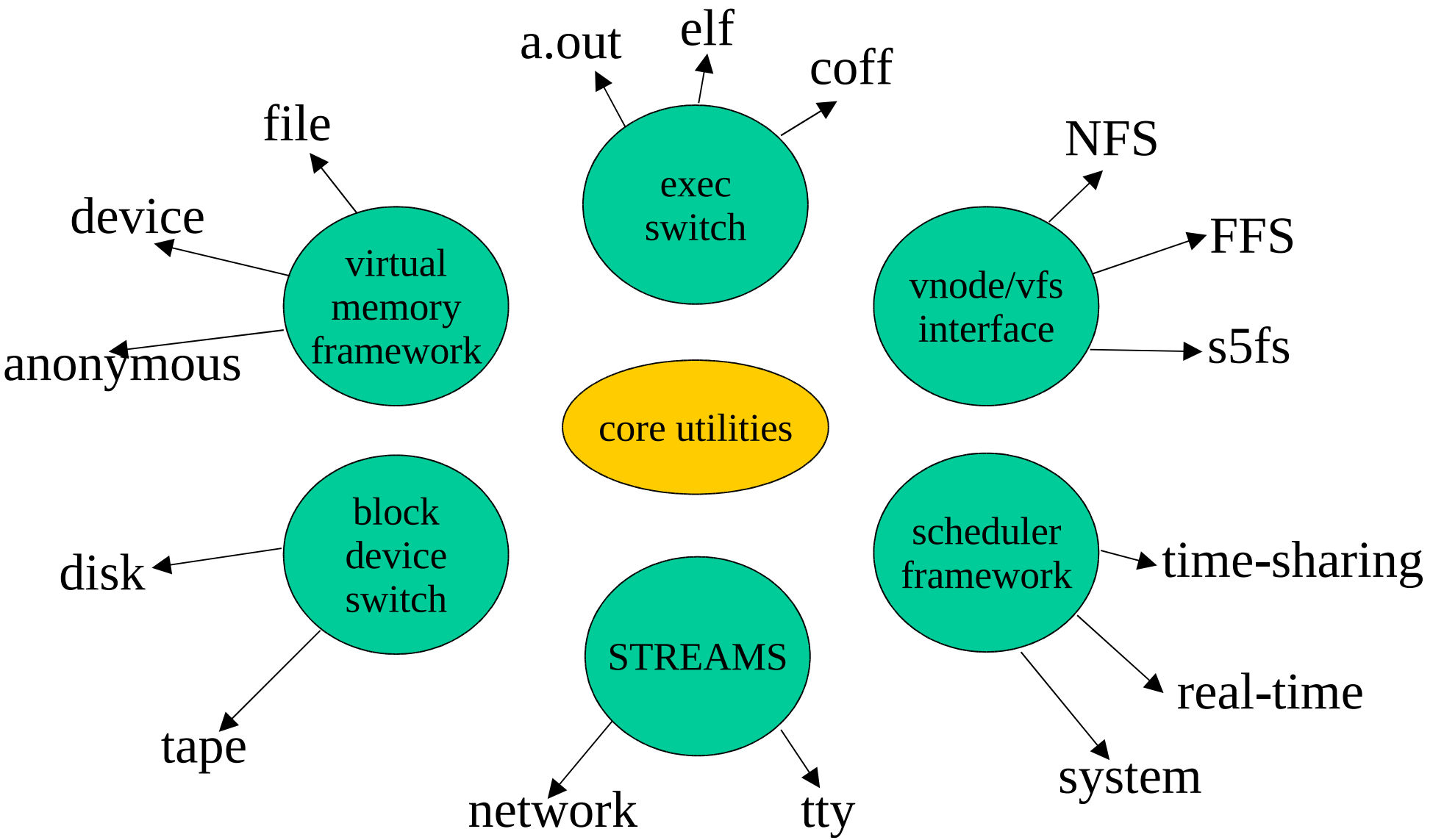
Stephen Tweedie's Diagram



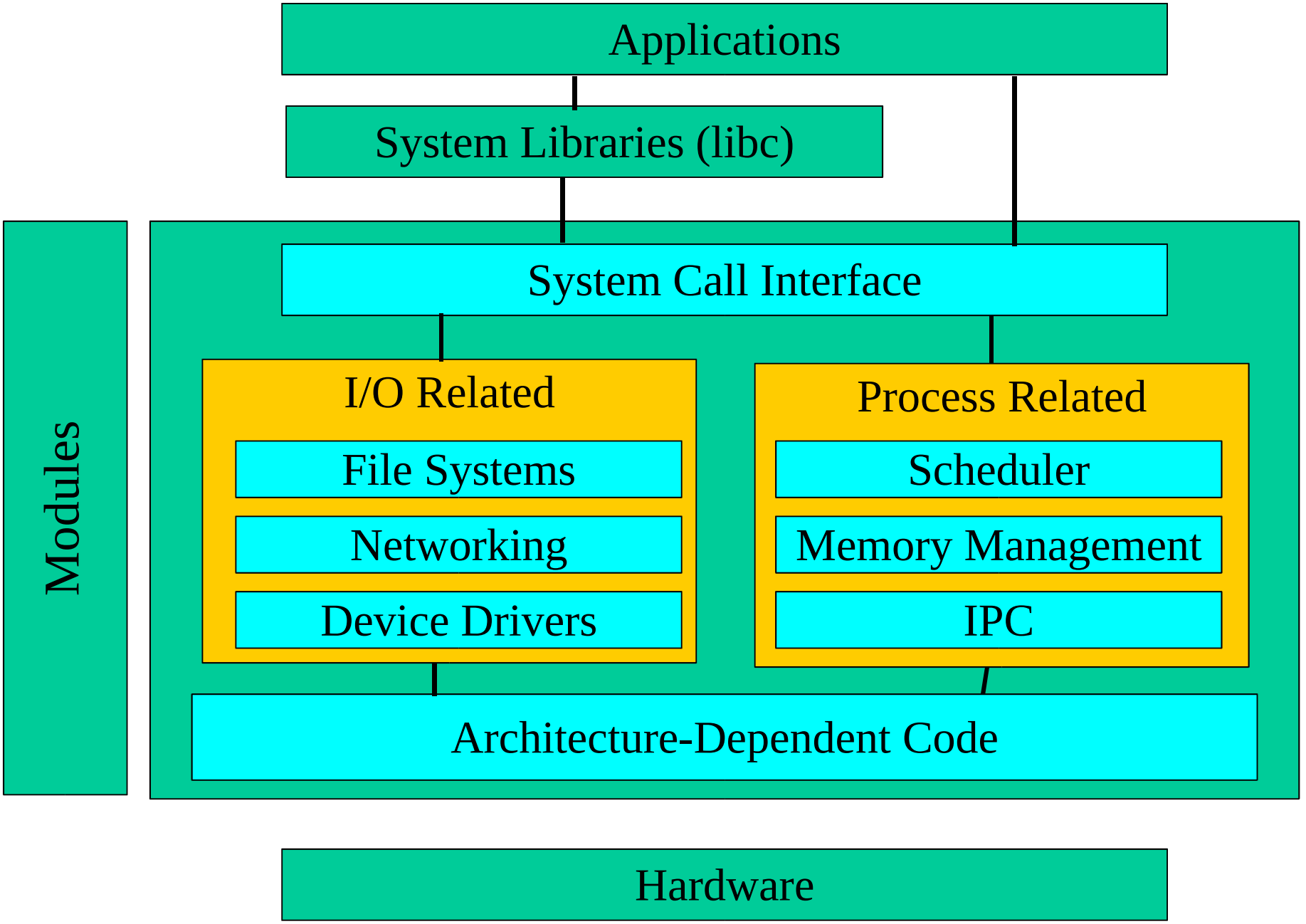
Vahalia's Diagram

from *Unix Internals: The New Frontiers*

Uresh Vahalia / Prentice-Hall 1996

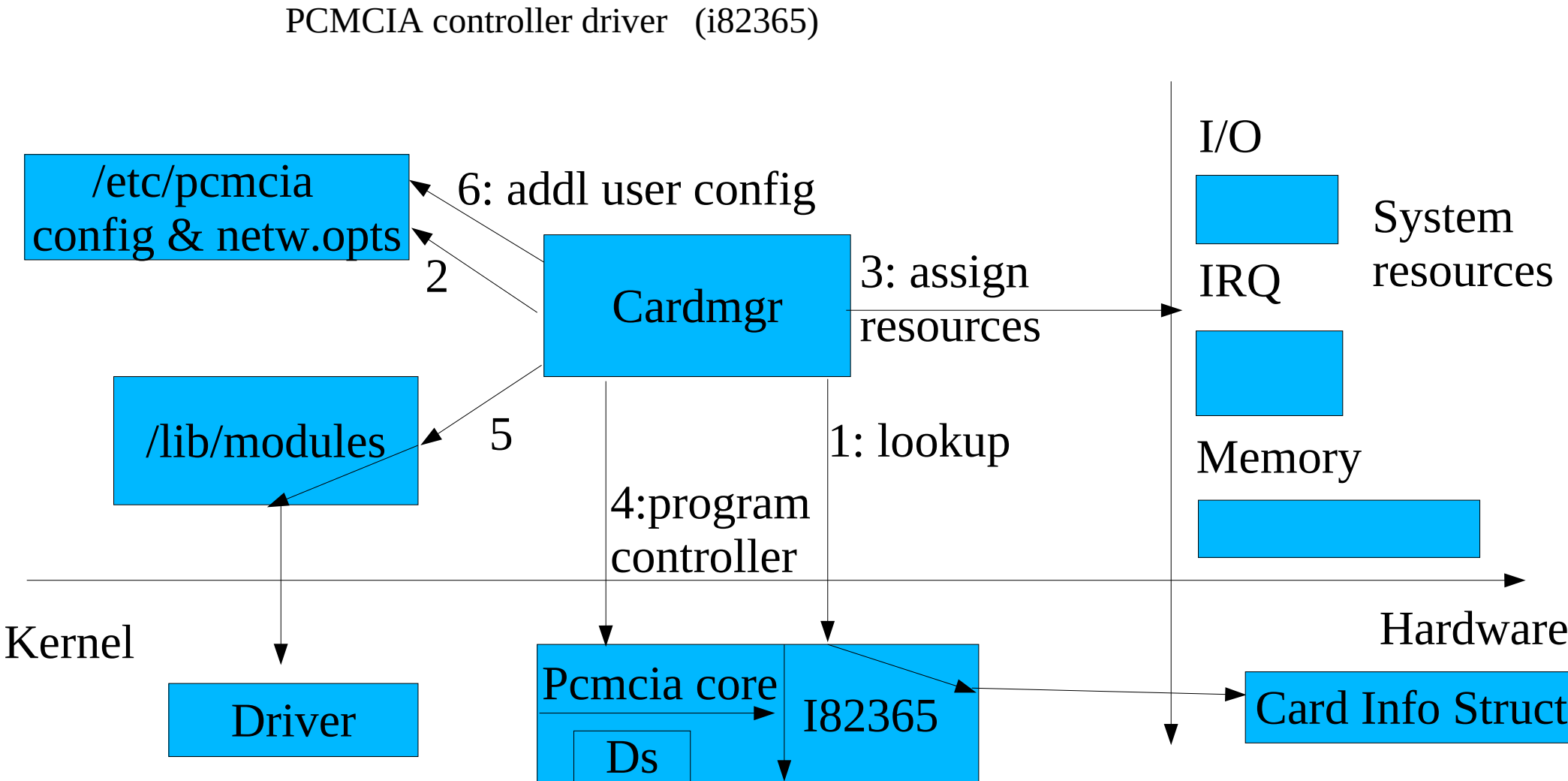


“Core” Kernel

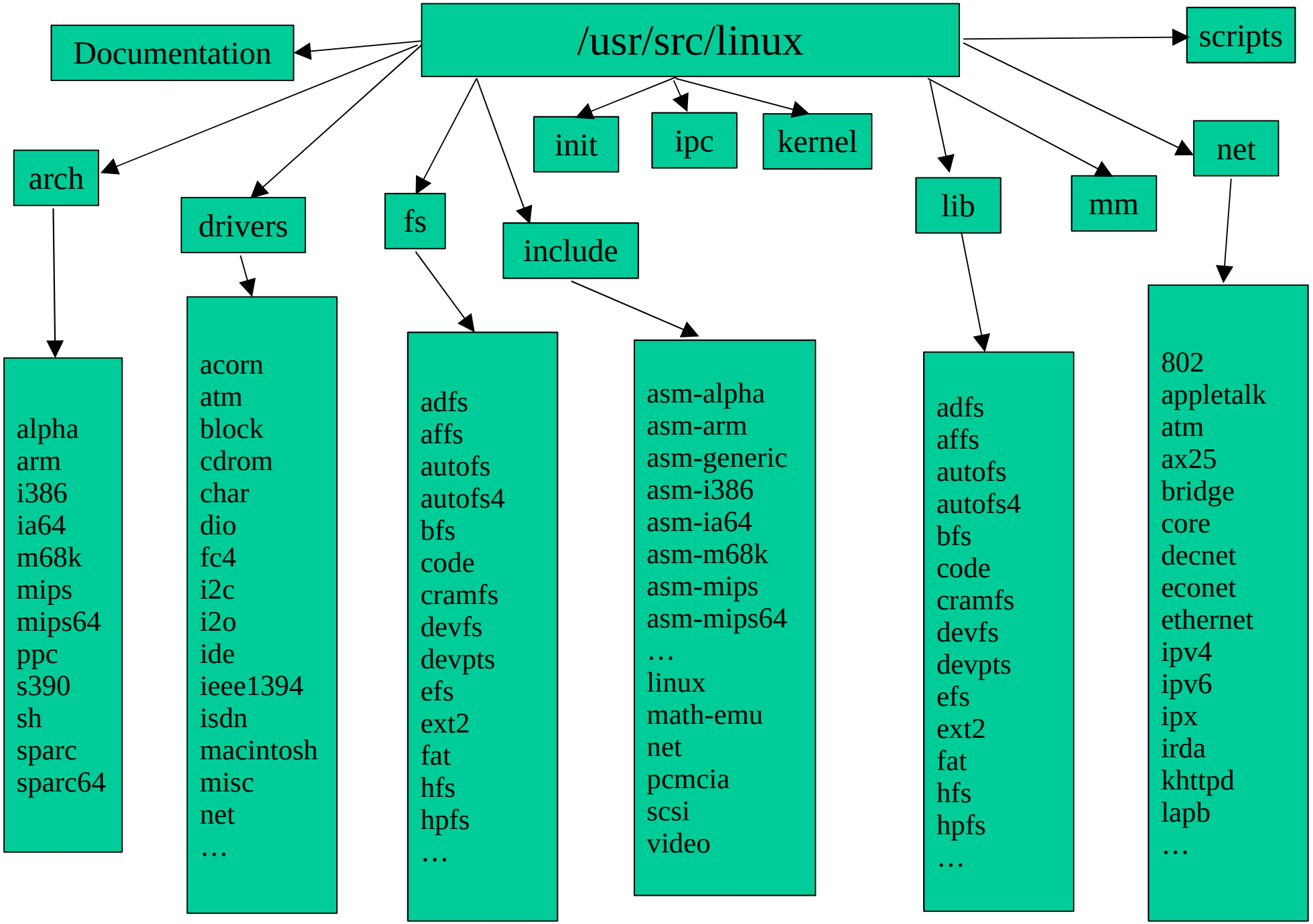

Applications
System Libraries (libc)
Modules
System Call Interface
I/O Related
File Systems
Networking
Device Drivers
Process Related
Scheduler
Memory Management
IPC
Architecture-Dependent Code
Hardware

D-Link 802.11 cards

- most D-Link PCMCIA cards: Prism II (Intersil chipset)
  - D-Link 650H (Symbol chipset)
- most D-Link USB cards (Atmel chipset)
- all 22 Mb/s cards such as the D-Link 650+/520+ (TI chipset)



Source Tree Layout



Sizes (linux-2.4.0)

size	directory	entries	files	loc	2.5.64 size
90M	/usr/src/linux/	19	7645	2.6M	208MB
4.5M	Documentation	97	380	na	14MB
16.5M	arch	12	1685	466K	38MB
54M	drivers	31	2256	1.5M	82MB
5.6M	fs	70	489	150K	18MB
14.2M	include	19	2262	285K	30MB
28K	init	2	2	1K	88KB
120K	ipc	6	6	4.5K	100KB
332K	kernel	25	25	12K	680KB
80K	lib	8	8	2K	412KB
356K	mm	19	19	12K	600KB
5.8M	net	33	453	162K	8.8MB
400K	scripts	26	42	12K	1.0MB
	security				60KB
	sound				12.5MB
	usr				16

Linux-2.5.64: 207768K (208MB!): 38M arch, 360 crypto, 14M Documentation, 82M drivers, 18M fs, 30M include, 88 init, 100 ipc, 680 kernel, 412 lib. 28 Makefile, 600 mm, 8784 net, 1044 scripts, 60 security, 12464 sound, 16 usr

Architectural Approaches

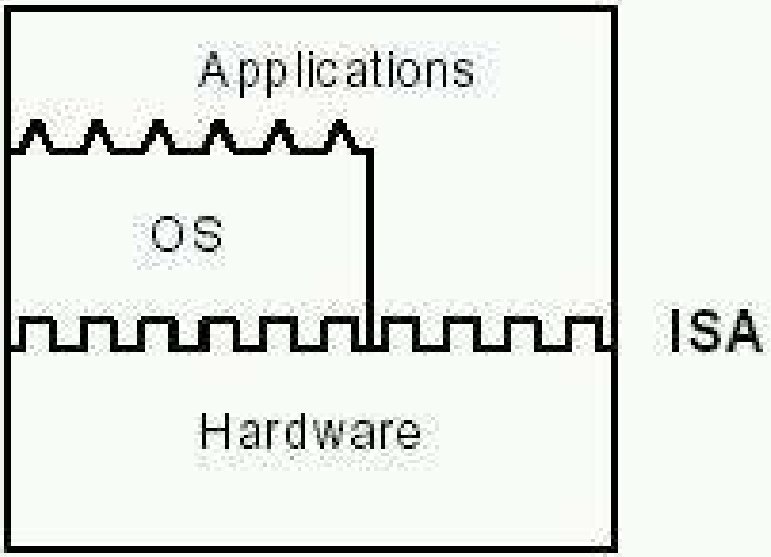
- Single “monolithic” address space
  - All code and data for kernel in same address space (not message passing based)
    - Includes: scheduling, virtual memory management, device drivers, file systems, networking
- Layered
- Modularized
  - Kernel modules can be dynamically loaded/unloaded
- Micro-kernel
- Virtual machine

