CS6465: Advanced Operating System Implementation

Lecture 09 - Microkernels

Anton Burtsev

November 2024

Monolithic kernels are bad

- Everything is in a single address space of the kernel
  - Processes
  - Memory management
  - Threads
  - Scheduling
  - File systems
  - Network stacks
  - Device drivers

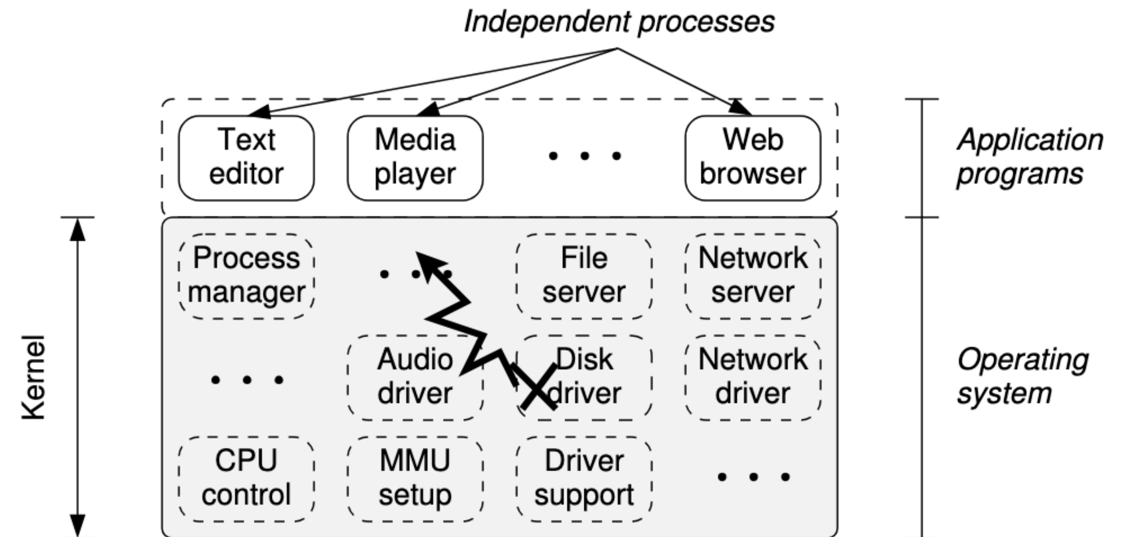


Figure 1.3: A monolithic design runs the entire OS in the kernel without protection barriers between the OS modules. A single driver fault can potentially crash the entire OS.

Linux kernel map

functionalities / layers: human interface, system, processing, memory, storage, networking

user space interfaces — system calls and system files
virtual
bridges — cross-functional modules
logical — functions implementations
device control
hardware interfaces — drivers, registers and interrupts
electronics

user peripherals | I/O | CPU | memory | disk controllers | network controllers

© 2007, 2010 Constantine Shulyupin www.MakeLinux.net/kernel_map

- 27,000,000 LoC
  - 40 subsystems
  - 8,900 device drivers

- One kernel vulnerability anywhere in the kernel
  - Any of hundreds of device drivers running on an active system
  - Some for legacy hardware
- Gives attacker a way to take over the entire system
  - Or crash it in a denial of service attack

Then why are we running monolithic kernels?

- They are fast
- Passing a network request from the network stack to the device driver is
  - A function invocation + passing a pointer

Alternative designs: microkernels and exokernels

Microkernels

Overview

- Minimal core
  - Address spaces
  - Threads
  - Inter-process communication mechanisms
  - Interrupt dispatch
  - Access control
  - Scheduler
    - Maybe
- Everything else is a user-process with an IPC

