Improving the Reliability of Commodity Operating Systems

Hank Levy
Dept. of Computer Science & Engineering
University of Washington

The High Level Picture

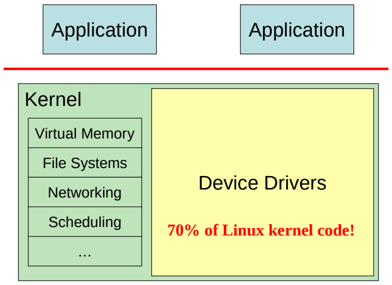
- A lot of research effort in the OS community has gone into *performance*, rather than reliability.
- The result: operating system crashes are still a huge problem today
  - 5% of Windows systems crash every day
- Device drivers are the biggest cause of crashes
  - Drivers cause 85% of Windows XP crashes
  - Drivers in Linux are 7 times buggier than the kernel

What is a Device Driver?

A module that translates high-level OS requests to device-specific requests

- 10s of thousands of device drivers exist
  - Over 35K drivers on Win/XP!
- 81 drivers running on this laptop
- *Drivers run inside the OS kernel*
  - A bug in a driver crashes the OS
- Small # of common interfaces

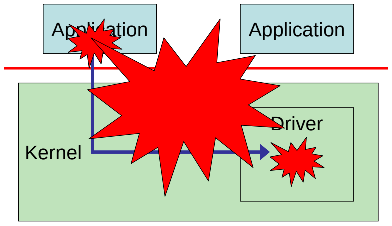
OS Today

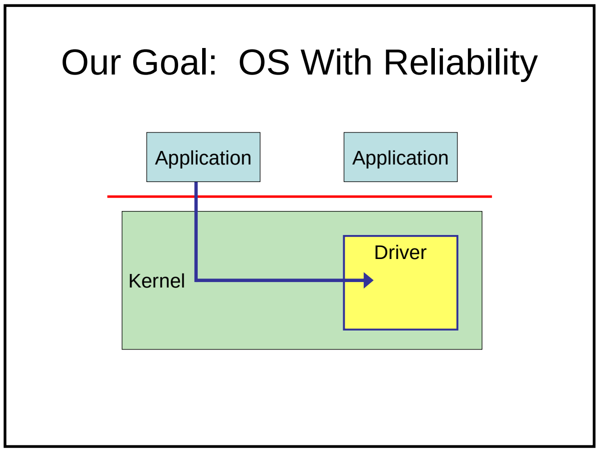


Why Do Drivers Fail?

- Complex and hard to write
  - Must handle asynchronous events
    - interrupts
  - Must obey kernel programming rules
    - Locking, synchronization
  - Difficult to test and debug
    - timing-related bugs
  - Non-reproducible failures
- Often written by inexperienced programmers
- Code often not available to OS vendors