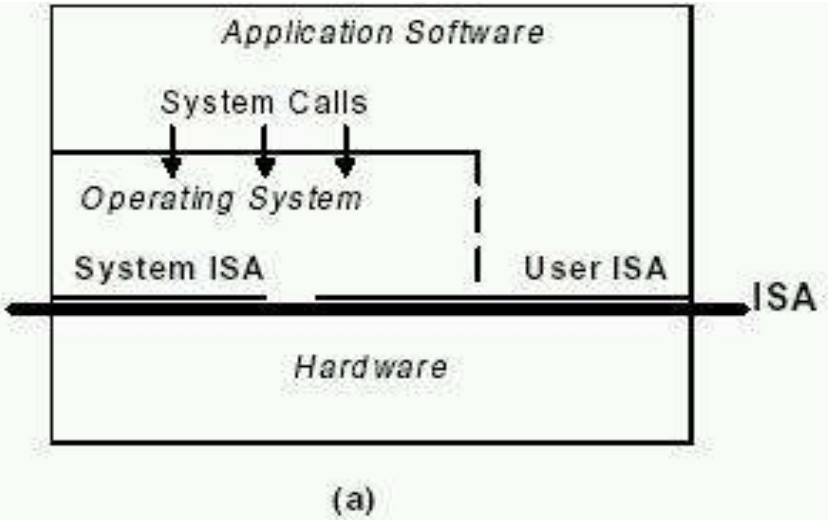
Isolating Hardware Dependencies

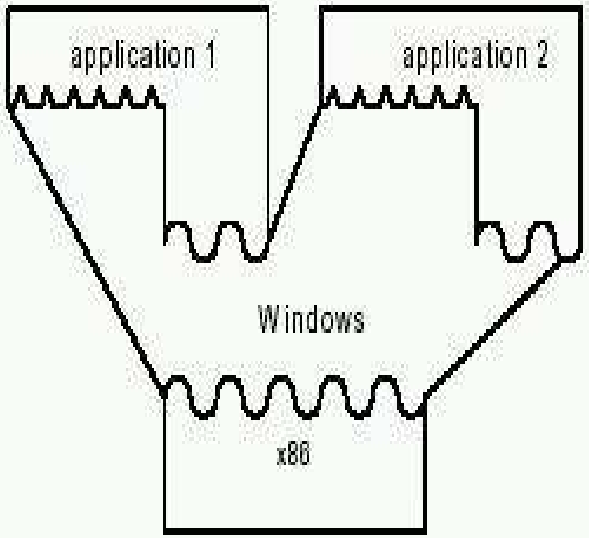
- architecture (cpu)
  - dependent (/arch)
  - independent (everything else)
- abstract dependencies behind functions and macros
- link appropriate version at compile-time
- device-dependencies isolated in device drivers
- provide general abstractions that map to reality
  - e.g. three-level page tables
- tradeoff: exploiting special hardware features

Interplay of Architecture & OS

- Std







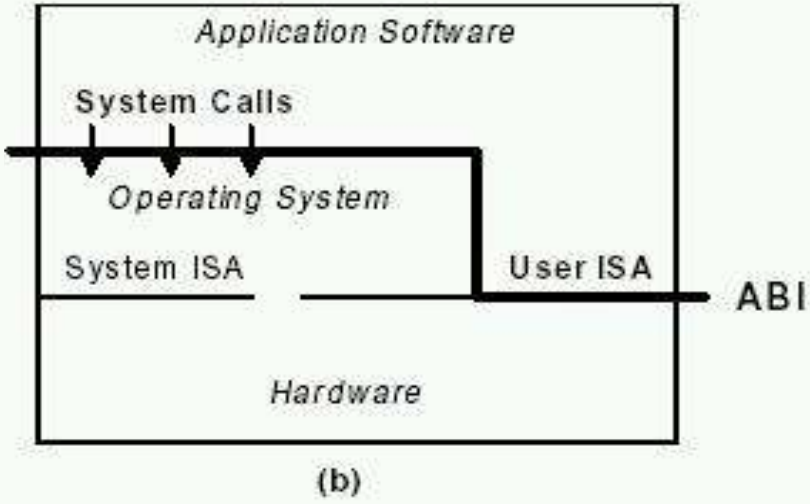
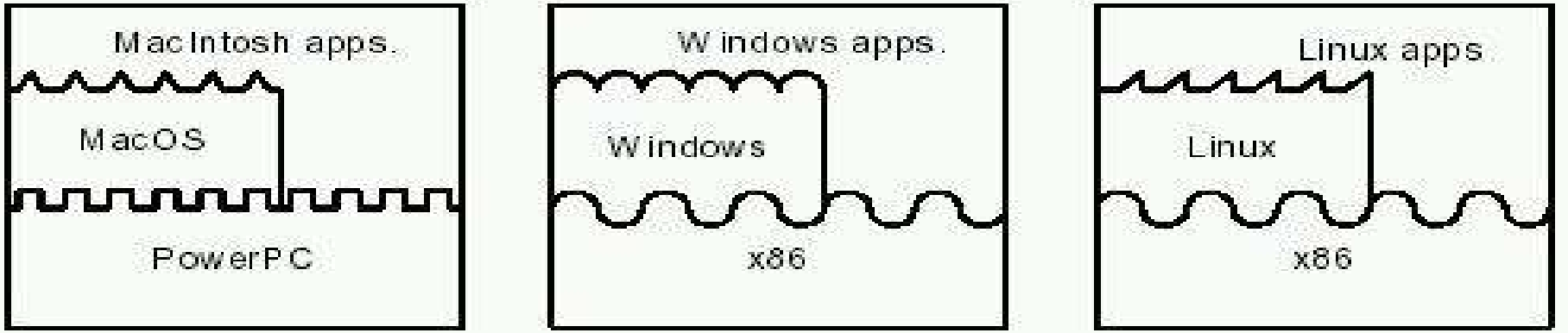


Fig. 5 Important System Interfaces

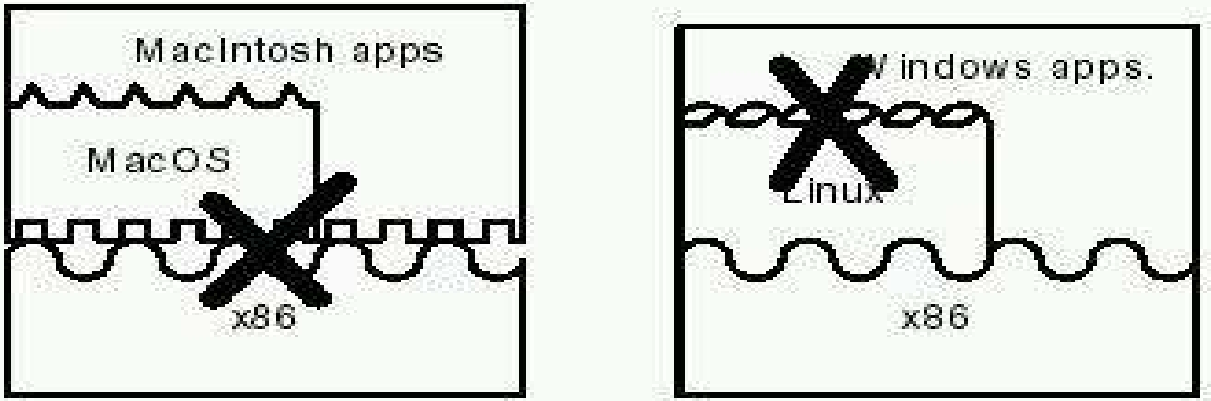
a) Instruction Set Architecture (ISA) interface

b) Application Binary Interface (ABI)

Problem?



(a)



(b)

a) Three popular computer systems composed of different ISAs, OSes, and Applications.
b) Although highly modular, the major system components are not intero-operable.

Virtual Machine

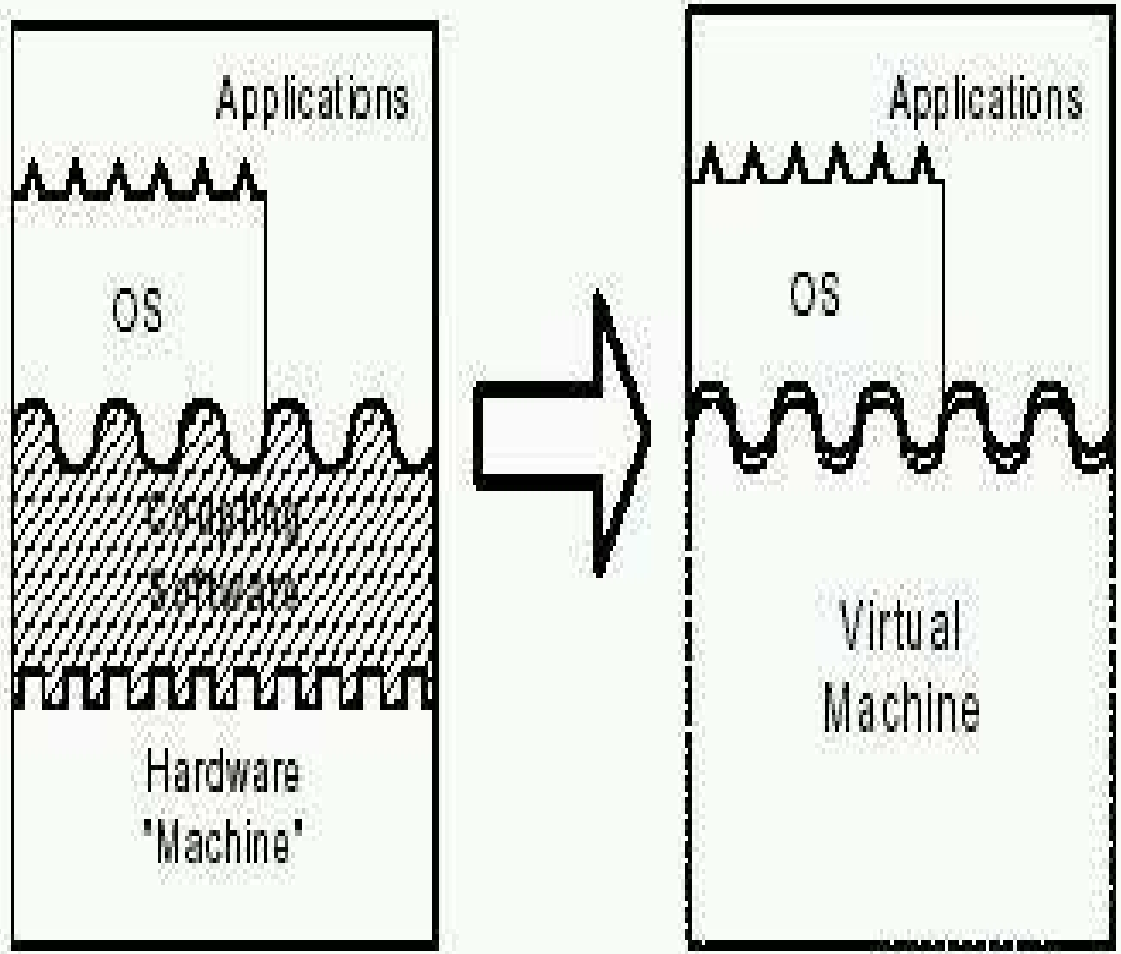


Fig. 3. "Coupling Software" can translate the ISA used by one hardware platform to another, forming a Virtual Machine, capable of executing software developed for a different set of hardware.

Use of VMs

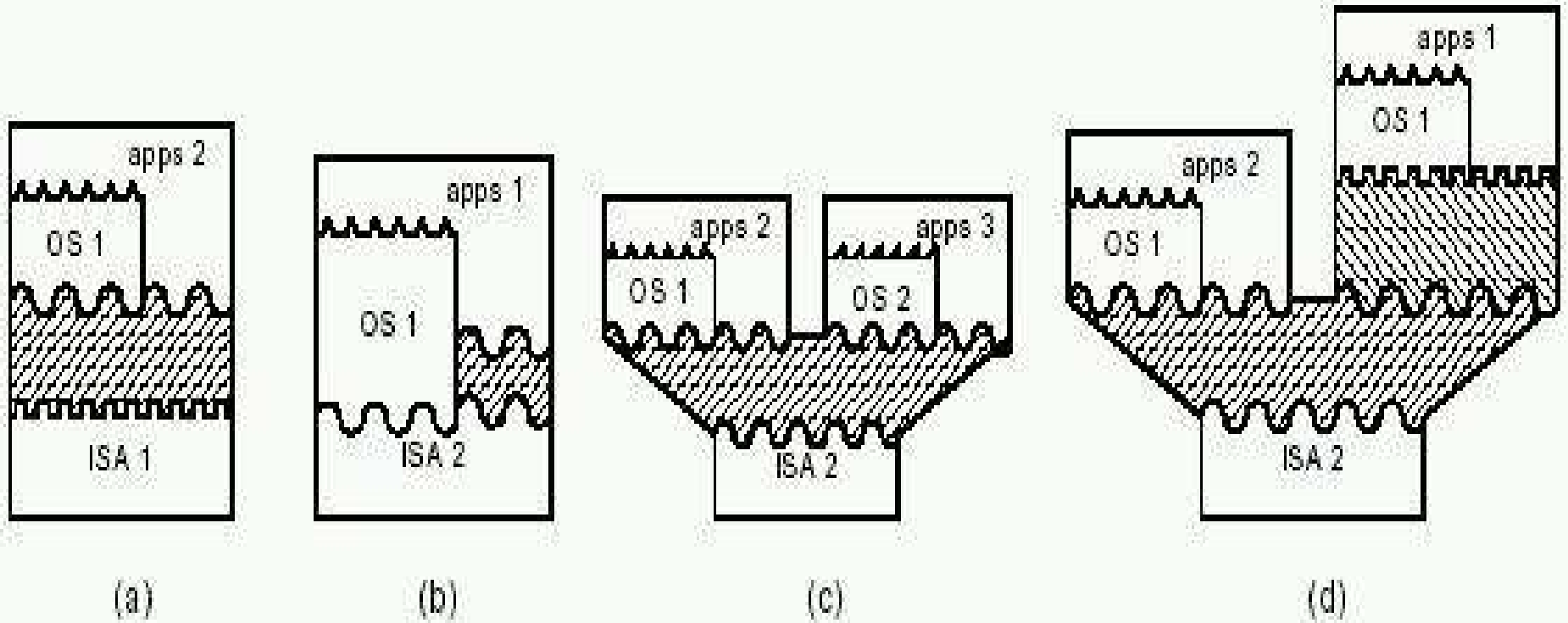


Fig. 4. Examples of virtual machine software being used to connect the major system components.

a) translating from one instruction set to another

b) optimizing an existing application binary for the same instruction set,

c) replicating a virtual machine so that multiple (possibly different) OSes can be supported simultaneously,

d) composing virtual machine software to form a more complex, flexible system.