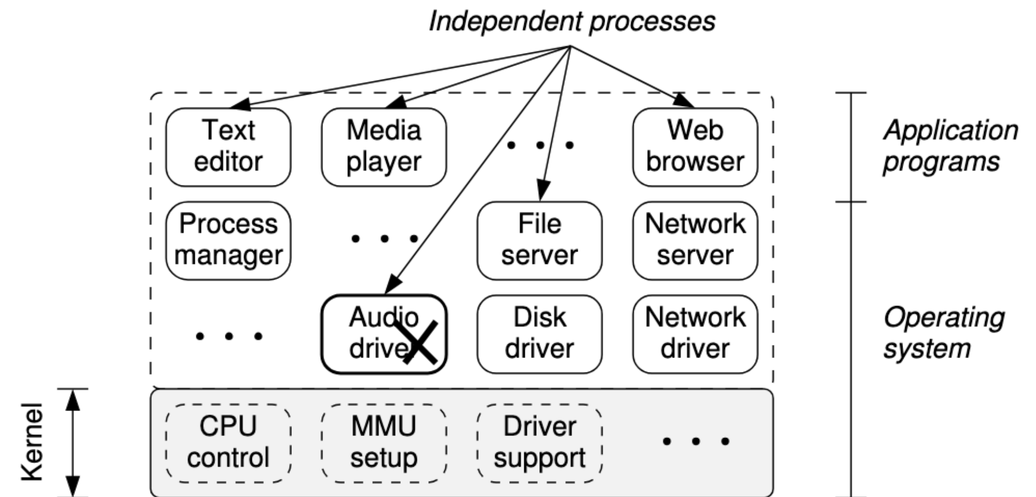


Figure 1.4: A multiserver design runs each server and driver as an independent, hardware-protected user process. This design prevents propagation of faults between the OS modules.

Benefits

- Small core
  - You can reason about correctness
  - Can even construct a mathematical proof
  - Example: seL4 (10,000 lines of code)
- Isolated subsystems
  - Even if one is exploitable, the effects are contained
  - Well, maybe it's a denial of service anyway

Challenges

- Performance:
  - Inter-process communication (IPC) is expensive
- Let’s actually take a look:
  - What is involved in an IPC?

IPC path

- sysenter/sysexit 106 cycles
- re-load CR3 170 cycles
- Total (106 + 170)*2 = 552

Synchronous vs asynchronous IPC

Synchronous IPC

- Virtual message buffer

Asynchronous IPC

- Shared memory
- Interrupt-like notification mechanism

Device drivers

- IOMMU
- Interrupts

Access control

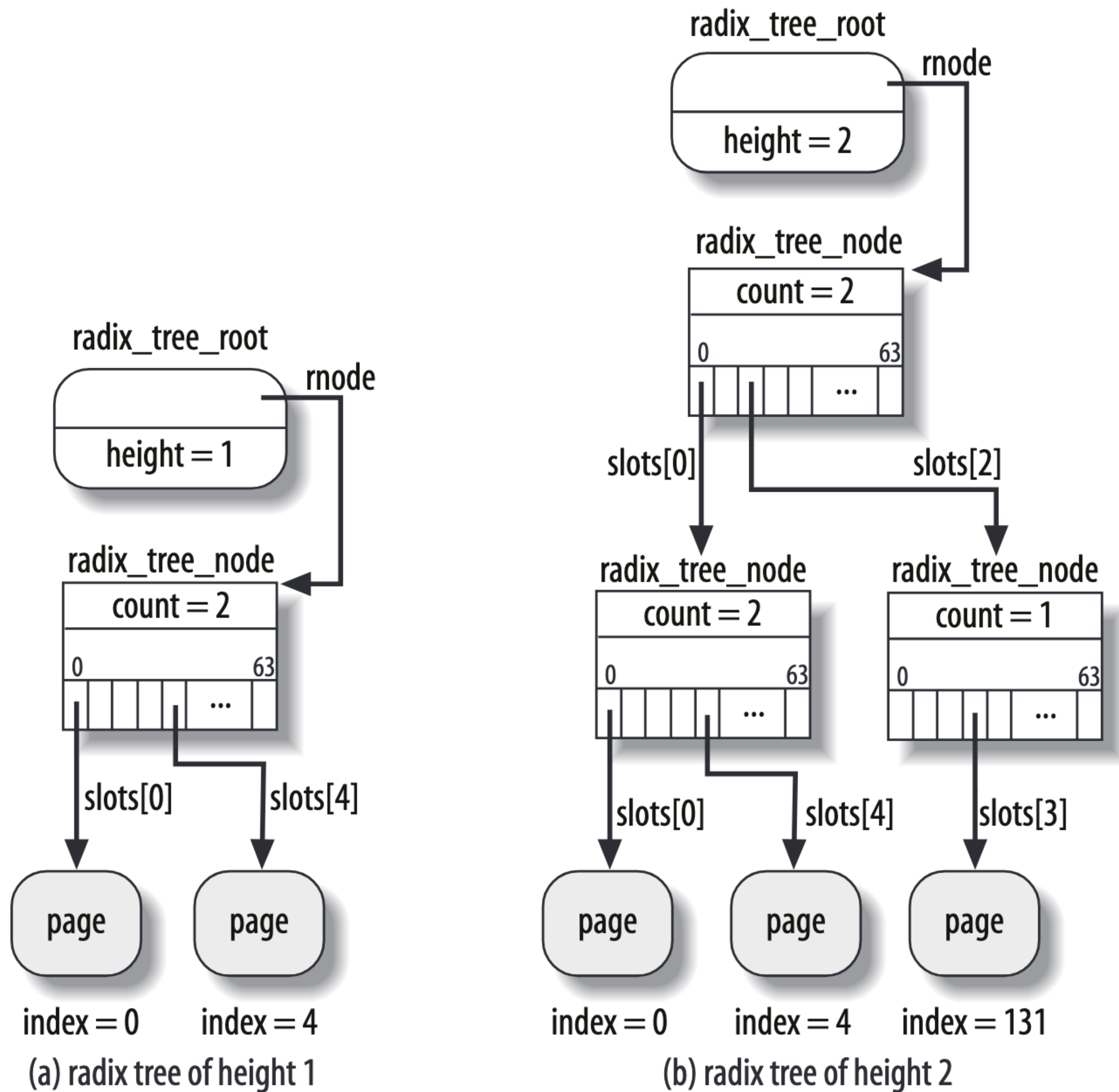
- Which process can access network card?
- Which processes can communicate?
- Is it possible to make sure processes have a guaranteed memory reservation?
  - I.e., cannot starve each other out of memory