OS Today





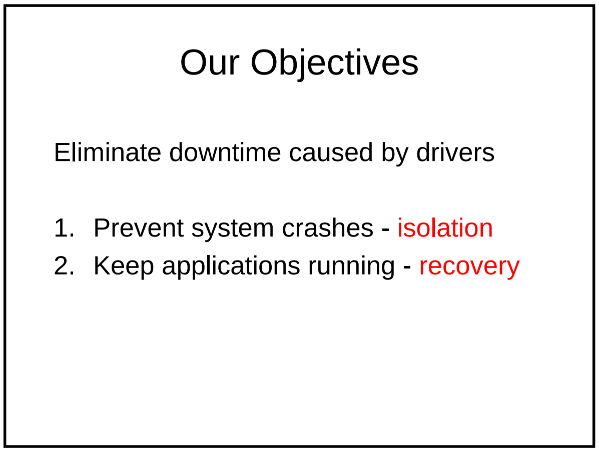
Our Goal: OS With Reliability

Application

Application

Kernel

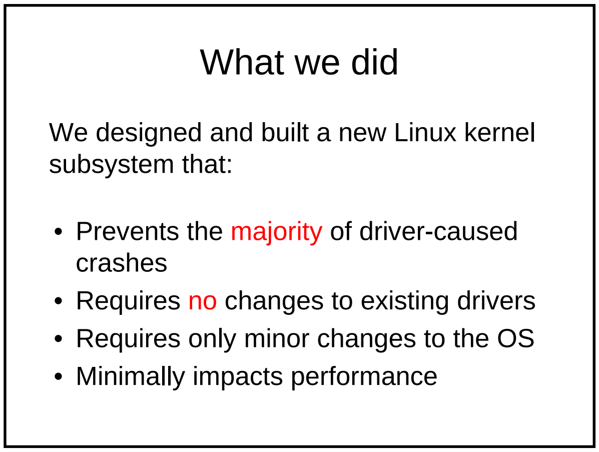
Driver



Our Objectives

Eliminate downtime caused by drivers

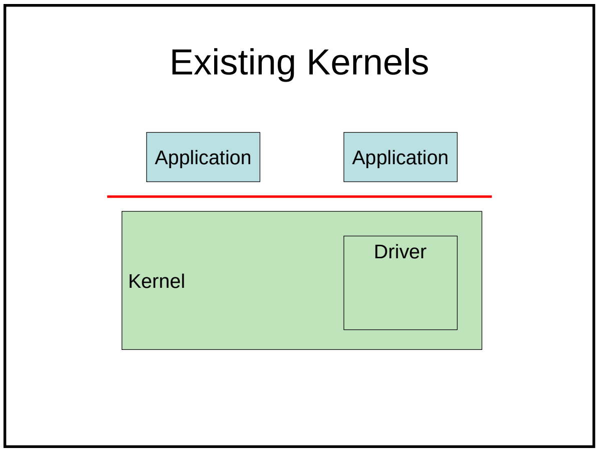
1. Prevent system crashes - isolation
2. Keep applications running - recovery



What we did

We designed and built a new Linux kernel subsystem that:

- Prevents the majority of driver-caused crashes
- Requires no changes to existing drivers
- Requires only minor changes to the OS
- Minimally impacts performance