Example: Linux+Windows on x86

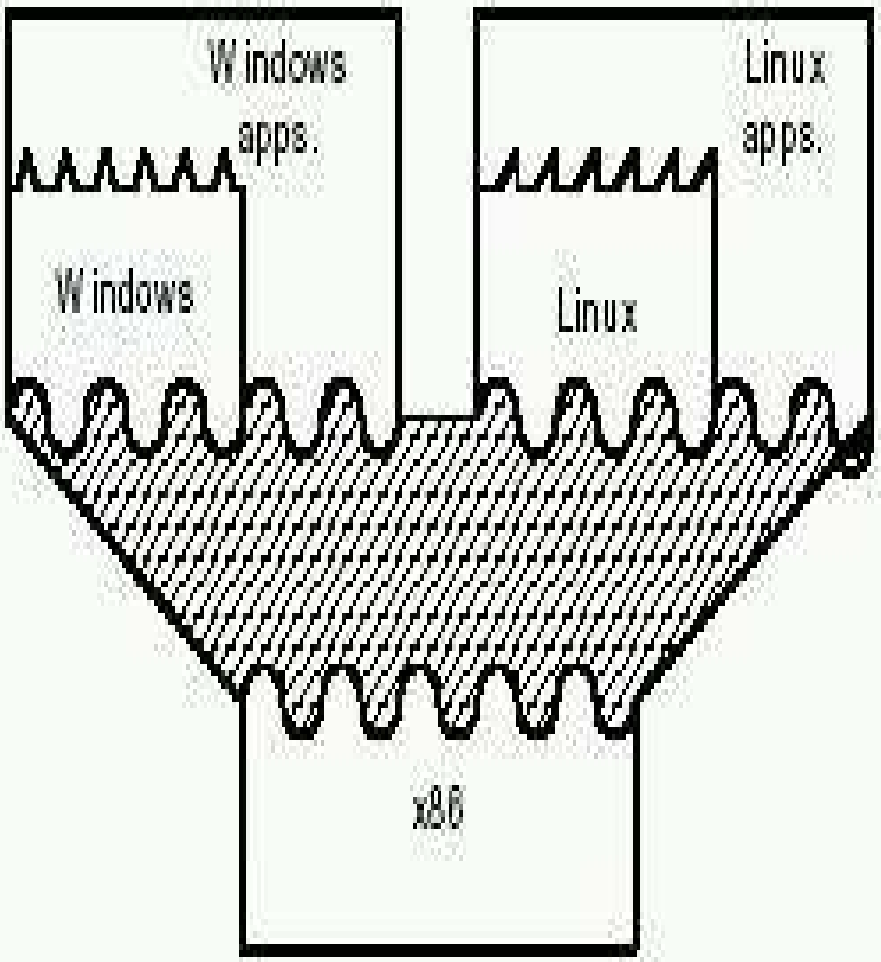
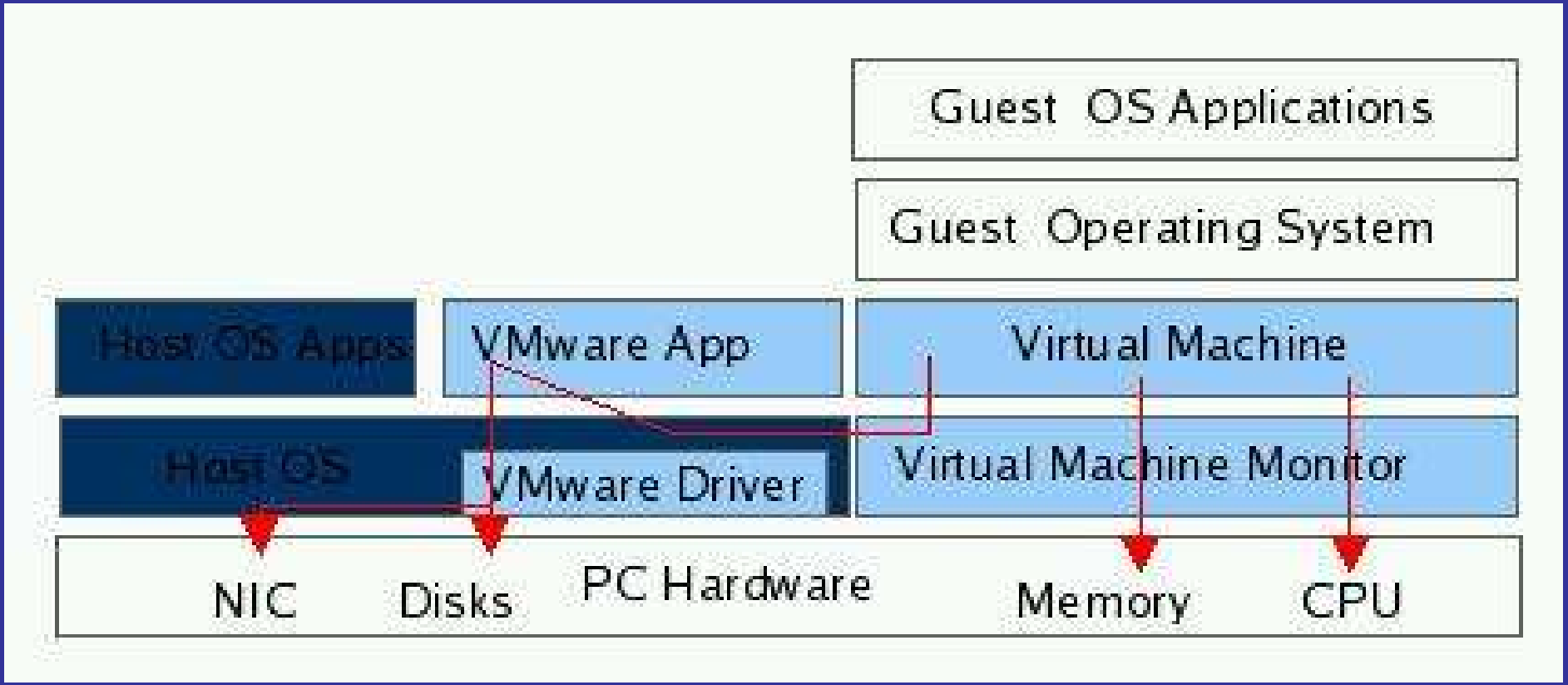


Fig. 7 An example OS VM -- supporting multiple OS environments on the same hardware.

VMWare

- Virtual machines on demand!
- Can construct network of VMs and virtualize almost all HW resources on a PC
- What techniques are used? *(from VMware doc)*



Views of OS

- Resource principle OS: an OS adjudicates use of resources betw users
- Beautification principle OS: process has an enhanced or simplified view of HW
- Virtual machine OS: process has i/f identically same as HW, incl use of privileged inst
  - can ignore protection and multiprogramming issues
  - useful for testing, release of new versions of OS,
  - enhance reliability by isolating diff sw components in diff VMs,
  - enhance security by isolating sensitive data and progs to their own VM,
  - test netw by simulating machine-machine comm thru VMs on 1 phys machine,
  - very simple OS possible (eg: CMS on virtualizing kernel IBM370/VM370: one process and no protection)

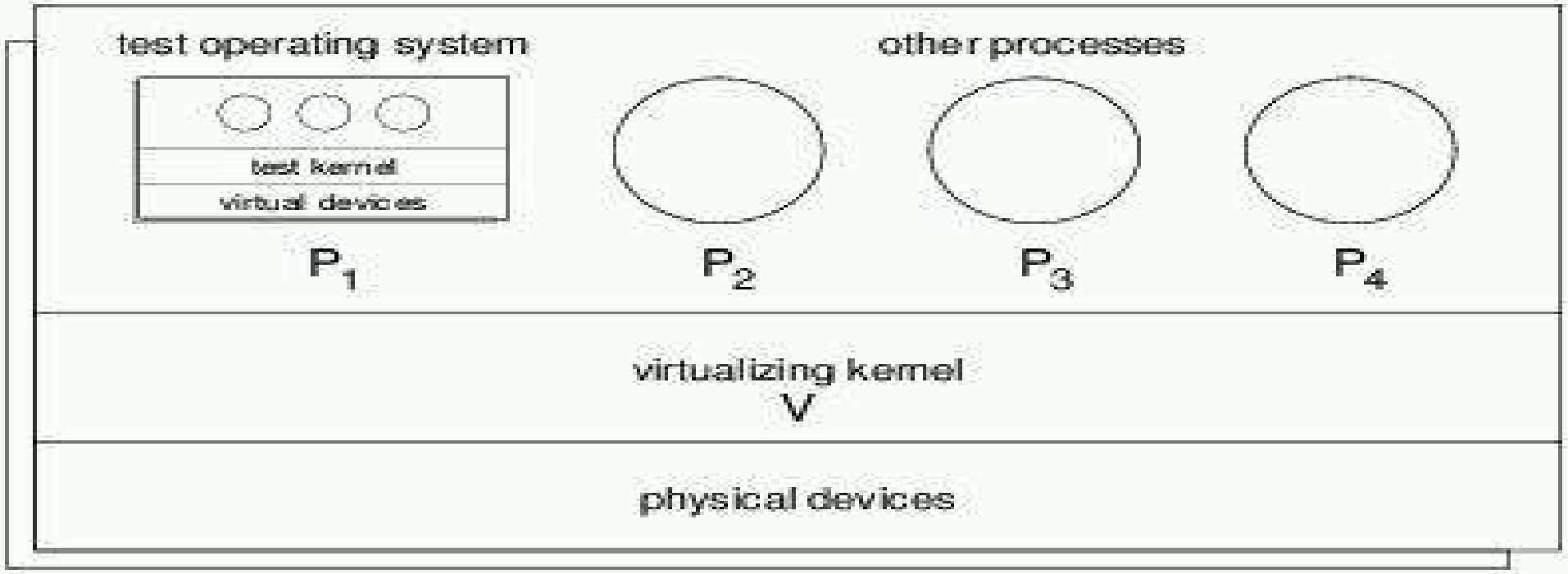


Figure 1.9 Testing a new operating system

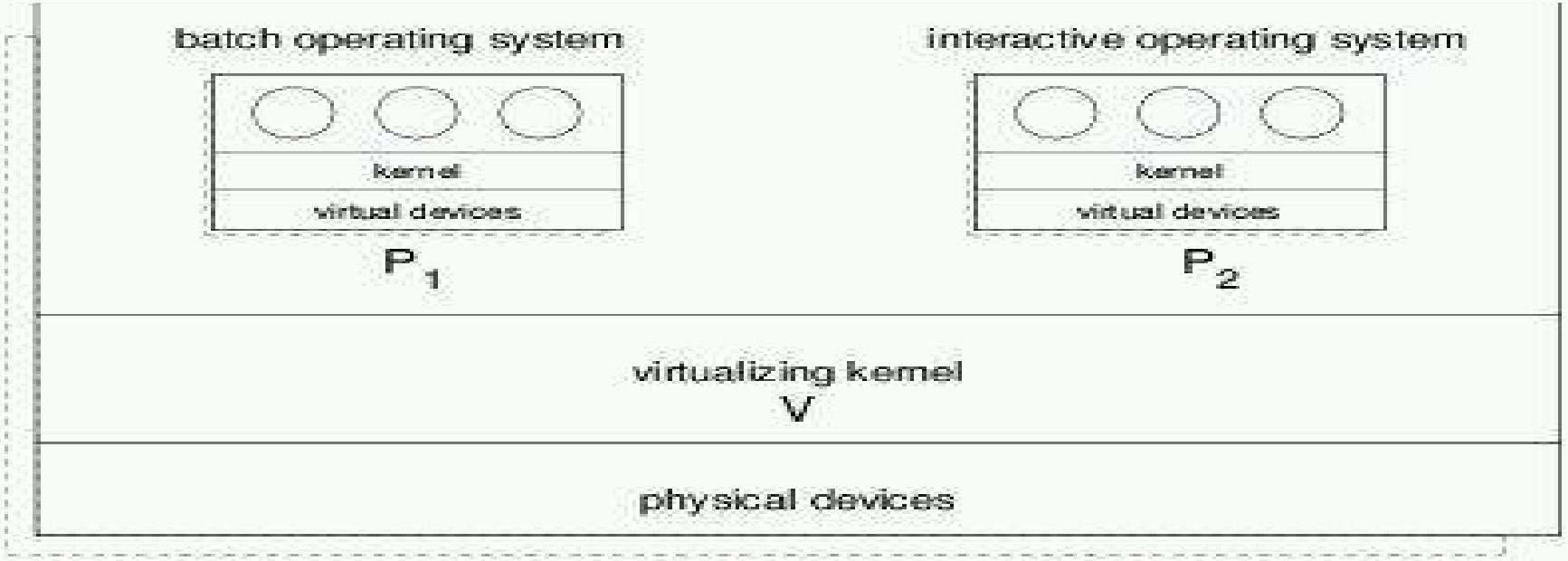


Figure 1.10 Integrating two operating systems

Meanings of Virtual?

- 1st meaning of virtual: provide an exact emulation so that, for eg., a disk looks like a tape
  - SCSI Target Emulation
  - Multiple disks look like a single disk (Striping or RAIDs)
  - Usually close to hw & important for supporting legacy devices
- 2nd meaning: provide an abstraction of a service that does not exist as a service in any subsystem
  - QoS, High Availability, Automatic migration of data across tiers, etc.
  - Usually at higher levels
- Another related meaning: abstract away some problematic aspect in real systems (failures, unregulated access, etc)
- Broadly, emulated vs attribute (or, "API") based storage

VM Operation

- V must run all processes of Pi in non-privileged mode
  - Otherwise: what if a process in Pi executes halt? or other "dangerous" inst like I/O, change addr mapping regs or processor state (interrupt/return, set priority)
  - a machine cannot be virtualized if "dangerous" insts ignored or not trapped in non-privileged mode
- all privileged insts of processes in Pi trapped to V
  - if Pi in virtual non-privileged state, V emulates a trap for Pi
    - puts Pi in virtual privileged state with PC for Pi set to trap addr in Pi

      => V has reflected trap to Pi
    - machine still in physical non-privileged state
  - if Pi in virtual privileged state, V emulates inst directly
    - terminate process if halt inst or interpret I/O inst

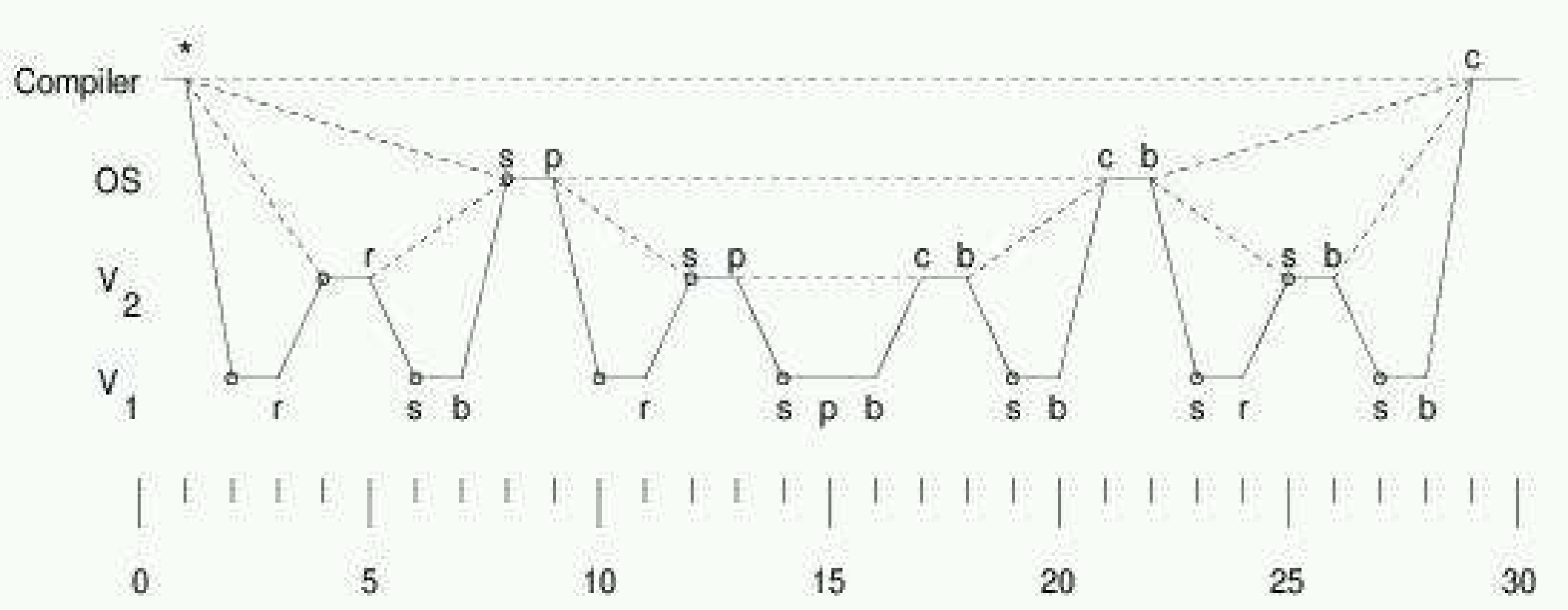


Figure 1.11 Emulating a service call

- c: compiler issues syscall at time 1 and continues at time 29
- p: privileged inst
- r: reflect trap
- s: service trap
- b: mode switch back

- OS: receives trap at time 8 and “executes” privileged inst; mode switch at time 22
- V2: receives 1st trap at time 4 when p executed in virt non-privileged mode. 2nd trap at time 12 when p executed in virt privileged mode. Last trap at time 25 when executing mode switch

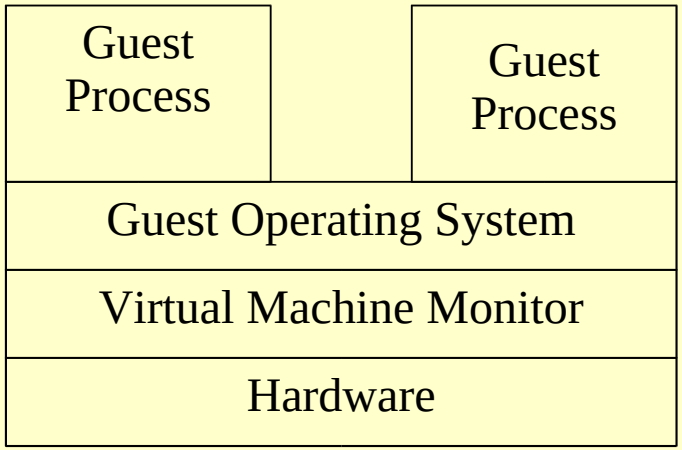
Problems

- difficulties in emulation:
  - I/O: channel programs, esp self-modifying ones; simulating one device by another (printer by disk, a small disk on a larger)
  - a device-completion interrupt for I/O started on behalf of Pi must be reflected to Pi
  - if an I/O involves multiple interrupts, each device-completion interrupt indicates to V to start next phase of emulation: should not be reflected! only last one to Pi
  - mem-mapped I/O: all accesses to these locs must be trapped
  - addr translation
- a good test of a virtualizing kernel: let one of its processes be another virtualizing kernel!
  - but "dangerous" ops become too slow due to many levels

1st Meaning: Some Background

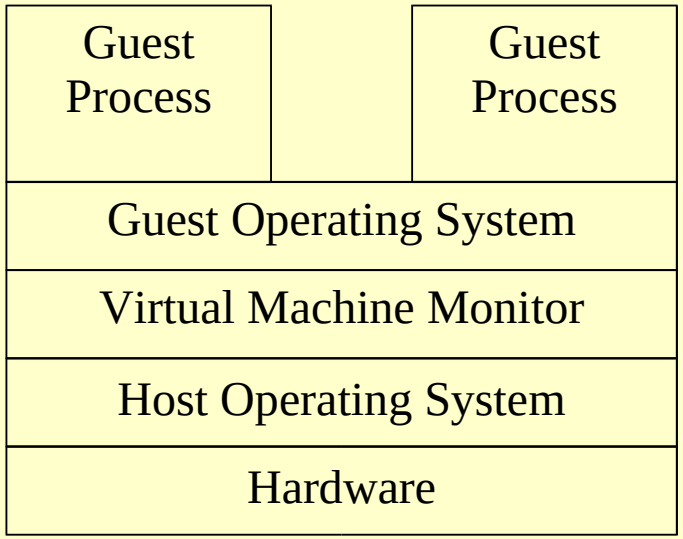
- Architectural Requirements on Processors:
  - VM run in non-privileged mode with VMM in privileged
  - An arch virtualizable only if all insts that *inspect* or modify privileged state trap if executed from any but the most privileged state
  - Some cannot be naturally virtualized:
    - Privileged state writes trapped but not all reads
    - PDP-10; PDP11/45
    - IA-32: SGDT, SIDT, SLDT; SMSW; PUSHF, POPF; LAR, LSL, VERR, VERW; POP, PUSH; CALL, JMP, INT n, RET; STR
- In the storage subsystem: transactional or similar models needed to provide isolation
  - But how is *durability* to be handled?

