L6: Operating Systems Structures

Frans Kaashoek

kaashoek@mit.edu

6.033 Spring 2013

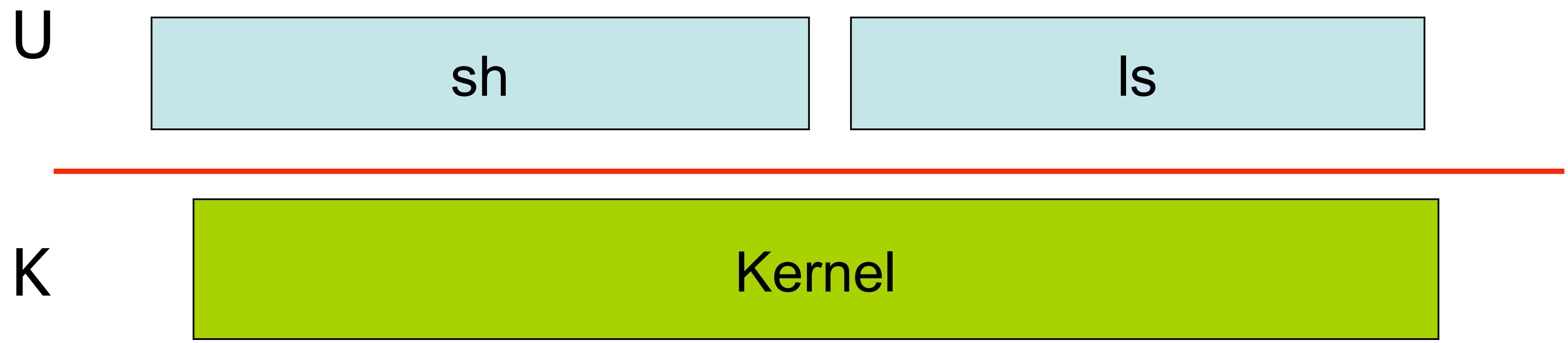
Overview

- Theme: strong isolation for operating systems
- OS organizations:
  - Monolithic kernels
  - Microkernel
  - Virtual machines

OS abstractions

- Virtual memory
- Threads
- File system
- IPC (e.g., pipes)
- …

Monolithic kernel (e.g., Linux)



- Kernel is one large C program
- Internal structure
  - E.g., object-oriented programming style
- But, no enforced modularity

Kernel program is growing

- 1975 Unix kernel: 10,500 lines of code
- 2012: Linux 3.2

    300,000 lines: header files (data structures, APIs)