Capability access control

- Capabilites are like pointers
- You cannot make up one
- You can pass them along (via the endpoints)
- You probably want to be able to “revoke” them
  - Get the things back under your control

Capability address spaces

- Fast lookup from a capability number to a kernel object



(a) radix tree of height 1

(b) radix tree of height 2

- If the file size grows beyond max height, must grow the tree
- Relatively simple: Add another root, previous tree becomes first child
- Scaling in height:
  - 1: 2^((6*1) +12) = 256 KB
  - 2: 2^((6*2) + 12) = 16 MB
  - 3: 2^((6*3) + 12) = 1 GB
  - 4: 2^((6*4) + 12) = 64 GB
  - 5: 2^((6*5) + 12) = 4 TB

Capability derivation tree

- Keeps track of how each capability propagated through the system
- Enables recursive revocation

Putting it all together

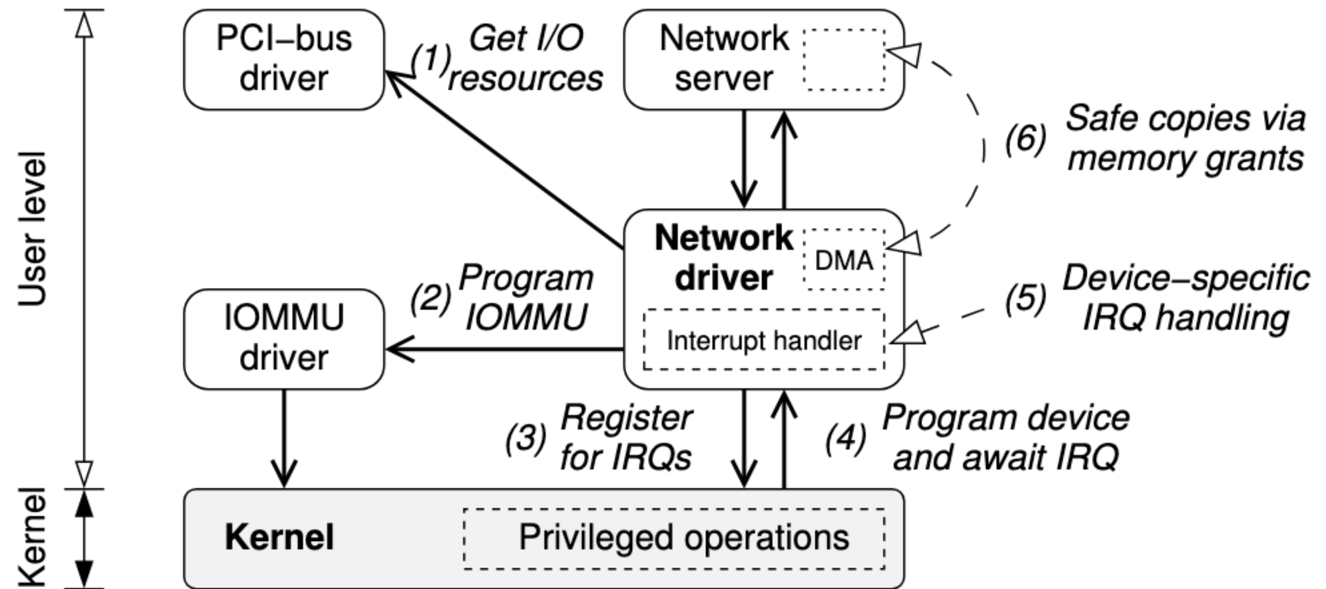


Figure 3.8: Interactions between an isolated network-device driver and the rest of the OS. Access to privileged resources is mediated by the PCI-bus driver, IOMMU driver, and the kernel.