Virtual Machine Configurations



Type I VMM

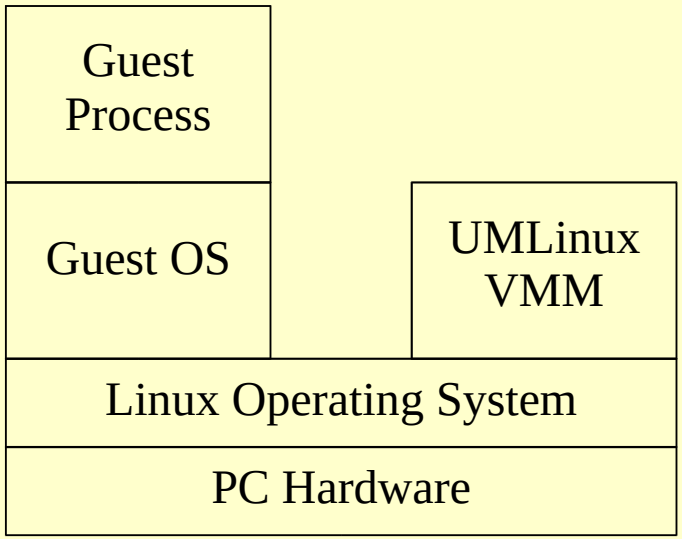
- Runs directly on hardware
- Good performance



Type II VMM

- Uses existing host OS abstractions to implement services
- Poor performance

UMLinux Architecture

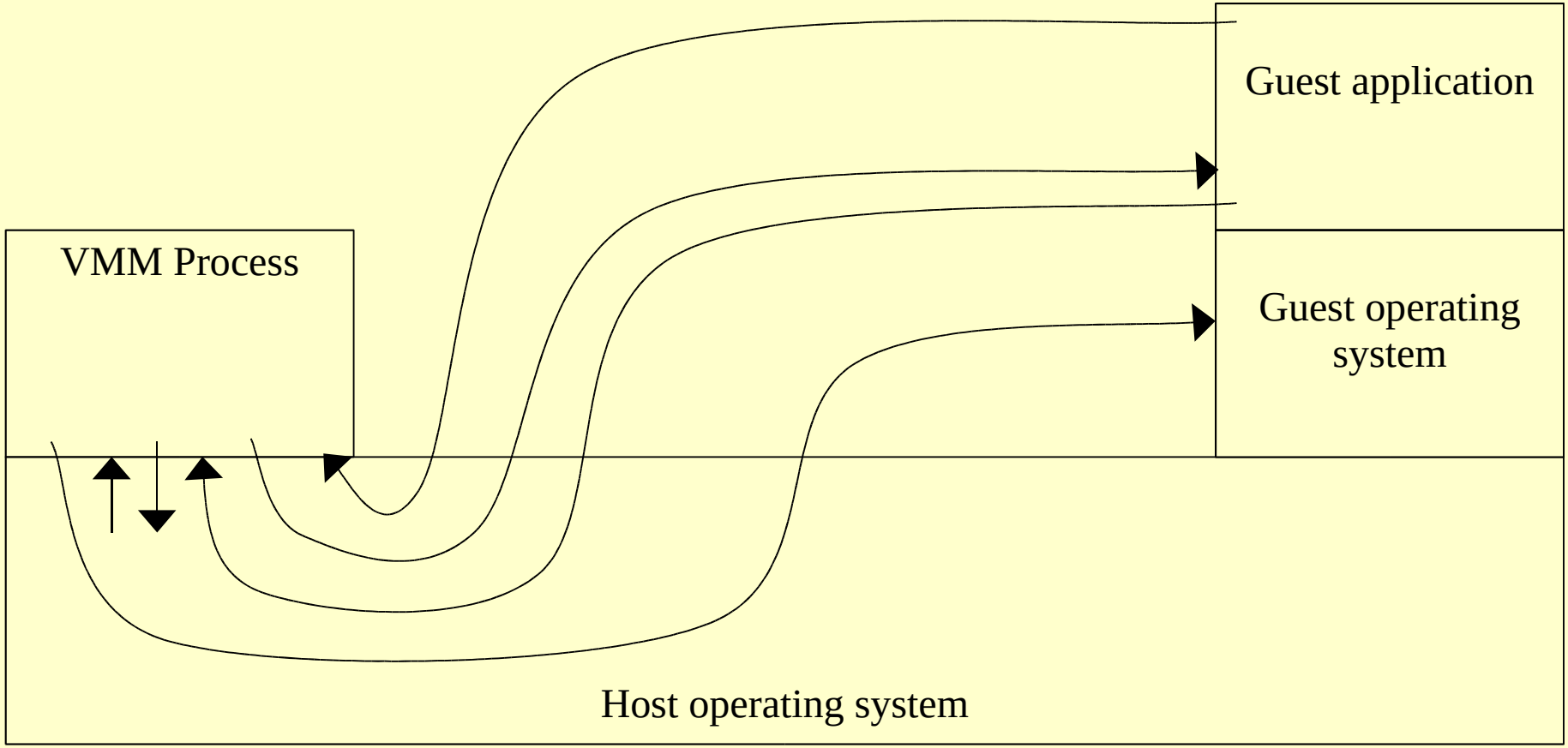


- Linux on top of Linux
- Port of Linux to run in UMLinux
- *ptrace* used for virtualization
  - intercept guest system calls
  - track guest user / kernel mode transitions

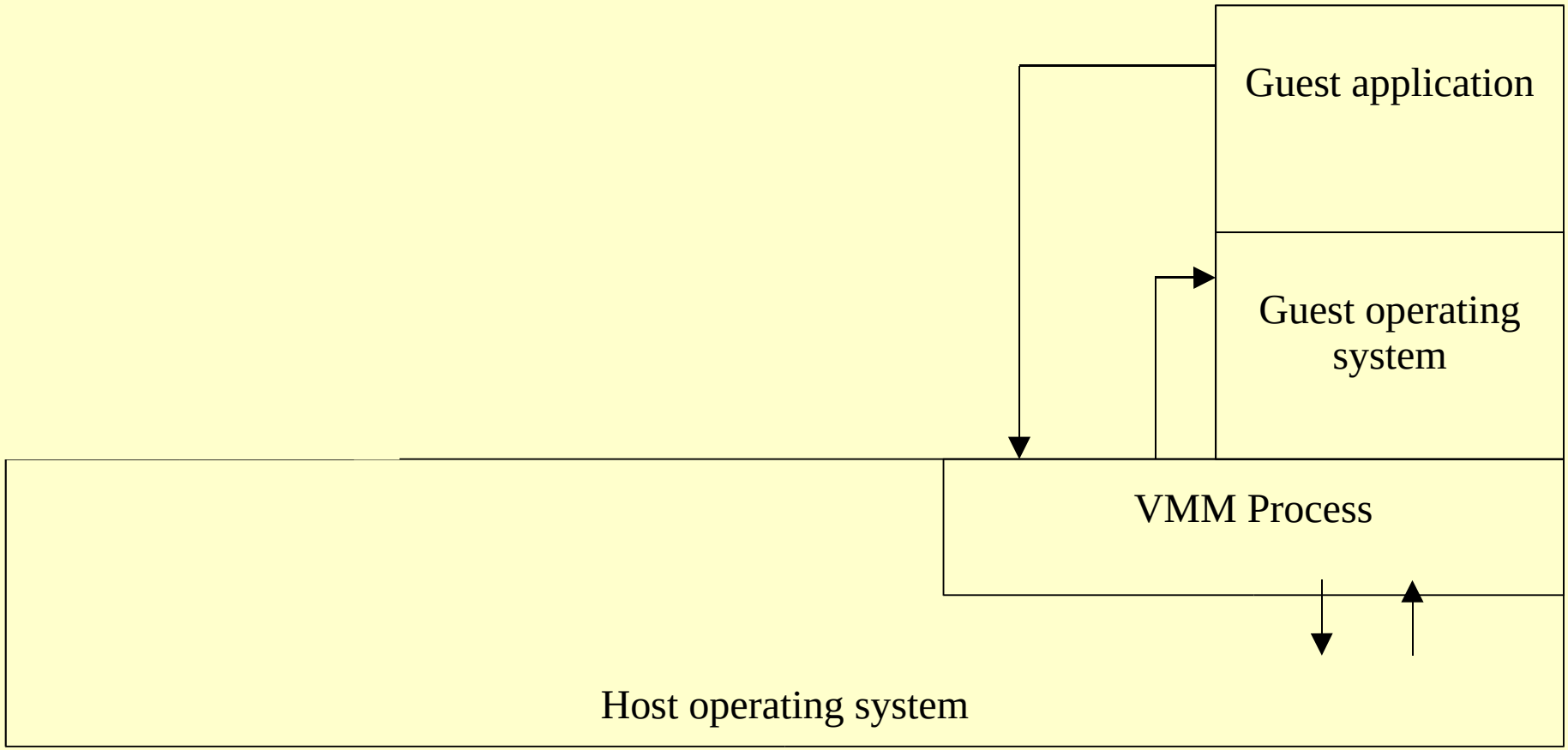
Hardware Equivalents

Hard Disk	Host /dev/hda raw partition
CD Rom	Host /dev/cdrom
Floppy	Host /dev/floppy
Network Card	Host TUN/TAP interface
Physical Memory	Host file accessed via *mmap*
MMU	Host *mmap*, *mprotect*
System Call (int 80)	Host SIGUSR1 signal
Timer Interrupt	Host SIGALRM signal
I/O Interrupt	Host SIGIO signal
Memory Exception	Host SIGSEGV signal

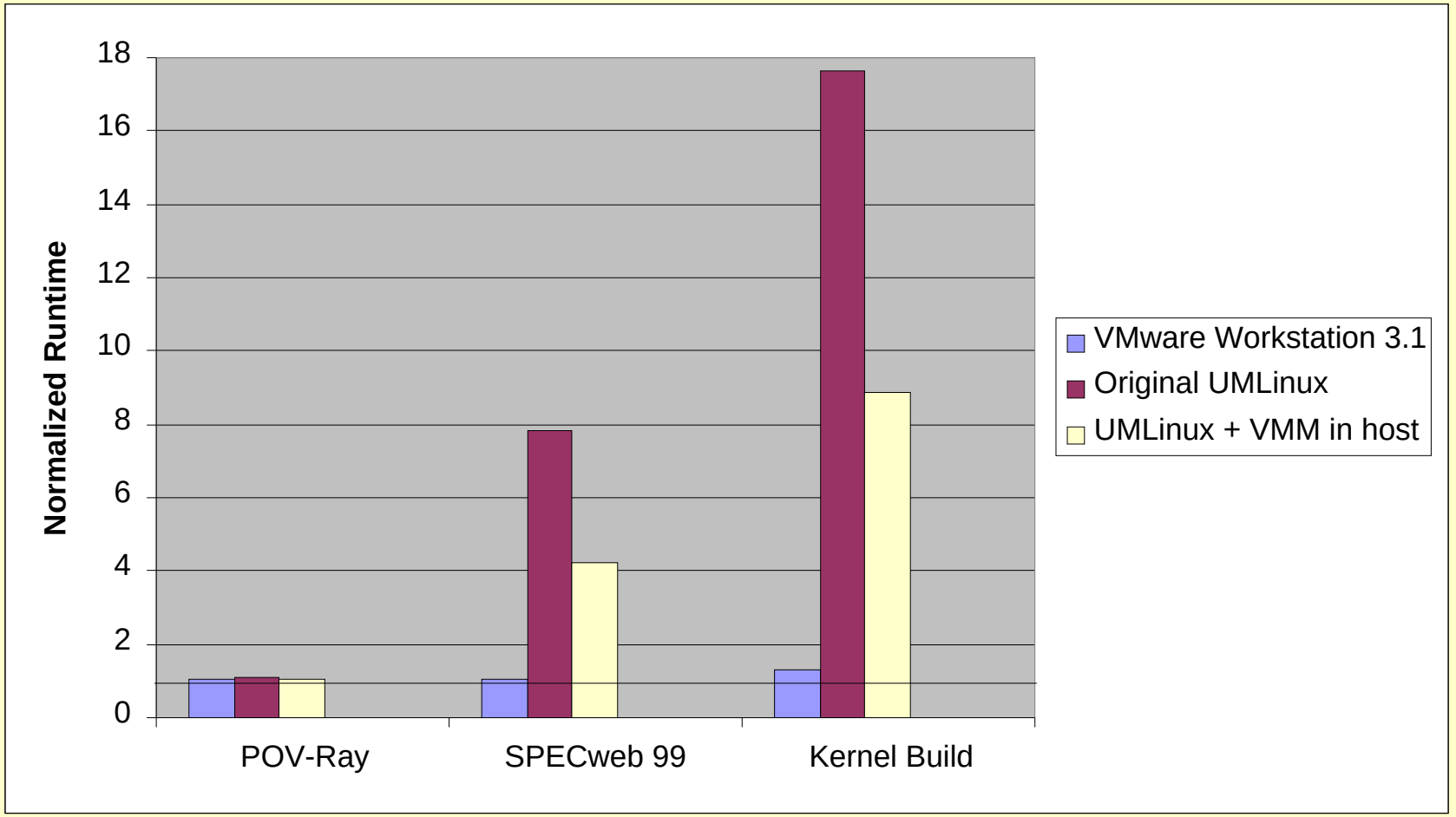
Original UMLinux Guest System Call



Move VMM to Host Kernel



Performance



Virtualization in Current Systems

Memory: paging/segmentation

- System should *(ideally!)* run correctly independent of available physical mem though perf ormance may be different with avlbl mem

CPU: processes, threads, virtual machines

- Each process/thread/VM should *(ideally!)* run correctly indep of others
- IBM 370/CMS+370/VM vs 360/TSS
- ***Note significant and deep interactions betw mem and cpu virtualization!***

Network

- User level network interfaces: Virtualize a netw card (VIA/Infiniband)
- Kernel based: eg: TCP streams: kernel virtualizes network for 1 stream and isolates it from other contending streams

Virtualization in Current Systems

Storage

- in mainframe environments (IBM)
- simpler or in isolation or in low level forms in
  - File systems: logical mapping to physical mapping
    - Online resizing, defragmentation, NFS, AFS, HSM
  - volume managers (VxVM, LVM)
    - Software-based RAID1, RAID5, hot sparing
  - Controller/HBA level: hardware RAID
  - SCSI level: bad block remapping
- Typically kernel "virtualizes" I/O streams & isolates them
  - User sees simpler interface (no concurrency or sharing)
- Virtualization difficult system wide without stds such as Bluefin
  - With Bluefin, virtualization to the fore!

Tape Virtualization

Tape media virtualization: use disk to emulate diff tape access methods, etc

- Reduce underutilized tape media in mainframe systems typ
- 1st meaning: emulation core part: decouples appl from technologies

Tape drive virtualization: share (costly/high perf) tape drives betw servers

- Very relevant in open systems using SANs (as FC gives connectivity)

Library virtualization: includes storage consolidation, fail-over, mirroring/striping

- 2nd meaning
- Usually through standardised interfaces

ABI VM

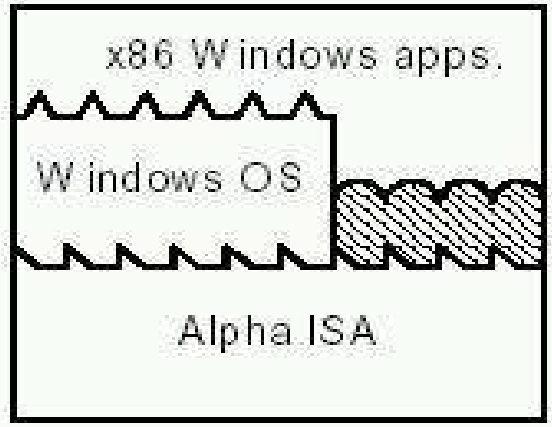


Fig. 8. An ABI VM with emulation/translation of guest applications. The Digital/Compaq FX!32 system allows Windows x86 applications to be run on an Alpha Windows platform.

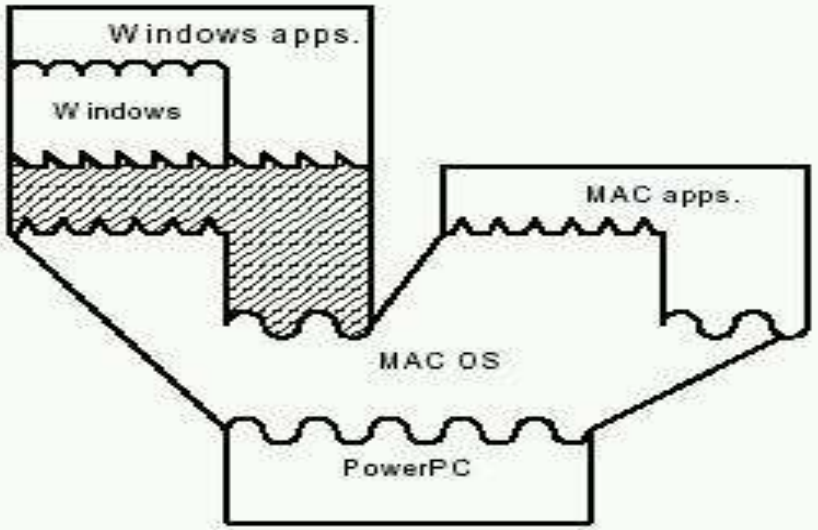


Fig. 9. A whole system VM that supports a guest OS and applications, in addition to host applications.