    490,000 lines: networking

    530,000 lines: sound

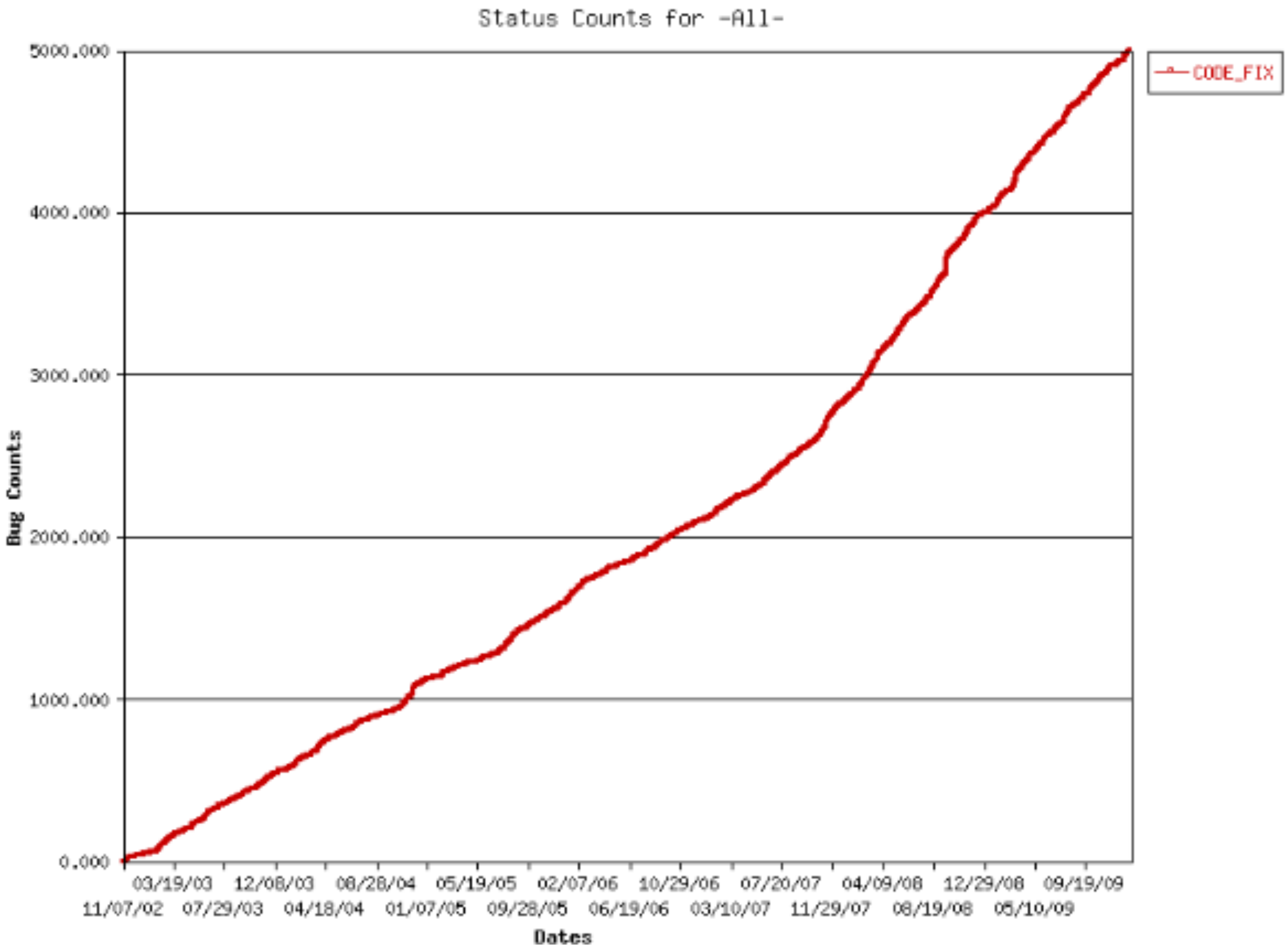
    700,000 lines: support for 60+ file systems

    1,880,000 lines: support for 25+ CPU architectures

    5,620,000 lines: drivers

    9,930,000 Total lines of code

Linux kernel has bugs



5,000 bug report fixed in ~7 years, 2+ day

How bad is a bug?

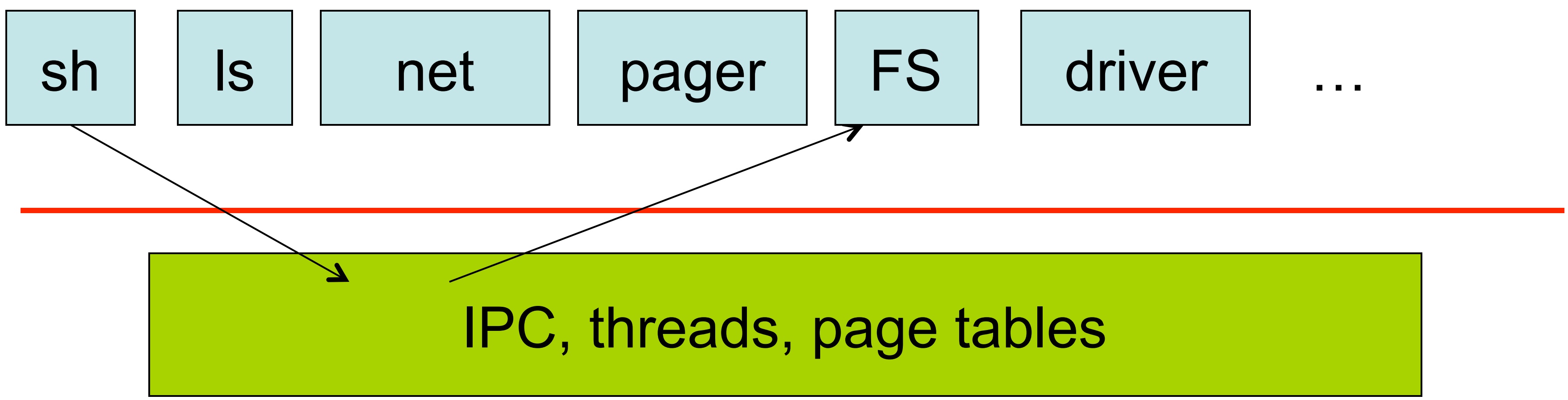
- Demo:
  - Insert kernel module
  - Every 10 seconds overwrites N locations in physical memory
  - N = 1, 2, 4, 8, 16, 32, 64, ….
- What N makes Linux crash?