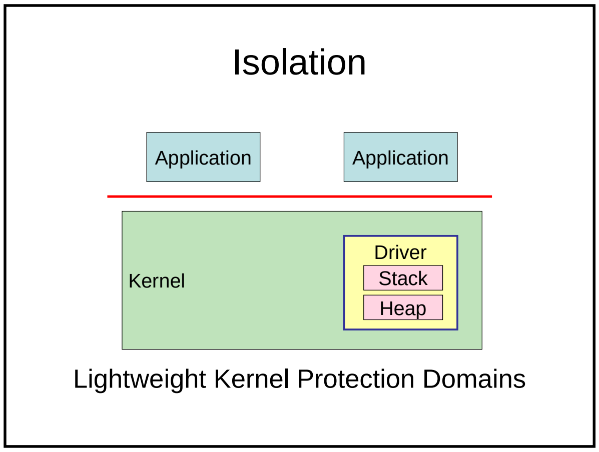
Existing Kernels

Application

Application

Kernel

Driver



Isolation

Application

Application

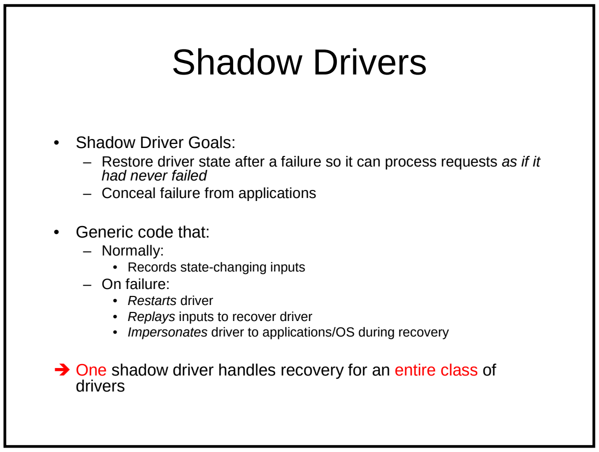
Kernel

Driver

Stack

Heap

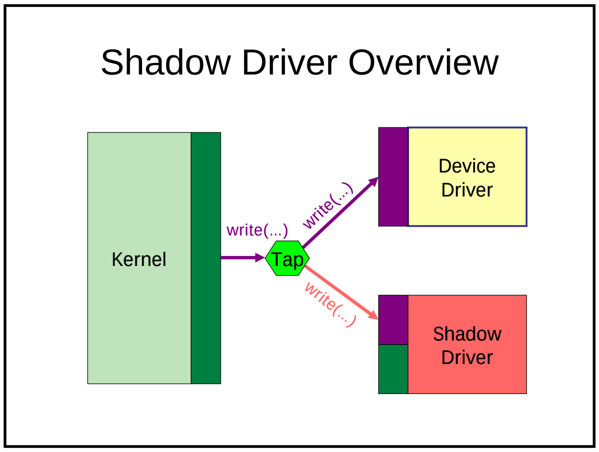
Lightweight Kernel Protection Domains



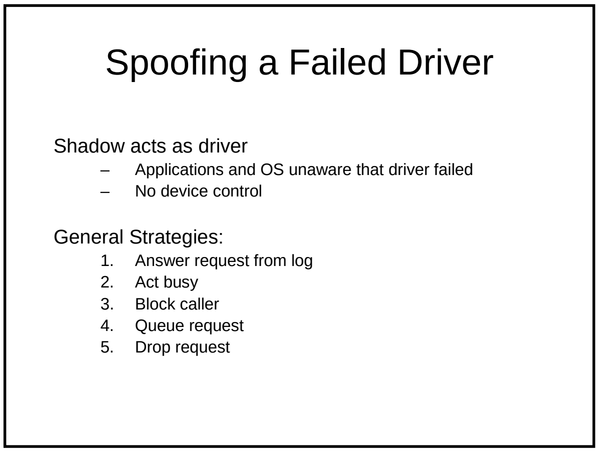
Shadow Drivers

- Shadow Driver Goals:
  - Restore driver state after a failure so it can process requests *as if it had never failed*
  - Conceal failure from applications
- Generic code that:
  - Normally:
    - Records state-changing inputs
  - On failure:
    - *Restarts* driver
    - *Replays* inputs to recover driver
    - *Impersonates* driver to applications/OS during recovery

➔ One shadow driver handles recovery for an entire class of drivers



Shadow Driver Overview



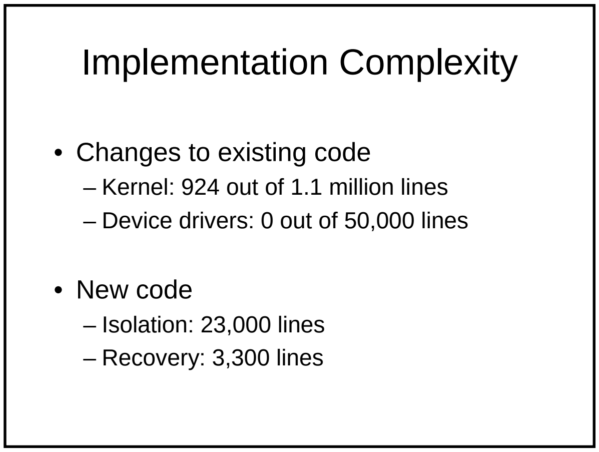
Spoofing a Failed Driver

Shadow acts as driver

- Applications and OS unaware that driver failed
- No device control

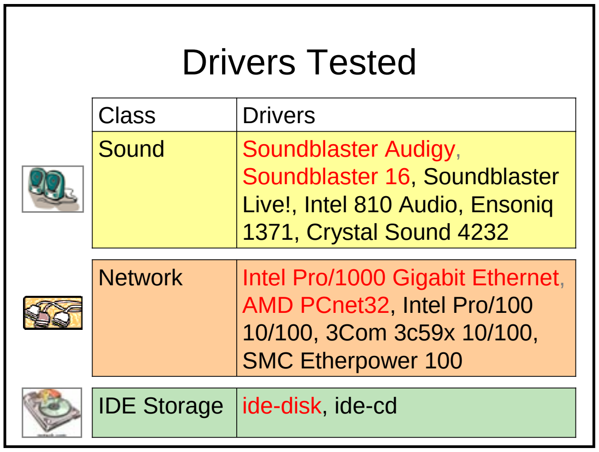
General Strategies:

1. Answer request from log
2. Act busy
3. Block caller
4. Queue request
5. Drop request



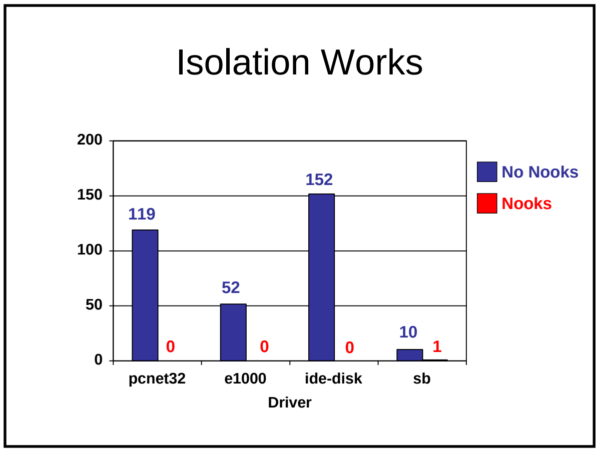
Implementation Complexity

- Changes to existing code
  - Kernel: 924 out of 1.1 million lines
  - Device drivers: 0 out of 50,000 lines
- New code
  - Isolation: 23,000 lines
  - Recovery: 3,300 lines

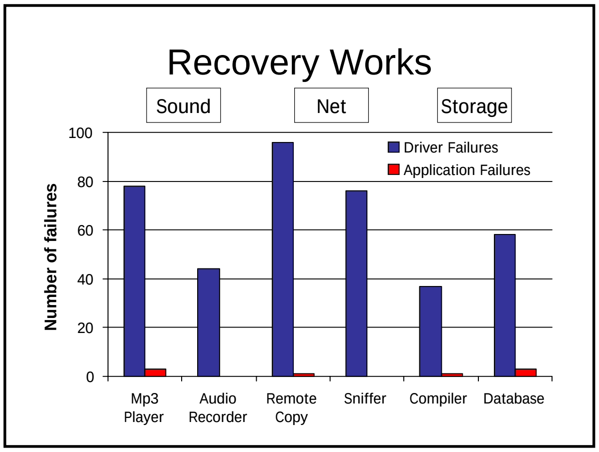


Drivers Tested

Class	Drivers
Sound	Soundblaster Audigy, Soundblaster 16, Soundblaster Live!, Intel 810 Audio, Ensoniq 1371, Crystal Sound 4232
Network	Intel Pro/1000 Gigabit Ethernet, AMD PCnet32, Intel Pro/100 10/100, 3Com 3c59x 10/100, SMC Etherpower 100
IDE Storage	ide-disk, ide-cd

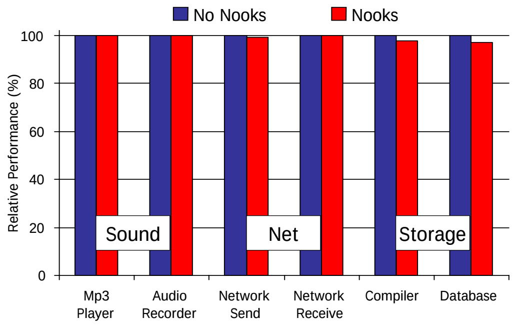


Isolation Works



Recovery Works

Relative Performance



Evaluation: Bottom Line

- Isolation works
  - We can avoid crashes in the majority of driver failures
- Recovery works
  - We can keep applications running in the majority of driver failures
- The cost is acceptable
  - In many cases, the performance cost is acceptable

Summary

- We took a very targeted and practical approach to improving reliability
- We defined a set of new components and techniques to create a new OS reliability layer
- We used these components to build isolation and recovery services
- Our experiments demonstrate that:
  - Nooks prevents 99% of the crashes caused by our tests
  - Nooks keeps applications running in 98% of tested driver failures
  - There is high leverage in this approach