HLL

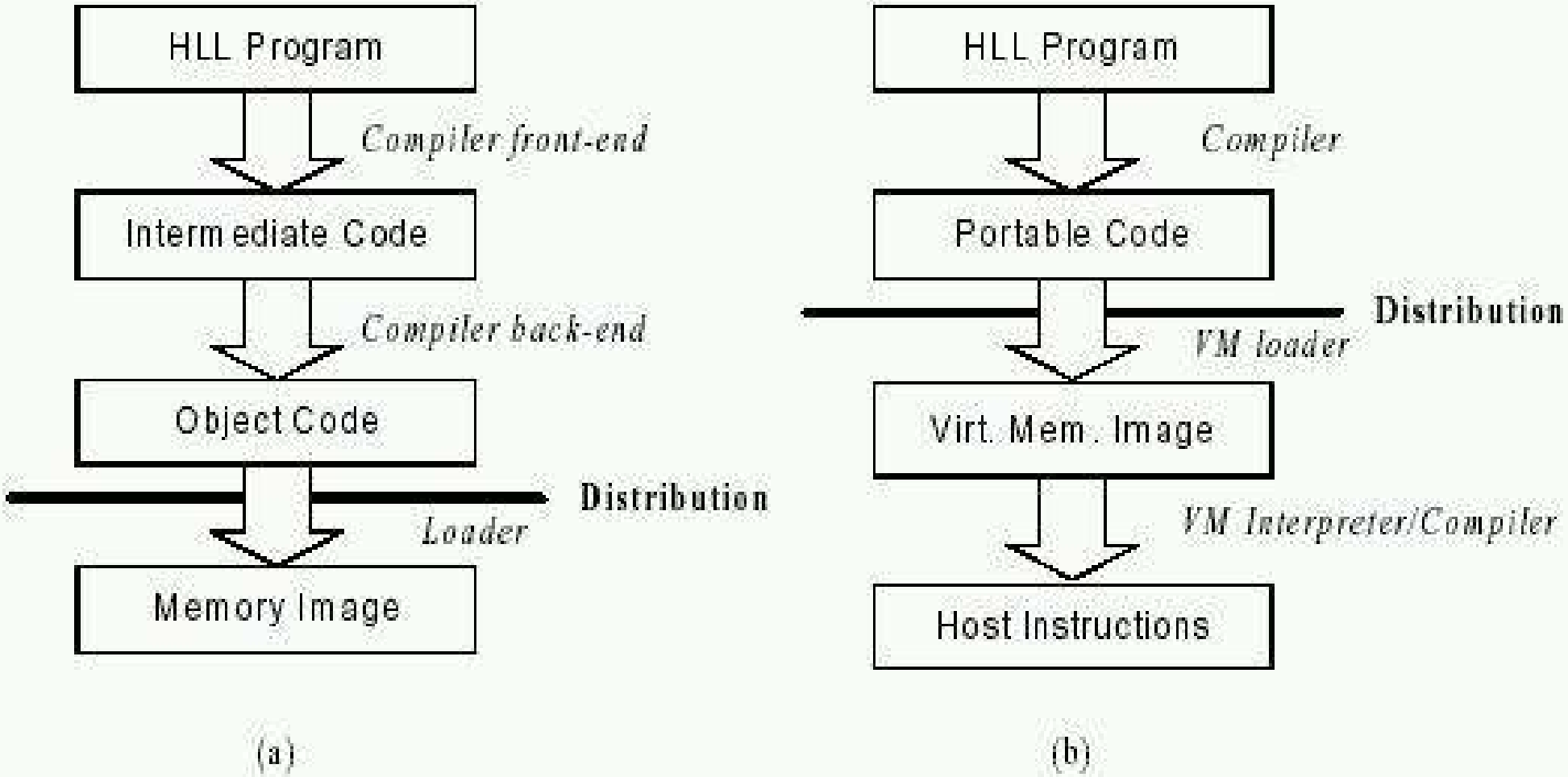


Fig. 10. High Level Language environments

a) A conventional system where platform-dependent object code is distributed

b) A HLL VM environment where portable intermediate code is "executed" by a platform-dependent virtual machine

Co-Designed VM

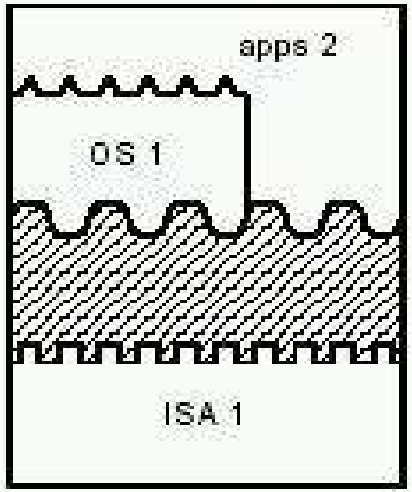


Fig. 11. Co-Designed VM -- VM software translates and executes code for a hidden native ISA.

Fig. 13. Three levels of VMs: a Java application running on a Java VM, running on an OS VM, running on a co-designed VM.

Taxonomy of VM Architectures

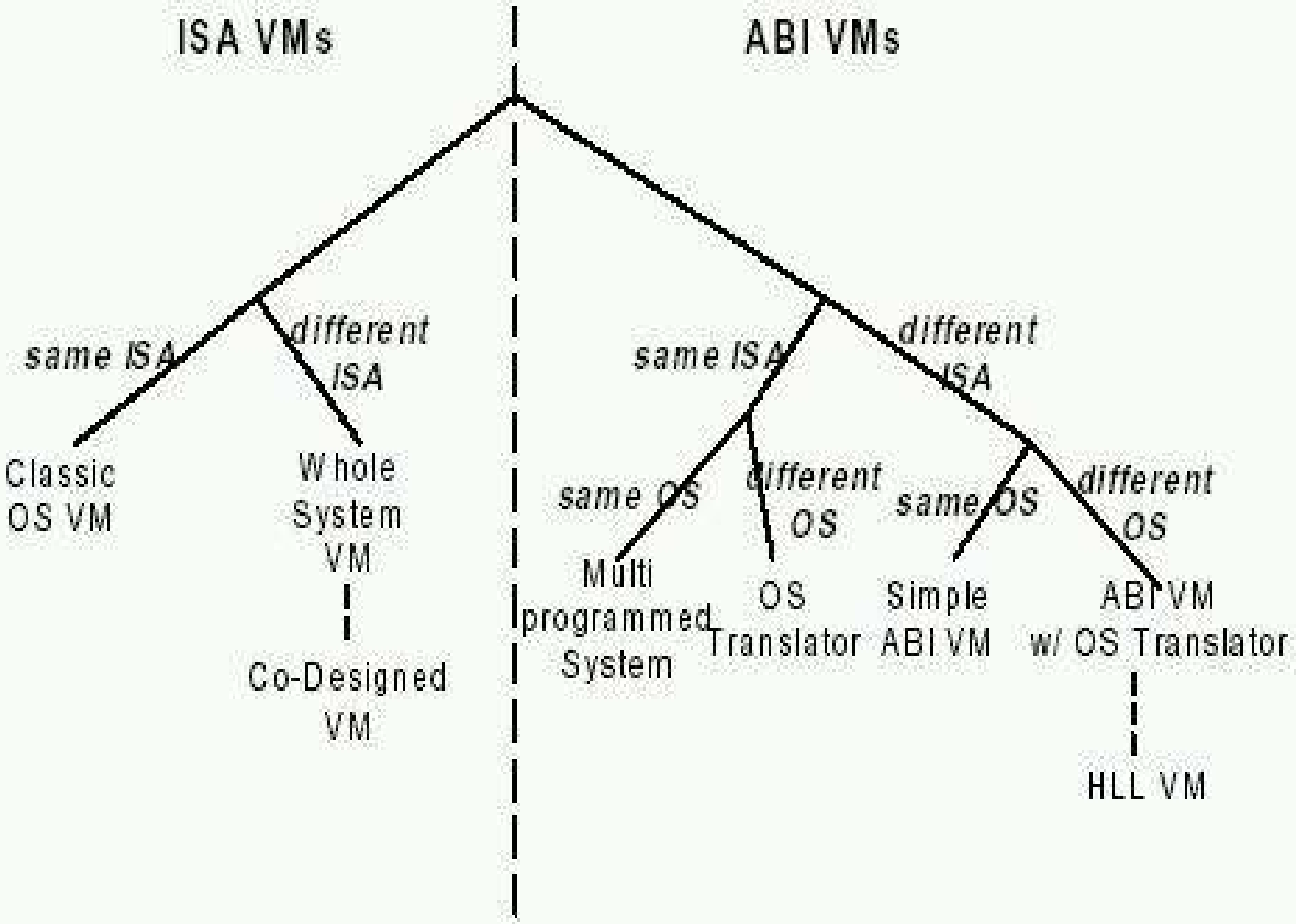
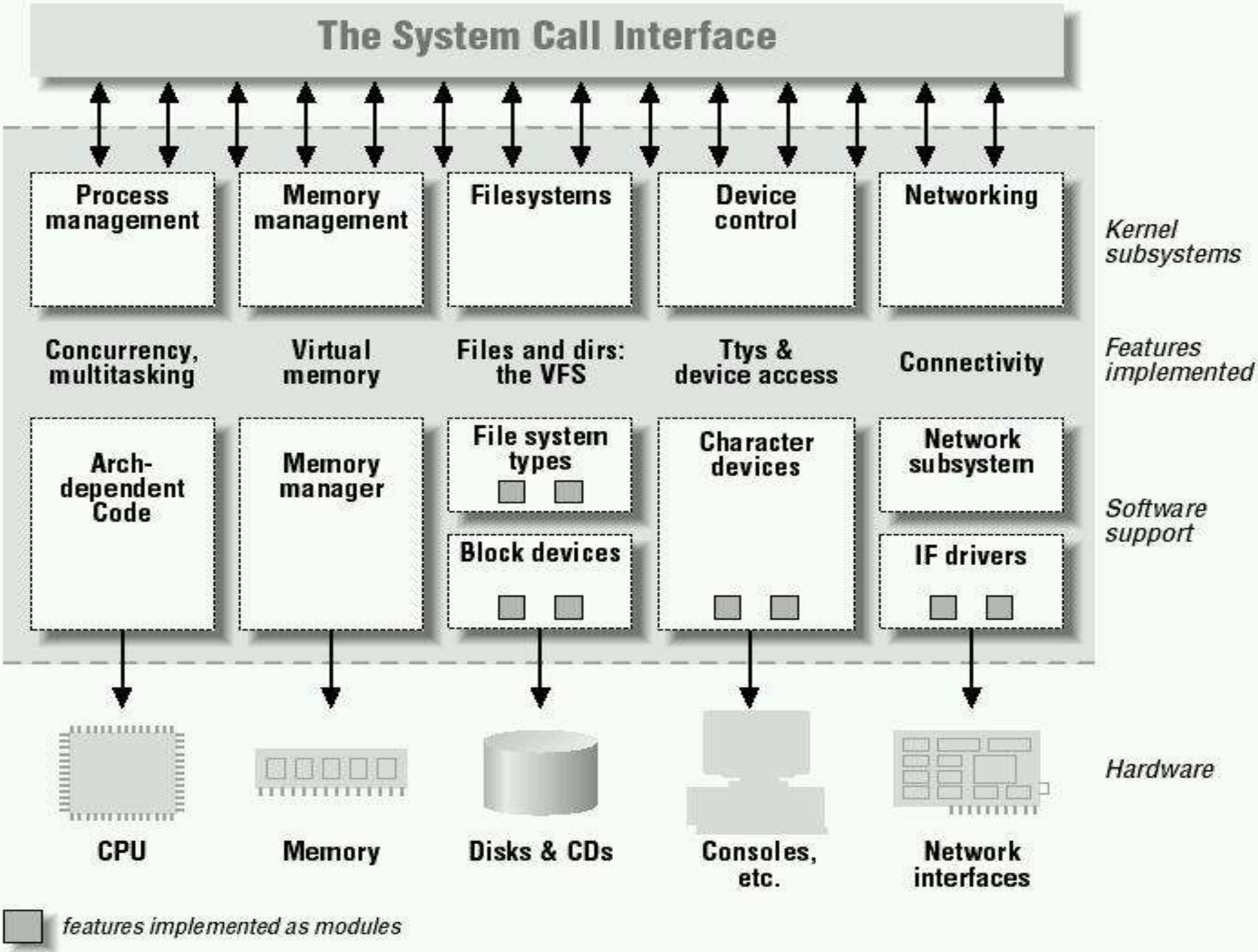
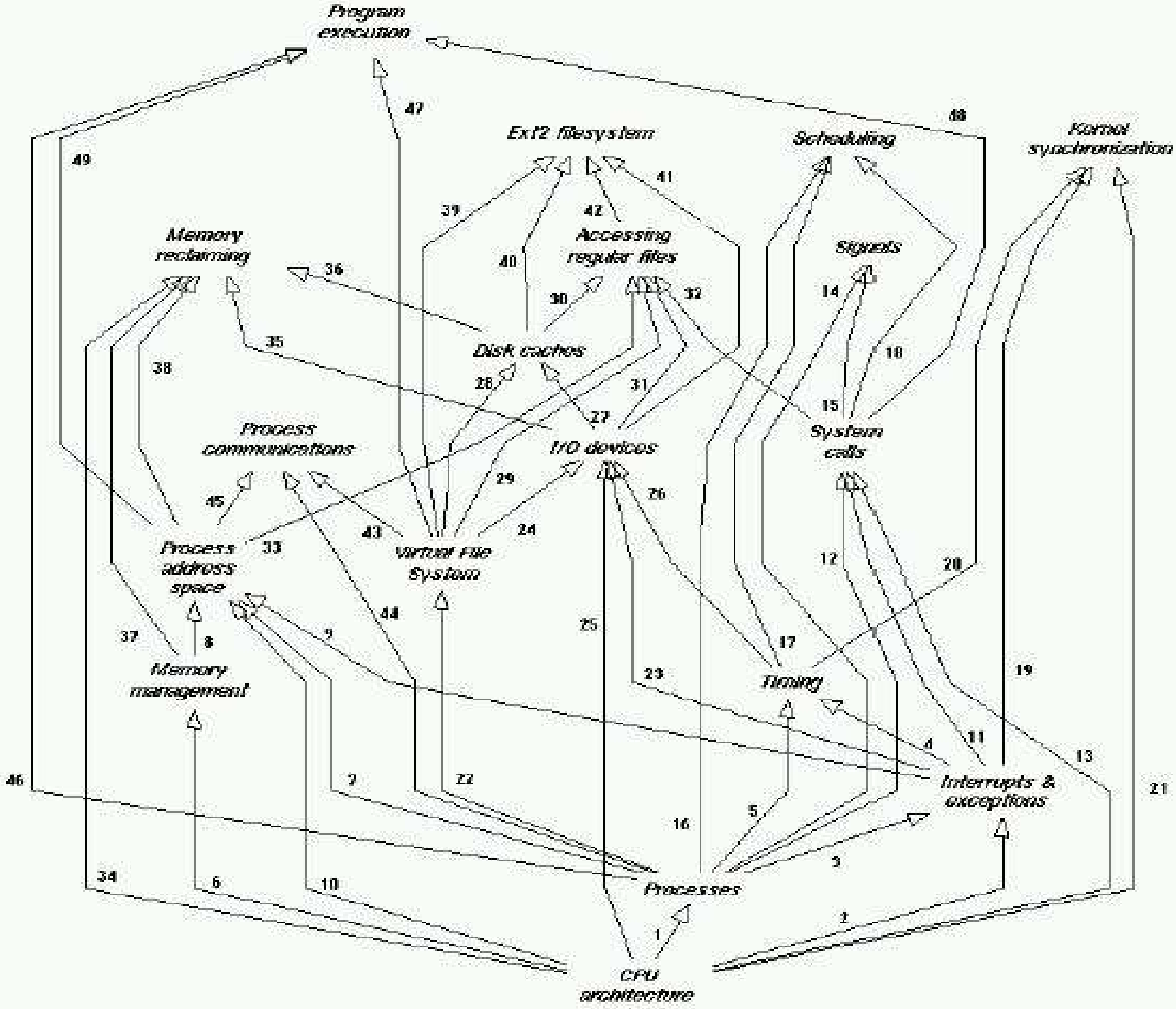


Fig. 12. A taxonomy of virtual machine architectures.


The System Call Interface
Process management
Memory management
Filesystems
Device control
Networking
Kernel subsystems
Concurrency, multitasking
Virtual memory
Files and dirs: the VFS
Ttys & device access
Connectivity
Features implemented
Arch-dependent Code
Memory manager
File system types
Block devices
Character devices
Network subsystem
IF drivers
Software support
CPU
Memory
Disks & CDs
Consoles, etc.
Network interfaces
Hardware
features implemented as modules


Program execution
Ext2 filesystem
Scheduling
Kernel synchronization
Memory reclaiming
Accessing regular files
Signals
Disk caches
I/O devices
System calls
Process communications
Process address space
Virtual File System
Memory management
Timing
Interrupts & exceptions
Processes
CPU architecture

Interrupt & Interrupt handling:

- When a hardware interrupt occurs the CPU stops executing the instructions that it was executing and jumps to a location in memory that contains the interrupt handling code.

- Mask register controls interrupts. In x86, writing a one to a particular bit of the mask register enables an interrupt, zero disables it. For example if we write one to bit 3, we would enable interrupt 3.