Observations

- Linux lasted that long
- Maybe files were corrupted
- Every bug is an opportunity for attacker

- Can we enforce modularity within kernel?

Microkernel organization: Apply Client/Server to kernel



- User programs interact w. OS using RPC
- Examples: QNX, L4, Minix, etc.

Challenges

- Communication cost is high
  - Much higher than procedure call
- Isolating big components doesn’t help
  - If entire FS crashes, system unusable
- Sharing between subsystems is difficult
  - Share buffer cache between pager and FS

- Requires careful redesign

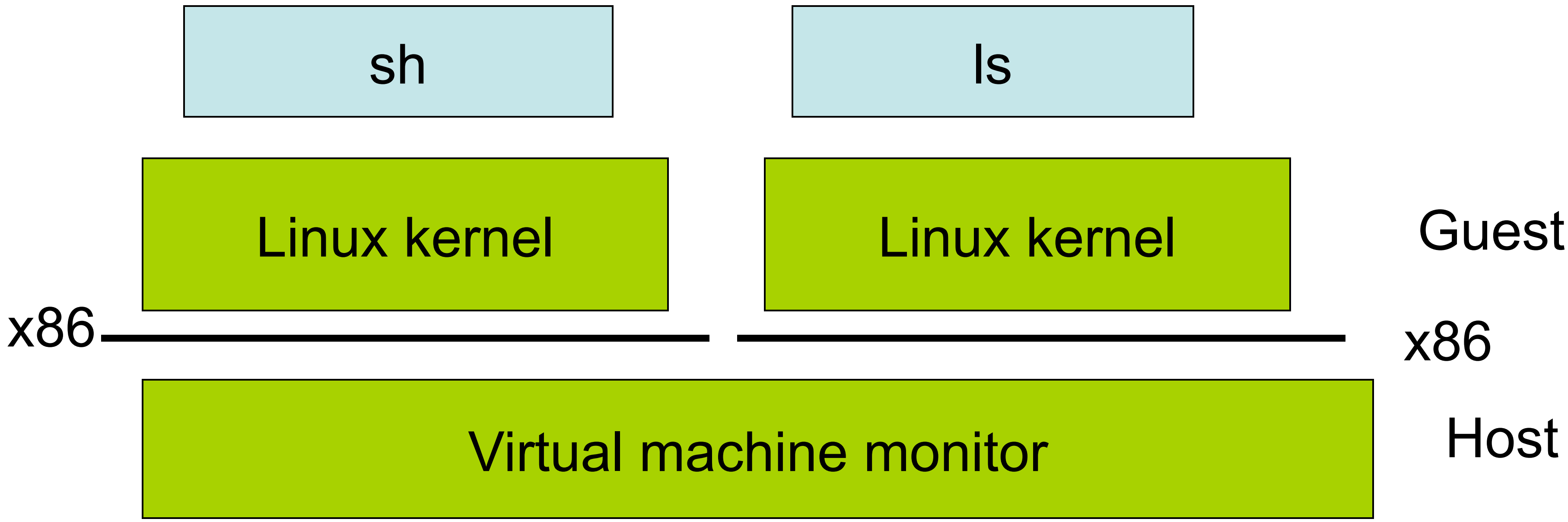
Why is Linux not a pure microkernel?

- Many dependencies between components
- Redesign is challenging
  - Trade-off: new design or new features?
- Some services are run as user programs:
  - X server, some USB drivers, SQL database, DNS server, SSH, etc.

Goal: isolation and compatibility

- Idea: run different programs on different computers
- Each computer has its on own kernel
  - If one crashes, others unaffected
  - Strong isolation
- But, cannot afford that many computers
  - Virtualization and abstraction ….
  - New constraint: compatibility

Approach: virtual machines



- Pure virtualization of hardware
  - CPU, memory, devices, etc.
- Provides strong isolation

How to implement VMM?

- One approach: pure emulation (e.g., QEMU)
  - VMM interprets every guest instruction

```
int32_t regs[8];
#define REG_EAX 1;
#define REG_EBX 2;
#define REG_ECX 3;
...
int32_t eip;
int16_t segregs[4];
...

char mem[256*1024*1024];
```

Emulation of CPU

```
for (;;) {
        read_instruction();
        switch (decode_instruction_opcode()) {
        case OPCODE_ADD:
                int src = decode_src_reg();
                int dst = decode_dst_reg();
                regs[dst] = regs[dst] + regs[src];
                break;
        case OPCODE_SUB:
                int src = decode_src_reg();
                int dst = decode_dst_reg();
                regs[dst] = regs[dst] - regs[src];
                break;
        ...
        }
        eip += instruction_length;
}
```

Goal: “emulate” fast

- Observation: guest instructions are same has hardware instructions
- Idea: run most instructions directly
  - Fine for user instructions (add, sub, mul)
  - But not for, e.g., privileged instructions
  - What hardware state must be virtualized to run several existing kernel?