- Linux routes the interrupts to the right pieces of interrupt handling code. Many of these codes belong to device drivers, so Linux uses a set of pointers to data structures containing the address of the routines that handle the interrupts. (`irq_action` is a vector of pointers to the `irqaction` data structure)

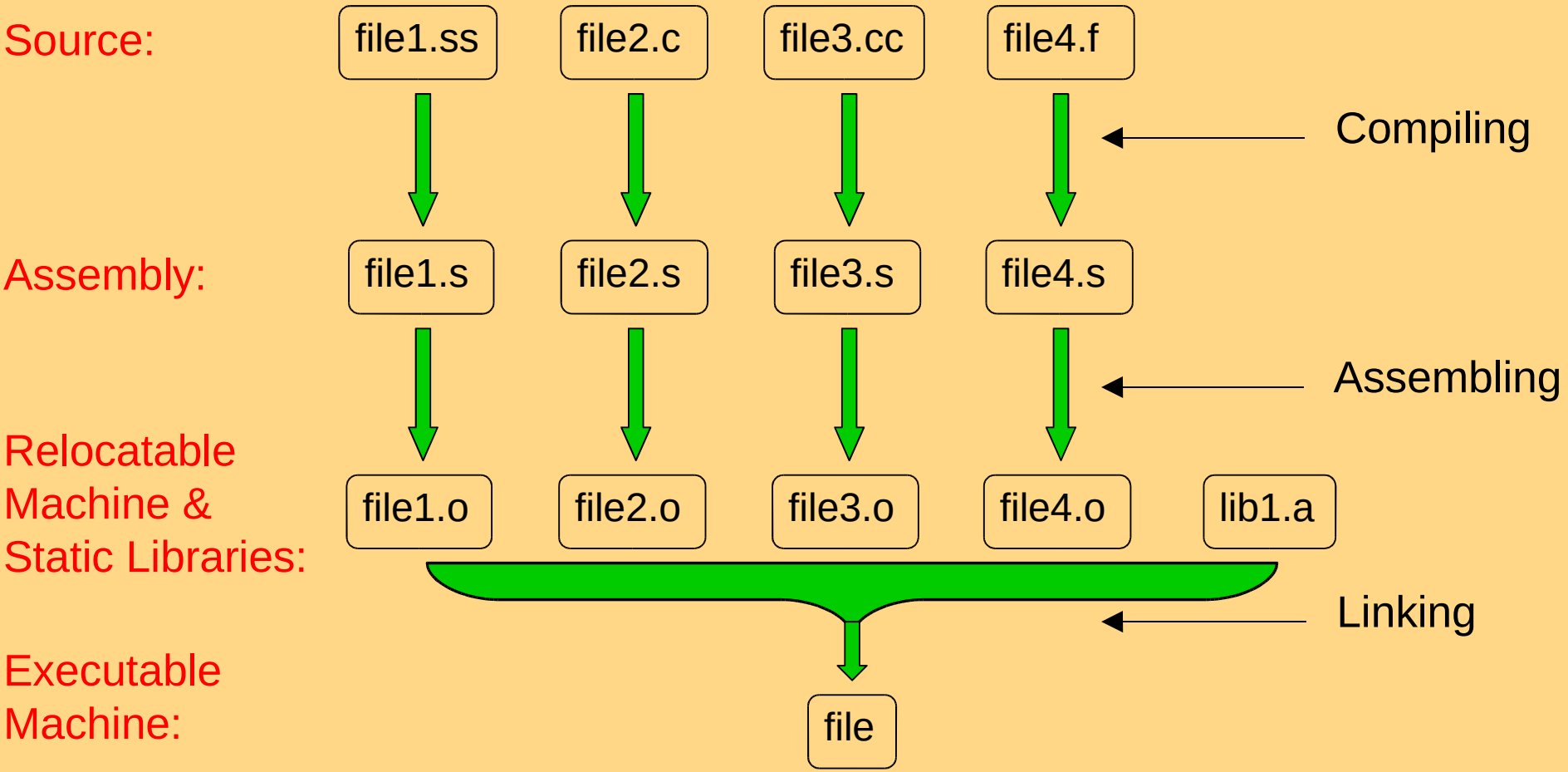
How System Calls Work on x86:

- 0x80 software interrupt is used for implementing system calls. This interrupt is caused by `int 0x80` instruction. In user mode, only the 0x80 interrupt can be called.

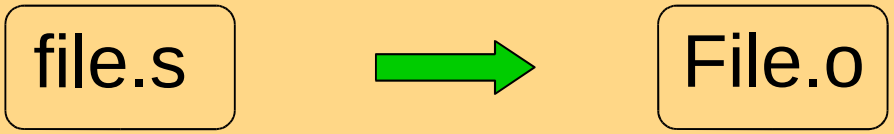
- When a user space application makes a system call, all of the arguments passed via registers and the application executes `"int 0x80"` instruction. This cause a trap into kernel mode and processor jumps to system call entry point.

- After the system call has executed, `_ret_from_sys_call()` is called. It checks to see if the scheduler should be run, and if so, calls it.

Compilation Basics



What is assembling?



- Parse & encode instructions.
- Build tables of label definitions & uses.
- Output relocatable file.

How to Assemble

For instructions, synthetics, & most pseudo-ops not using labels:

- Convert to appropriate word(s).

  Fairly easy – syntactic, not semantic.

- Check alignment of each item.

  Sufficient to know address… Can't, because of relocation!

  Sufficient to know offset… Since section starts aligned.

How to Assemble

`.align` *`N`*

Ensure following item is N-byte aligned.

I.e., skip bytes based on current offset.

`.section` *`“segmentname”`*

Following info is in the named segment.

Need to keep track of each segment’s contents separately.

How to Assemble

Labels stand for the memory address of their definition.

- Computing addresses difficult.
  - Forward references.
  - References to other sections.
- Computing all addresses impossible.
  - References to other files.
  - Only possible during linking.
- Build symbol table to describe label definition & uses.

`file1.s`:

```
label1:     …
            beq label2
            …

label2:     …
            call foo
            …
            ba label1
```

`file2.s`:

```
foo:        …
```

Background: x86

- x86 real mode
- x86 protected mode
- x86 registers and assembly

x86 real mode

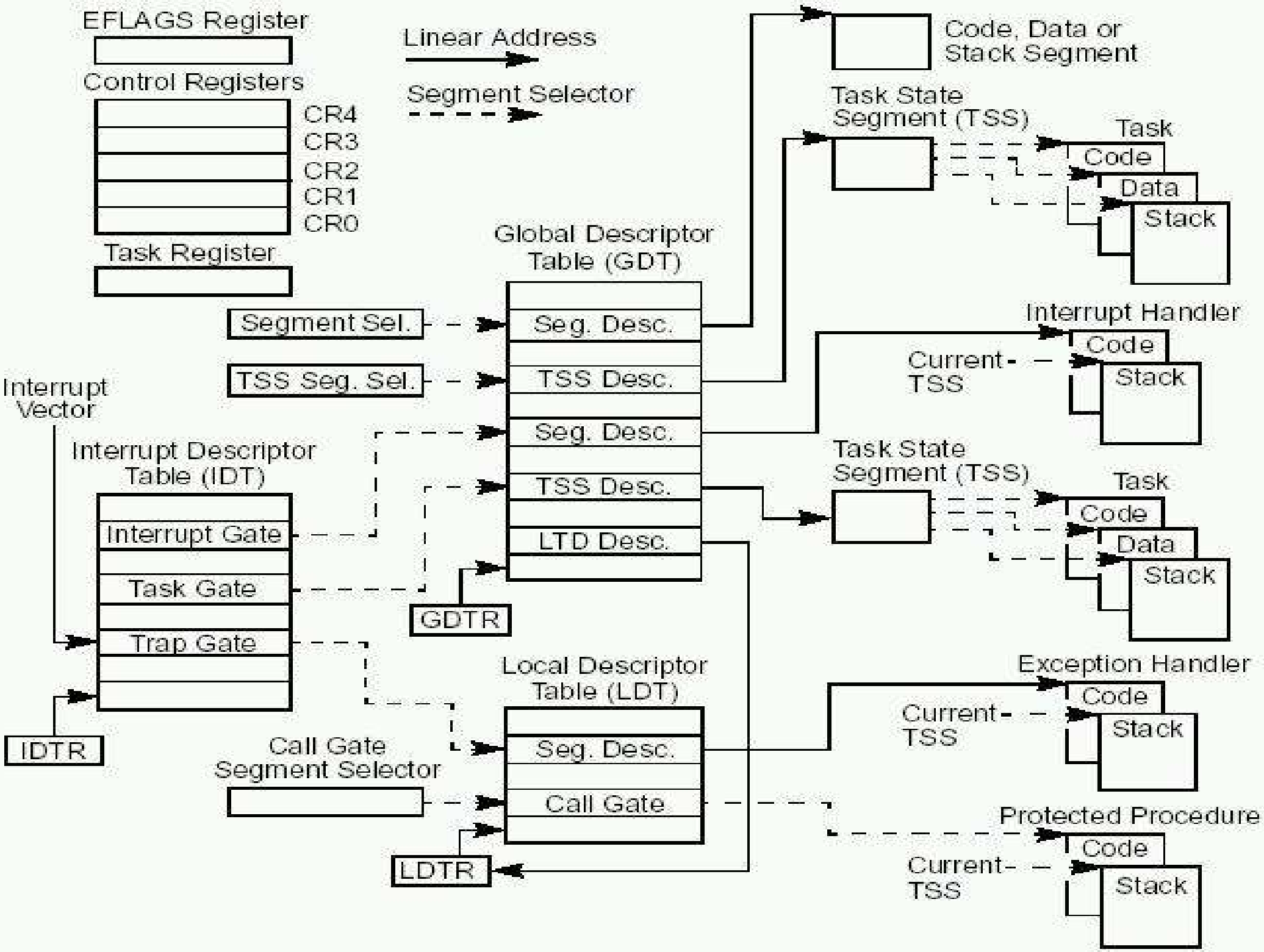
- Real mode refers to compatibility with 16 bit Intel CPUs (8086, 80286)
- All x86 CPUs start in real mode
  - The system BIOS only works in real mode
  - So, the boot code has to work in real mode?
- Segmented memory
  - Memory addresses formed by 16 bit segment and offset: address = (segment<<4) + offset
  - Effectively gives a 20 bit address space
  - Awkward to work with objects larger than a segment (64KB)
  - No paging or memory protection

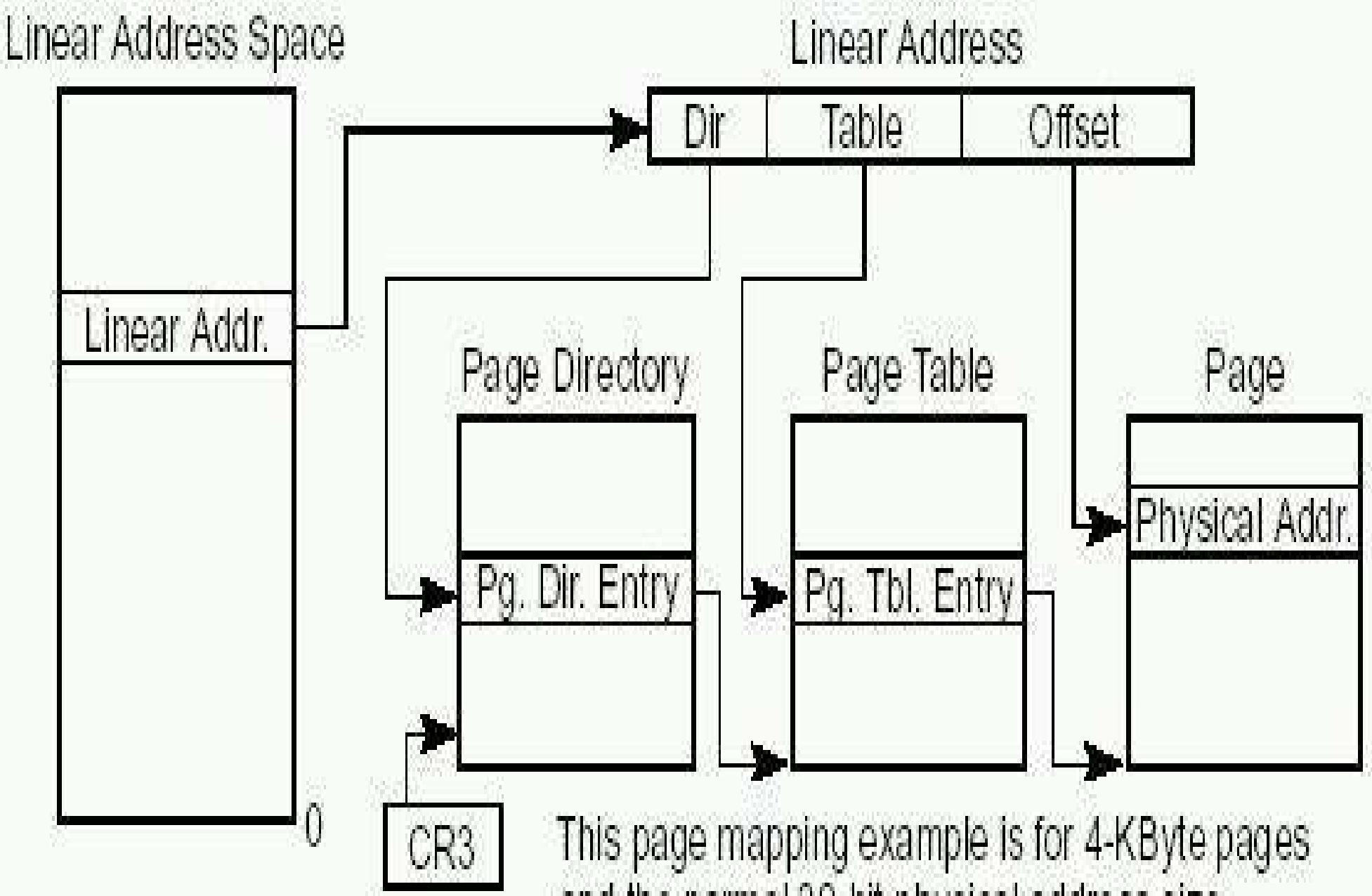
x86 real mode

- Segment registers
  - cs - code segment
  - ds - data segment (default segment for data accesses)
  - es, fs, gs - "extra" data segments, may be accessed through a "segment override"
  - ss - stack segment
- The GNU assembler has very limited support for targeting real mode
  - Can do so to a limited extent by putting a prefix code in front of every instruction
  - Eg: FreeBSD boot code uses special hacks to work around this limitation (m4 macros to "hand-assemble" some instructions)

x86 protected mode

- The 386 and higher CPUs have a "protected mode"
  - 32 bit memory addresses
  - VM paging
  - memory protection
- The 286 has a 16 bit protected mode, but rarely used
- Protected mode changes the meaning of the segment registers
  - Instead of a segment address, the segment address is (essentially) an index into a descriptor table (GDT or LDT)
  - Contents of segment registers referred to as "selectors"


EFLAGS Register
Control Registers
CR4
CR3
CR2
CR1
CR0
Task Register
Linear Address
Segment Selector
Code, Data or
Stack Segment
Task State
Segment (TSS)
Task
Code
Data
Stack
Global Descriptor
Table (GDT)
Segment Sel.
Seg. Desc.
TSS Seg. Sel.
TSS Desc.
Seg. Desc.
TSS Desc.
LTD Desc.
Interrupt Handler
Code
Current
TSS
Stack
Interrupt
Vector
Interrupt Descriptor
Table (IDT)
Interrupt Gate
Task Gate
Trap Gate
GDTR
IDTR
Task State
Segment (TSS)
Task
Code
Data
Stack
Local Descriptor
Table (LDT)
Exception Handler
Code
Current
TSS
Stack
Call Gate
Segment Selector
Seg. Desc.
Call Gate
LDTR
Protected Procedure
Code
Current
TSS
Stack


Linear Address Space
Linear Address
Dir
Table
Offset
Linear Addr.
Page Directory
Page Table
Page
Physical Addr.
Pg. Dir. Entry
Pg. Tbl. Entry
0
CR3
This page mapping example is for 4-KByte pages
and the normal 32-bit physical address size.

x86 protected mode

- Descriptor tables
  - GDT - Global Descriptor Table (required)
  - LDT - Local Descriptor Table (optional)
- The GDT and LDT can contain several types of descriptors
  - memory segments (code, data, or stack)
    - may define base address and limit
    - most OSes just set base=0 and limit=4GB (flat addressing)
  - task segments (TSS)
  - Task gates (not used by Linux), call gates

x86 protected mode

- Privileges
  - There is a privilege model that determines which segments may be accessed
  - Current privilege level (CPL) == privilege level of current code segment
    - 0 is most privileged, 3 is least
    - Most OSes use 0 for kernel code, 3 for user code
    - Privilege levels 1 and 2 could be used for more fine-grained protection (e.g., device drivers)

x86 protected mode

- Interrupt handling in protected mode
  - Interrupts can result from:
    - hardware devices
    - software interrupts (i.e., the int instruction)
    - processor exceptions and faults
  - In protected mode, the IDTR stores the address and size of interrupt table
  - Each entry in the table contains a selector and 32 bit offset
  - The selector refers to an entry in the GDT
    - Generally, it will be a code segment (with the offset specifying the ISR)
    - However, it could also reference an interrupt gate, trap gate, or task gate

x86 registers and assembly

- General registers
  - eax - used for return value
  - ebx - "base register"
  - ecx - used for count in string and loop instructions
  - edx - destination for string operations
- Each may accessed as 8 or 16 bit registers
  - E.g.: al, ah, ax
- Other registers
  - esi - source index for string instructions
  - edi - desination index for string instructions
  - ebp - base pointer (i.e., frame pointer)
  - esp - stack pointer

x86 registers and assembly

- Segment registers (see include/asm-i386/segment.h)
  - cs - code segment
    - #define __KERNEL_CS 0x10 //2.5.64:(GDT_ENTRY_KERNEL_CS * 8) // 12*8
    - Base 0x00000000 Limit 0xfffff Granularity=1 (pagesize) System=1 DPL=00 (kernel)
    - #define __USER_CS 0x23 // 2.5.64: (GDT_ENTRY_DEFAULT_USER_CS * 8 + 3)
    - Base 0x00000000 Limit 0xfffff Granularity=1 (pagesize) System=0 DPL=03 (user)
  - ds - data segment
    - Similar to cs: __KERNEL_DS & __USER_DS
  - es, fs, gs – extra data segments
    - Not used (used in older Linuxes)
  - ss - stack segment: same as data segment
- Other registers
  - eflags - flags register
  - eip - instruction pointer

/proc/1/maps

```
 Addr range              perms   offset     file    inode
08048000-0804e000 r-xp   00000000 03:01 154976   /sbin/init                  CS
0804e000-08050000 rw-p  00005000 03:01 154976   /sbin/init                  DS
08050000-08054000 rwxp 00000000 00:00 0                                     bss
40000000-40016000 r-xp   00000000 03:01 77442    /lib/ld-2.2.4.so            CS
40016000-40017000 rw-p  00015000 03:01 77442    /lib/ld-2.2.4.so            DS
40025000-40026000 rw-p  00000000 00:00 0                                    bss
40026000-40158000 r-xp   00000000 03:01 170372   /lib/i686/libc-2.2.4.so CS
40158000-4015d000 rw-p  00131000 03:01 170372  /lib/i686/libc-2.2.4.so  DS
4015d000-40161000 rw-p  00000000 00:00 0                                    bss
 bfffe000-c0000000 rwxp  fffff000     00:00 0                                stack
```

process 1: init process

usually execve() takes care of mapping ld.so into address space

perms: p means private mapping (disk image not upd)

x86 registers and assembly

- Assembly syntax
  - Linux, FreeBSD, ... use the GNU assembler
  - Mnemonics are slightly different than those defined by Intel
    - Use of suffix to designate operand size: addb, addw, addl
  - Destination operand is last
  - Registers preceded by `%'
  - Constants preceded by `$'
  - Indirection through register specified by parens
  - Indirection is assumed for symbols (must use `$' prefix to get an address)
  - In C code: __asm__("<asm routine>" : output : input : modify)

x86 registers and assembly

- Instruction formats
  - Most instructions modify one of the operands
  - Most instructions support
    - register/register
    - register/memory
    - register/immediate
    - memory/immediate
  - Lots of complex addressing modes

x86 registers and assembly

- Test and conditional instructions
  - Tests modify the flags register as a side-effect
    - e.g., carry bit, zero bit
  - Conditional branches use a flag bit as input
  - Example:

```
cmpl %eax, var     # compare %eax with variable
jl $label          # branch to label if %eax is less
```

Special instructions

- cli - disable interrupts
- sti - enable interrupts
- cld - clear direction flag
  - direction flag used in string instructions
- bswap - byte swapping (endian conversion)
- cmpxchg - atomic compare and exchange (see include/asm-i386/system.h)

x86 registers and assembly

- Examples:

```
addl $1, %eax     # add 1 to contents of %eax
addl $1, (%eax)   # add 1 to dword addressed by %eax
addl (%eax), %ebx  # add contents of dword addressed  by %eax to %ebx
addl $1, 10(%eax)  # add 1 to dword at address %eax+10
addl var, %eax    # add contents of var to %eax
movl $var, %eax   # store address of var in %eax
```

```
#ifdef CONFIG_SMP
#define LOCK "lock ; "
#else
#define LOCK ""
#endif
```

```
    Typedef struct { volatile int counter; } atomic_t;
    #define atomic_read(v)        ((v)->counter)
    #define atomic_set(v,i)       (((v)->counter) = (i))
Static __inline__ void atomic_add(int i, atomic_t *v){
      __asm__ __volatile__(
            LOCK "addl %1,%0"
            :"=m" (v->counter)
            :"ir" (i), "m" (v->counter)); // m: memory
}                                          // ir: imm reg
```

Spinlocks & Semaphores

- Many shared data between different parts of code in the kernel
  - most common: access to data structures shared between user process context and interrupt context
- In uniprocessor system: mutual excl by setting and clearing interrupts
- SMP: three types of spinlocks: vanilla (basic), read-write, big-reader
  - Read-write spinlocks when many readers and few writers
    - Eg: access to the list of registered filesystems.
  - Big-reader spinlocks a form of read-write spinlocks optimized for very light read access, with a penalty for writes.
    - limited number of big-reader spinlocks users.
    - used in networking part of the kernel.
- These semaphores different from IPC's.
- two types of semaphores: basic and read-write semaphores.
  - Mutex or counting
  - `up()` & `down()` to +/- semaphore value; interruptible or not also

Spinlocks: (cont'd)

- A good example of using spinlocks: accessing a data strucuture shared betw a user context and an interrupt handler

```
spinlock_t my_lock = SPIN_LOCK_UNLOCKED;
my_ioctl() {
     spin_lock_irq(&my_lock);
     /* critical section */
     spin_unlock_irq(&my_lock);
}
my_irq_handler() {
     spin_lock(&lock);
     /* critical section */
     spin_unlock(&lock);
}
```

spin_lock: if interrupts disabled or no race with interrupt context
spin_lock_irq: if interrupts enabled and has to be disabled
spin_lock_irqsave: if interrupt state not known

Spinlock

- static inline void **spin_lock**(spinlock_t *lock) {
- #if SPINLOCK_DEBUG
- __label__ here;
- here:
- if (lock->magic !=SPINLOCK_MAGIC){
- printk("eip: %p\n", &&here);
- BUG();
- }
- #endif
- __asm__ __volatile__(
- spin_lock_string
- :"=m" (lock->lock) : : "memory");
- }

- #define **spin_lock_string** \
- "\n1:\t" \
- "lock ; decb %0\n\t" \
- "js 2f\n" \
- LOCK_SECTION_START("") \
- "2:\t" \
- "cmpb $0,%0\n\t" \
- "rep;nop\n\t" \
- "jle 2b\n\t" \
- "jmp 1b\n" \
- LOCK_SECTION_END
- #define **spin_unlock_string** \
- "movb $1,%0" \
- :"=m" (lock->lock) : : "memory"

Cmpxchg

```
static inline unsigned long __cmpxchg(volatile void *ptr, unsigned long old,
                                      unsigned long new, int size) {
        unsigned long prev;
        switch (size) {
        case 1:
                __asm__ __volatile__(LOCK_PREFIX "cmpxchgb %b1,%2"
                                     : "=a"(prev)
                                     : "q"(new), "m"(*__xg(ptr)), "0"(old)
                                     : "memory");
                return prev;
        case 2: ... “w %w1”... instead of “b %b1”
        case 4:... “l %1”... instead of  “b %b1”
        }
        Return old;
}
struct __xchg_dummy { unsigned long a[100]; };
#define __xg(x) ((struct __xchg_dummy *)(x))
```

x86 registers and assembly

- System registers
  - cr0 - system flags register (enable paging, cache control)
- cr4 - virtual memory extension
  - cr2 - page fault linear address
  - cr3 - page directory base addressns
- Descriptor tables
    - These registers only used in protected mode
  - gdtr - address and limit of GDT
  - idtr - address and limit of IDT
  - ldtr - index in GDT of LDT descriptor
  - tr - index in GDT of current task descriptor

x86 protected mode

- Entering protected mode
  - Construct a valid GDT and IDT, load GDTR and IDTR
    - GDT must define code, data, and stack segments
  - Set PE bit in CR0 register
  - Jump to a valid code address in a code segment
- Paging
  - 386 and higher CPUs support paging
  - Enable by setting PG bit in CR0 register
  - Three levels
    - page directory: physical addresses of page tables (CR3 register)
    - page table: physical addresses of pages
    - Pages

Switch to Protected Mode

```
#ifdef __ELF__
#define EXT(x)        x
#else
#define EXT(x)        _ ## x
#endif
.code16
.globl  EXT(_start)
.type EXT(_start), @function

EXT(_start):
   cli
   xorl    %eax, %eax
   movl    %eax, %cr3 // inval  TLB
   invd /* inval cache, no writeback*/
   movw    %cs, %ax
   shlw    $4, %ax
   movw    $EXT(gdtptr16_offset), %bx
   subw    %ax, %bx // get
   data32  lgdt %cs:(%bx)
     // cs: GDT base; bx: limit
   movl    %cr0, %eax
   andl    $0x7FFAFFD1, %eax  /
      *PG,AM,WP,NE,TS,EM,MP = 0 */
  /* PG: Paging (bit 31 of CR0): disable paging
     AM: Alignment Mask (bit 18): no align check
     WP: Write Protect (bit 16 of CR0): no trap if
      ker writes to user RO page
     NE: Numeric Error (bit 5 of CR0).
     TS: Task Switched (bit 3): no delayed saving
      of FPregs
     EM; Emulation (bit 2 of CR0); no sw i387
     MP: Monitor Coprocessor (bit 1 of CR0) */
   orl     $0x60000001, %eax
      /* CD, NW, PE = 1 */
  /* CD: cache disable
     NW: Not Write-through (bit 29 of CR0)
     PE: Prot Enable (bit 0): Enable prot mode */
   movl    %eax, %cr0
  /* Now that we are in protected mode jump to
      a 32 bit code segment. */
   data32  ljmp    $ROM_CODE_SEG,
      $__protected_start
```

Syscall use and get user AS variable

- ssize_t write(int fd, const void *buf, size_t count)

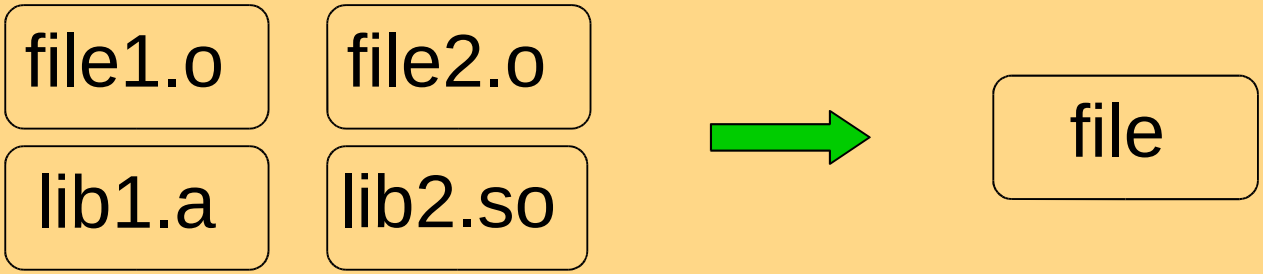
```
.include "defines.h"
.data
hello:
        .string "hello world\n"

.globl  main
main:
        movl    $SYS_write,%eax // write
        movl    $STDOUT,%ebx  //  fd
        movl    $hello,%ecx         // buf
        movl    $12,%edx            // count
        int     $0x80
        movl    $SYS_exit,%eax    // exit
        xorl     %ebx,%ebx         // (0)
        int     $0x80
        ret
```

- Inputs: %eax contains address
- Outputs:
  - %eax error code (0 or -EFAULT)
  - %edx zero-extended value

```
.align 4
.globl __get_user_4
__get_user_4:
        addl $3,%eax
        movl %esp,%edx
        jc bad_get_user
        andl $0xffffe000,%edx
        cmpl 12(%edx),%eax
        jae bad_get_user
3:      movl -3(%eax),%edx
        xorl %eax,%eax
        ret
```

Overview: Linking

What is linking?

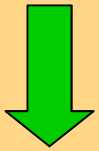


- “Glue” relocatable code & libraries together.
- Assign addresses to labels throughout code.
  - 2-pass assembler
  - 1 ½-pass assembler
  - 1-pass assembler
  - Output executable file.

Linking: Example

```
int  x = 3;
int  y = 4;
int  z;

int foo(int a) {…}
int bar(int b) {…}
```

```
extern int  x;
static int  y = 6;
int         z;

int foo(int a);
static int bar(int b) {…}
```



?

?

Note: Linking uses object files. Example uses source-level for convenience.

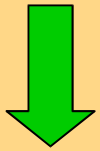
Linking: Example

```
int  x = 3;
int  y = 4;
int  z;

int foo(int a) {…}
int bar(int b) {…}
```

Defined in one file.

```
extern int  x;
static int  y = 6;
int         z;

int foo(int a);
static int bar(int b) {…}
```

Declared in other files.



```
int  x = 3;



int foo(int a) {…}
```

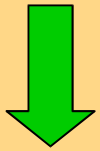
Only one copy exists.

Linking: Example

```
int  x = 3;
int  y = 4;
int  z;

int foo(int a) {…}
int bar(int b) {…}
```

```
extern int  x;
static int  y = 6;
int         z;

int foo(int a);
static int bar(int b) {…}
```

Private names not in symbol table. Can't conflict with other files' names.



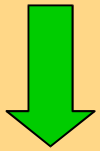
```
int  x = 3;
int  y = 4;
int  y' = 6;


int foo(int a) {…}
int bar(int b) {…}
int bar'(int b) {…}
```

Renaming is a convenient source-level way to understand this.

Linking: Example

```
int  x = 3;
int  y = 4;
int  z;

int foo(int a) {…}
int bar(int b) {…}
```

```
extern int  x;
static int  y = 6;
int         z;

int foo(int a);
static int bar(int b) {…}
```



```
int  x = 3;
int  y = 4;
int  y' = 6;
int  z;

int foo(int a) {…}
int bar(int b) {…}
int bar'(int b) {…}
```

C allows you to omit "**`extern`**" in some cases. **Don't!**

Static linking: execve

execve.c

- int execve(const char *path, char *const argv[], char *const envp[])
- #include <stdio.h>
- void main() {
- char *name[2];
- name[0] = "/bin/sh";
- name[1] = NULL;
- execve(name[0], name,NULL);
- }

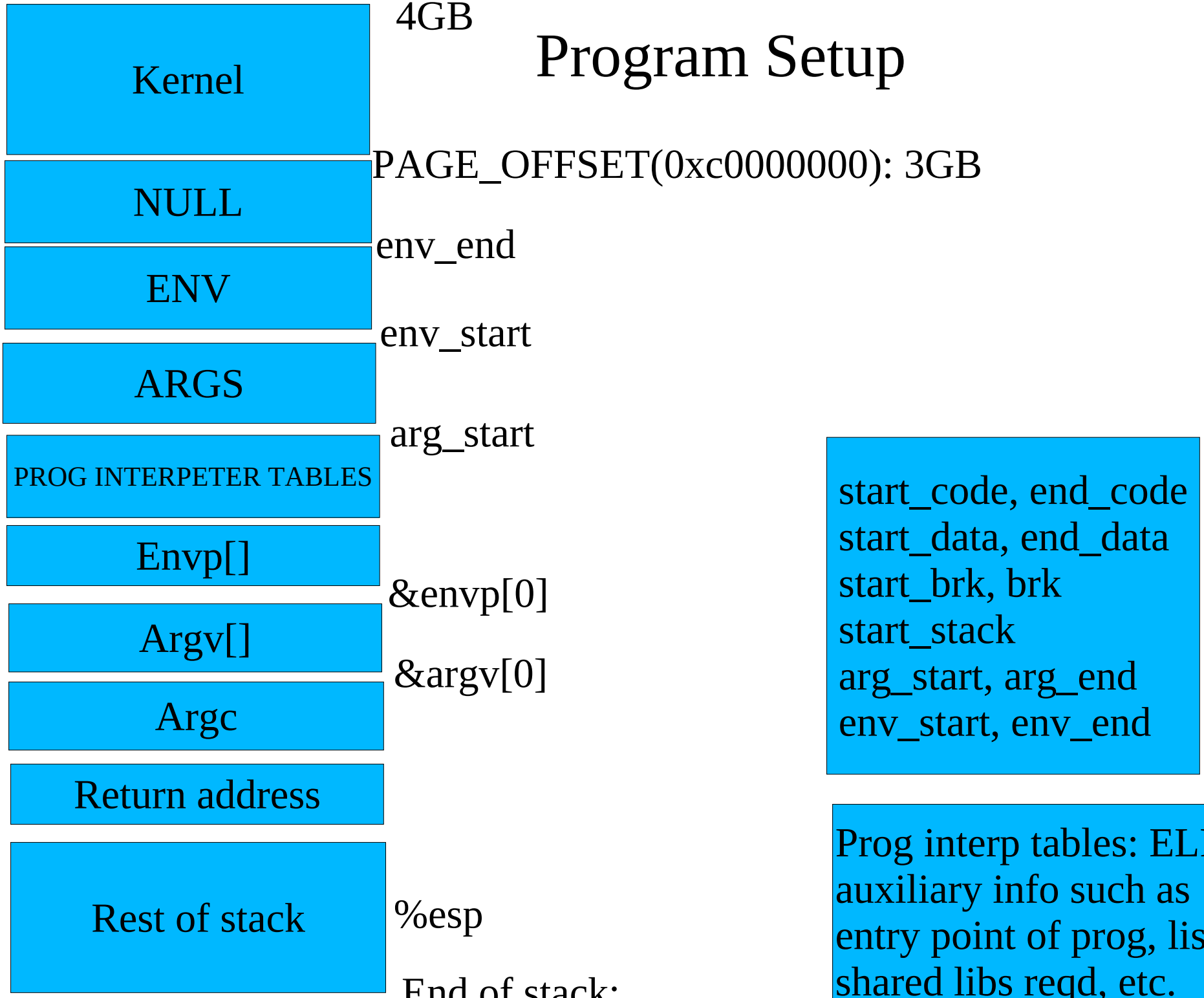
gcc -o execv -ggdb -static execve.c

  - static: includes execve code; o.w., picked up from libc

- gdb execve

- disassemble main
- 0x8000130 <main>: pushl %ebp // save bp
- 0x8000131 <main+1>: movl %esp,%ebp // bp=sp
- 0x8000133 <main+3>: subl $0x8,%esp // name[2]
- 0x8000136 <main+6>: movl $0x80027b8, // “bin/sh”
  0xfffffff8(%ebp) // name[0]
- 0x800013d <main+13>: movl $0x0, 0xfffffffc(%ebp)
- 0x8000144 <main+20>: pushl $0x0 // push 0
- 0x8000146 <main+22>: leal 0xfffffff8(%ebp),%eax
- 0x8000149 <main+25>: pushl %eax //push name
- 0x800014a <main+26>: movl 0xfffffff8(%ebp),%eax
- 0x800014d <main+29>: pushl %eax //push name[0]
- 0x800014e <main+30>: call 0x80002bc <__execve>
- 0x8000153 <main+35>: addl $0xc,%esp
- 0x8000156 <main+38>: movl %ebp,%esp
- 0x8000158 <main+40>: popl %ebp
- 0x8000159 <main+41>: ret

Program Setup



__execve

- (gdb) disassemble __execve
- 0x80002bc <__execve>: pushl %ebp
- 0x80002bd <__execve+1>: movl %esp,%ebp
- 0x80002bf <__execve+3>: pushl %ebx
- 0x80002c0 <__execve+4>: movl $0xb,%eax
  - Copy addr of “/bin/sh” into reg
- 0x80002c5 <__execve+9>: movl 0x8(%ebp),%ebx
  - 11 is syscall number for execv
- 0x80002c8 <__execve+12>: movl 0xc(%ebp),%ecx
  - Copy address of name[] into reg
- 0x80002cb <__execve+15>:movl0x10(%ebp),%edx
  - Copy address of null pointer into reg
- 0x80002ce <__execve+18>: int $0x80
  - Change into kernel mode!
- 0x80002d0 <__execve+20>: movl %eax,%edx
- 0x80002d2 <__execve+22>: testl %edx,%edx
- 0x80002d4 <__execve+24>: jnl 0x80002e6<__execve+42>
- 0x80002d6 <__execve+26>: negl %edx
- 0x80002d8 <__execve+28>: pushl %edx
- 0x80002d9 <__execve+29>: call 0x8001a34 <__normal_errno_loc>
- 0x80002de <__execve+34>: popl %edx
- 0x80002df <__execve+35>: movl %edx,(%eax)
- 0x80002e1 <__execve+37>: movl $0xffffffff,%eax
- 0x80002e6 <__execve+42>: popl %ebx
- 0x80002e7 <__execve+43>: movl %ebp,%esp
- 0x80002e9 <__execve+45>: popl %ebp
- 0x80002ea <__execve+46>: ret
- 0x80002eb <__execve+47>: nop

Static & Dynamic Libraries

Library: Collection of pre-compiled relocatable code & data.
May be linked with other code.