Kernel virtualization

- Each kernel assumes its manages:
  - Physical memory
  - Page-table pointer
  - U/K bit
  - Interrupts, registers, etc.
- How to virtualize these?

Memory virtualization

- Idea: an extra level of page tables

Guest virtual address

↓ Kernel page table

Guest physical addresses

↓ VMM page table

Host physical addresses

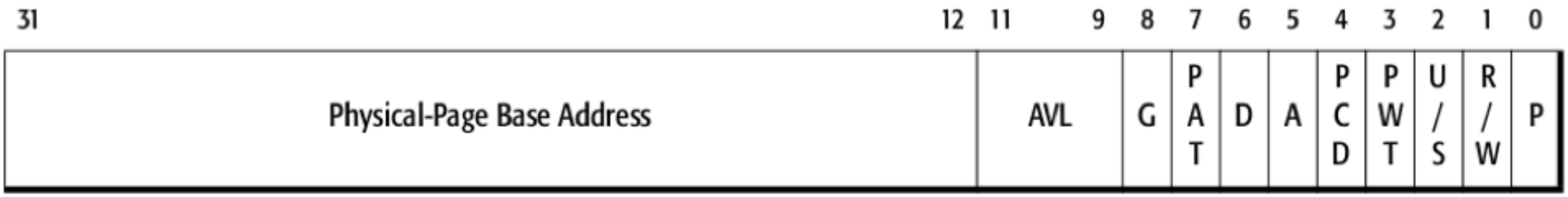
Virtualizing page table pointer

- Guest OS cannot load PTP
  - Isolation violated
  - Guest OS will specify guest physical addresses
    - Not an actual DRAM location

A solution: shadow page tables

- VMM intercepts guest OS loading PTP
- VMM iterates over guest PT and constructs shadow PT:
  - Replacing guest physical addresses with corresponding host physical addresses
- VMM loads host physical address of shadow PT into PTP

Computing shadow PT




```
compute_shadow_pt(guest_pt)
    For gva in 0 .. 2^20:
        if guest_pt[gva] & PTE_P:
                gpa = guest_pt[gva] >> 12
                hpa = host_pt[gpa] >> 12
                shadow_pt[gva] = (hpa << 12)| PTE_P
        else:
                shadow_pt[gva] = 0
```

Guest modifies its PT

- Host maps guest PT *read-only*
- If guest modifies, hardware generates page fault
- Page fault handled by host:
  - Update shadow page table
  - Restart guest

Virtualizing U/K bit

- Hardware U/K bit must be U when guest OS runs
  - Strong isolation
- But now guest cannot:
  - Execute privileged instructions
  - …

A solution: trap-and-emulate

- VMM stores guest U/K bit in some location
- VMM runs guest kernel with U set
- Privileged instructions will cause an exception
- VMM emulates privileged instructions, e.g.,
  - Set or read virtual U/K
  - if load PTP in virtual K mode, load shadow page table
  - Otherwise, raise exception in guest OS

Hardware support for virtualization

- AMD and Intel added hardware support
  - VMM operating mode, in addition to U/K
  - Two levels of page tables
- Simplifies job of VMM implementer:
  - Let the guest VM manipulate the U/K bit, as long as VMM bit is cleared.
  - Let the guest VM manipulate the guest PT, as long as host PT is set.

Virtualizing devices (e.g., disk)

- Guest accesses disk through special instructions:
- Trap-and-emulate:
  - Write “disk” block to a file in host file system
  - Read “disk” block from file in host file system

Benefits of virtual machines

- Can share hardware between unrelated services, with enforced modularity
  - “Server consolidation”
- Can run different operating systems
- Level-of-indirection tricks:
  - Snapshots
  - Can move guest from one physical machine to another

VMs versus microkernels

- Solving orthogonal problems
  - Microkernel: splitting up monolithic designs
  - VMs: run many instances of existing OS

Summary

- Monolithic kernels are complex, error-prone
  - But, not that unreliable …
- Microkernels
  - Enforce OS modularity with client/server
  - Designing modular OS services is challenging
- Virtual machines
  - Multiplex hardware between several operating systems