Static	Dynamic
Linked at compile-time	Linked at run-time
UNIX: foo.a	UNIX: foo.so
Old style	New style

What are the differences?

Static & Dynamic Libraries

Static

Library code is in executable file

- – Larger executables
- – Must recompile to use newer libraries.
- \+ Executable is self-contained.
- • Some time to load libraries at startup-time

Library code not shared

- – Higher memory usage.

Dynamic

Library code not in executable file

- \+ Smaller executables
- \+ Can use newest (or smallest, fastest, …) library.
- – Depends on libraries at run-time.
- • Some time to load libraries at run-time

Library code shared

- \+ Lower memory usage.

Static & Dynamic Libraries

Static

Linking implementation:

Append all corresponding bss segments together, & resolve addresses.

\+ Easy to understand & implement.

– Links all code & data from library.

Common, but now obsolete.

Dynamic

Linking implementation:

When unresolved symbol is referenced, find it in libraries, & load the object into memory.

– More difficult to understand & implement.

\+ Links only needed code & data from library.

Common, current standard.

Static & Dynamic Libraries

Static

Creation

```
ar –r libfoo.a bar.o baz.o
ranlib libfoo.a //create index
```

Use

```
gcc –o quux quux.o -lfoo
```

Dynamic

Creation

```
gcc –shared libfoo.so bar.o baz.o
```

Use

```
gcc –o quux quux.o -lfoo
```

Dynamically linked program

- #include <stdio.h> main () { printf ("Hello!"); }
- file a.out: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs), not stripped
- nm gives all the symbols in the executable file with addr & *type*:
  - Lowercase: the symbol is local; if uppercase, the symbol is global (external)
  - A: absolute
  - B: BSS
  - D: data
  - R: read-only data
  - T: text
  - U: undefined
  - W: weak symbol

nm -v a.out

- w __gmon_start__
- w _Jv_RegisterClasses
- U __libc_start_main@@GLIBC_2.0
- U printf@@GLIBC_2.0
- 08048230 T _init
- 08048278 T _start
- 0804829c t call_gmon_start
- 080482c0 t __do_global_dtors_aux
- 080482fc t frame_dummy
- 08048328 T main
- 0804834c t __do_global_ctors_aux
- 08048370 T _fini
- 0804838c R _fp_hw
- 08048390 R _IO_stdin_used
- 0804939c D __data_start
- 0804939c W data_start
- 080493a0 d __dso_handle
- 080493a4 d p.0
- 080493a8 d __EH_FRAME_BEGIN__
- 080493a8 d __FRAME_END__
- 080493ac D _DYNAMIC
- 08049474 d __CTOR_LIST__
- 08049478 d __CTOR_END__
- 0804947c d __DTOR_LIST__
- 08049480 d __DTOR_END__
- 08049484 d __JCR_END__
- 08049484 d __JCR_LIST__
- 08049488 D _GLOBAL_OFFSET_TABLE_
- 080494a0 A __bss_start
- 080494a0 b completed.1
- 080494a0 A _edata
- 080494a4 A _end

Code

- (gdb) disassemble main
- 0x8048328 <main>: push %ebp
- 0x8048329 <main+1>: mov %esp,%ebp
- 0x804832b <main+3>: sub $0x8,%esp
- 0x804832e <main+6>: and $0xfffffff0,%esp
- 0x8048331 <main+9>: mov $0x0,%eax
- 0x8048336 <main+14>: sub %eax,%esp
- 0x8048338 <main+16>: sub $0xc,%esp
- 0x804833b <main+19>: push $0x8048394
- 0x8048340 <main+24>: call 0x8048268 <printf>
- 0x8048345 <main+29>: add $0x10,%esp
- 0x8048348 <main+32>: leave
- 0x8048349 <main+33>: ret
- 0x804834a <main+34>: nop
- 0x804834b <main+35>: nop

- (gdb) disassemble _init
- 0x8048230 <_init>: push %ebp
- 0x8048231 <_init+1>: mov %esp,%ebp
- 0x8048233 <_init+3>: sub $0x8,%esp
- 0x8048236 <_init+6>: call 0x804829c <call_gmon_start>
- 0x804823b <_init+11>: nop
- 0x804823c <_init+12>: call 0x80482fc <frame_dummy>
- 0x8048241 <_init+17>: call 0x804834c <__do_global_ctors_aux>
- 0x8048246 <_init+22>: leave
- 0x8048247 <_init+23>: ret

strace ./a.out

- Execve("./a.out", ["./a.out"], [/* 35 vars */]) = 0
- uname({sys="Linux", node="todi", ...}) = 0
- brk(0) = 0x80494a4
- open("/etc/ld.so.preload", O_RDONLY) = -1 ENOENT (No such file or directory)
- open("/etc/ld.so.cache", O_RDONLY) = 3
- fstat64(3, {st_mode=S_IFREG|0644, st_size=57857, ...}) = 0
- old_mmap(NULL, 57857, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40013000
- close(3) = 0
- open("/lib/i686/libc.so.6", O_RDONLY) = 3
- read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\220Y\1"..., 1024) = 1024
- fstat64(3, {st_mode=S_IFREG|0755, st_size=1395734, ...}) = 0
- old_mmap(0x42000000, 1239844, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0x42000000

strace ./a.out (contd)

- mprotect(0x42126000, 35620, PROT_NONE) = 0
- old_mmap(0x42126000, 20480, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED, 3, 0x126000) = 0x42126000
- old_mmap(0x4212b000, 15140, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x4212b000
- close(3) = 0
- old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40022000
- munmap(0x40013000, 57857) = 0
- fstat64(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(136, 3), ...}) = 0
- mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40013000
- write(1, "Hello!", 6Hello!) = 6
- munmap(0x40013000, 4096) = 0
- _exit(6) = ?

Dynamically Loaded Code

- Normally, a program has all references to data and functions resolved
- Dynamically loadable code can have unresolved references
  - E.g., a program refers to a function named Foobar, but the function is not defined in program code
  - This code is not designed to run independently
- Process of loading dynamic code (or libraries)
  - File of code is loaded into memory
  - Run-time linker attempts to resolve unresolved references using a run time symbol table
  - Run time symbol table is defined by the program loading the shared library
    - Run time link errors if some references unresolved
- Some parts of a live kernel can be from Loadable Kernel Modules. They become part of kernel, but they can be loaded or unloaded dynamically.

Kernel Modules

- Modules are arbitrary sections of kernel code
- Run in kernel (privileged) mode
- Typically: device driver, file system access, networking prototocol
- Explicit & implicit loading/unloading
- Implicit:
  - When a process requires use of a module (e.g., device driver), “module requestor” can load
  - When a module has not be used in a sufficiently long time, it can be automatically unloaded
- Startup & cleanup routines
  - Startup: register services, reserve resources
  - Cleanup: called before unloading
- Kernel: service & h/w resource tables

Loadable Kernel Modules:

- support for loadable modules made at compile time
  - CONFIG_MODULES option
- Support for module autoloading via request_module() mechanism
  - CONFIG_KMOD option
- command interface for users:

  insmod: insert a single module.

  modprobe: insert a module including all other modules it depends on.
  rmmod: remove a module.

  modinfo: print some information

Object Code Formats

Many formats exist:

a.out	Old UNIX format – several variations
ELF	Modern UNIX format
COM	Very old Microsoft format
EXE	Old Microsoft format
PE	Modern Microsoft format
…	

Formats have basic similarities...

Object Code Structure

Distinguish read-only and writable info.

Allows enforcement of read-only code & data.

How? Later. (virtual memory)

Why? Multiple users share; self-protection.

Distinguish initialized & uninitialized data.

Initialized: Must specify initial data in file.

Uninitialized: Only specify how much space needed.

Typical *segments* (or *sections*) used:

- Text Read-only code & data
- Data Initialized, writable data (and potentially code)
- BSS Uninitialized, writable data

?? Abbreviation for “Block Started by Symbol” pseudo-op in IBM 704 assembler.

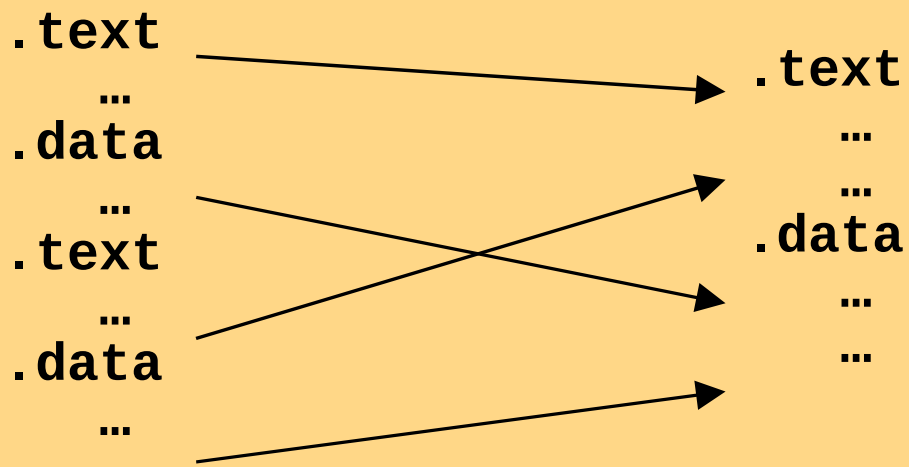
Object Code Structure (contd)

Programmer may divide segments into subparts, for convenience.

Different segments are in separate *pages*.

Page = a system-dependent unit of memory, typically ~4KB.

UNIX: Size specified by **`getpagesize()`**.



Object Code: Other Contents

Header

Description of file contents, including

- *Magic number* to identify file format
- Segment sizes
- Entry point

Symbol table

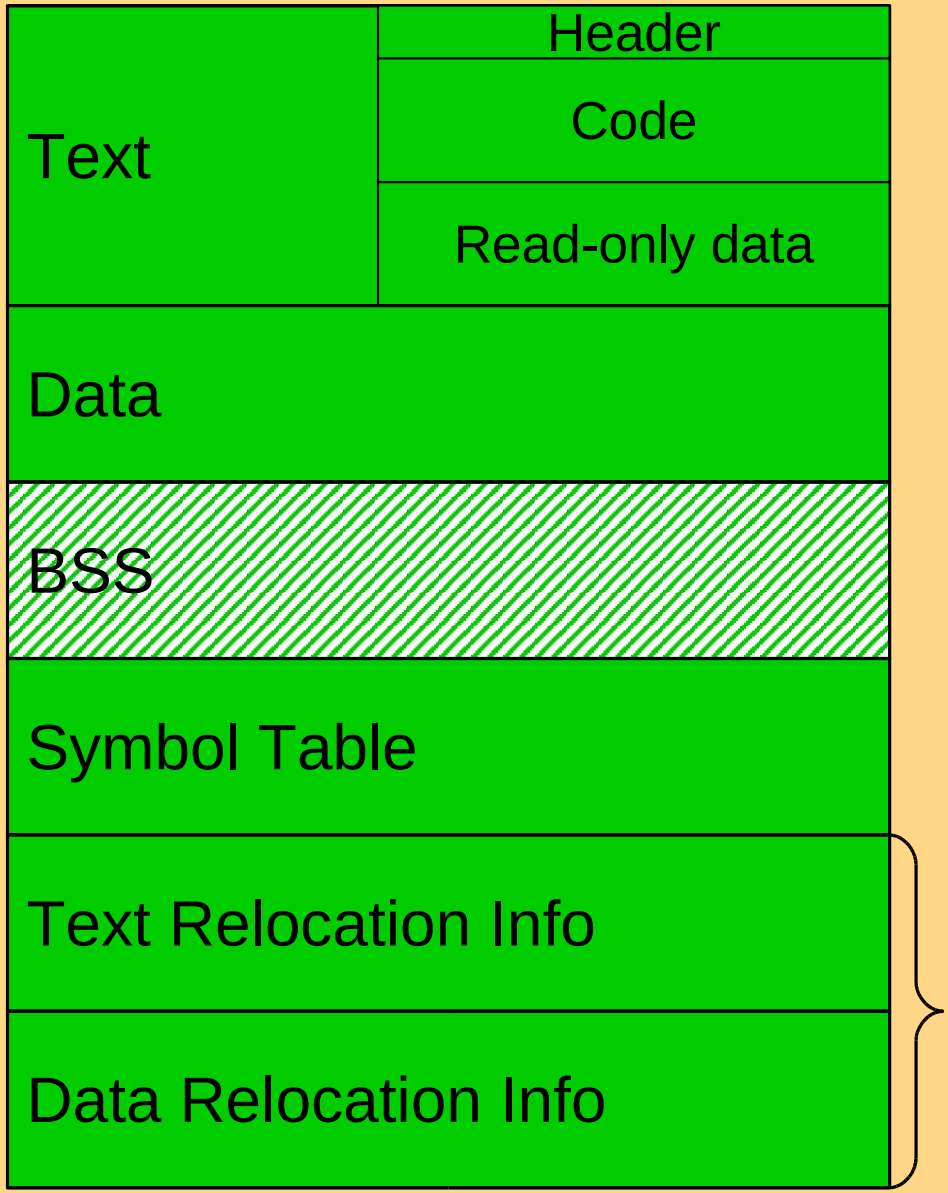
Description of label definitions.

Relocation info

Description of unresolved label uses.

- Allows separate compilation.
- May refer to labels defined in other files.

a.out Format



32 bytes

Double-word alignment

Doesn't occupy space in file, but size noted in header.

Optional for executable files.

Only in relocatable files.

Each section is page aligned.

If last Data page not full, move some BSS space into Data.

a.out Format: Loading

Create read-only blank page at beginning.

- Text will start on second page.
- User's code must remember this "hidden" page in address calculations.

Copy Text & Data into virtual memory.

- Copies header, too – small waste of memory improves simplicity.

Initialize BSS contents to zero.

- a.out file doesn't need to contain values – just the size.

a.out has some shortcomings

- no standard way of defining initialization and cleanup code (e.g., C++ static constructors and destructors)
- no standard shared library mechanism

ELF fixes these problems

ELF Format: Outline

Similar to a.out, but much more flexible.

More kinds of segments available, including…

- Dynamically-loaded symbol tables
- C++ initialization & finalization code
- Debugging information

Can specify more information about each segment.

File-specified byte-ordering.

File-specified address size.

Lots more details.

Simplifies shared libraries & dynamic linking.

ELF

- ELF - Executable and Linking Format
  - Defines format for object files, shared libraries, and executables
  - The default format for Linux since 2.0 and FreeBSD since 3.0
  - ELF is well-supported by the GNU compiler tools
  - ELF files consist of sections and segments
    - Sections are parts of the file with a distinct purpose (code, data, read-only data, init code, etc.)
    - Segments describe how the sections are to be loaded into memory by the program loader

ELF

- Structure of an ELF file
  - ELF header
  - Program header (optional)
  - Sections
  - Section table (optional)
- Kinds of sections
  - .text - program code
  - .data - read/write initialized data
  - .rodata - read-only initialized data
  - .bss - uninitialized (zero-filled) data
  - .init, .fini - initialization and cleanup code
  - also symbol tables and relocation sections
  - the format allows arbitrary sections

ELF

- Segments
  - Provide a concise way to specify how an executable file should be loaded into memory
  - Defined by the ELF program header
  - Typically allows an executable to be loaded and mapped in two operations
    - Text and read-only data
    - Read/write data and uninitialized data

GOT

- Position-independent code cannot contain absolute virtual addresses
  - Needed for shared libraries
- Global offset tables hold absolute addresses in private data
  - static int a; extern int b; a = 1; b = 2;
  - => movl $1, a@GOTOFF(%ebx) // a =1
  - => movl b@GOT(%ebx), %eax; movl $2, (%eax) // b =2
  - (Load GOT addr in %ebx as follows:
  - call .L2 //push PC
  - L2: popl %ebx // PC into %ebx
  - addl $_GLOBAL_OFFSET_TABLE_ + [. - .L2], %ebx //
- Runtime linker processes all global offset table relocations before giving control to any code in the process image
- The table's entry zero is reserved to hold the address of the dynamic structure, referenced with the symbol _DYNAMIC.
  - For runtime linker to find its own dynamic structure without having yet processed its relocation entries; Runtime linker must initialize itself without relying on other programs to relocate its memory image.

PLT

- Procedure linkage table converts position-independent function calls to absolute locations
- Link-editor cannot resolve execution transfers such as function calls from one executable or shared object to another.
  - transfer control thru entries in PLT
  - runtime linker redirects the entries without compromising the position-independence and shareability of the program's text
- Executable files and shared object files have separate procedure linkage tables.

```
PLT0: pushl  GOT+4    // push id code for dynlib
       jmp   *GOT+8    // special call to dynlinker
PLT1: jmp   *GOT+m    // jump to next line (1st time); to func (afterwards)
      push  #reloc_offset
      jmp   PLT0
PLT1: jmp   *GOT+m (%ebx) // PIC code
      push  #reloc_offset
      jmp   PLT0
```

Before runtime linking

- .PLT0:
  unimp
  unimp
- .PLT1:
  unimp
- .PLT101:
  ld (.-.PLT0), %g1
  br .PLT0
- .PLT102:
  sethi (.-.PLT0), %g1
  ba,a .PLT0
  nop

After

- .PLT0:
  save %sp, -64, %sp //allocstack
  call runtime_linker
- .PLT1: .
  word identification
- .PLT101:
  nop
  br name101
- .PLT102:
  sethi (.-.PLT0), %g1
  sethi %hi(name102), %g1
  jmpl %g1+%lo(name102), %g0

Loading a dynamically linked program

- Execve() notices ".interp" (use of interpreter) in ELF file
  - Assume interpreter (loader) is ld.so
- Maps ld.so into some convenient range and passes to ld.so on stack:
  - addr of program hdr, size of each entry in hdr & count of entries
  - starting addr of program to jump to after init by ld.so
  - addr where ld.so is loaded
- Bootstrap code of ld.so finds its own GOT
  - 1st entry points to dynamic segment in ld.so itself! _DYNAMIC!
  - Linker now can find its own relocation entries, relocate pointers in its own data segment and resolve code references to the routines to load all else
  - Linux ld.so names all essential routines starting as _dl_ and special code looks for symbols with this string to resolve it