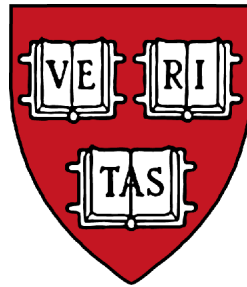


# CS161: Operating Systems

Matt Welsh  
mdw@eecs.harvard.edu



Lecture 23: Alternative OS Designs  
May 1, 2007

# Today: Alternative OS Designs

## What's wrong with traditional OS designs?

- Applications have a specific interface into the kernel
- e.g., TCP and UDP sockets, filesystem, virtual address spaces, etc.

## These abstractions are not always ideal!

- Databases prefer to manage layout of disk blocks directly
- But, FS mandates a particular block management policy

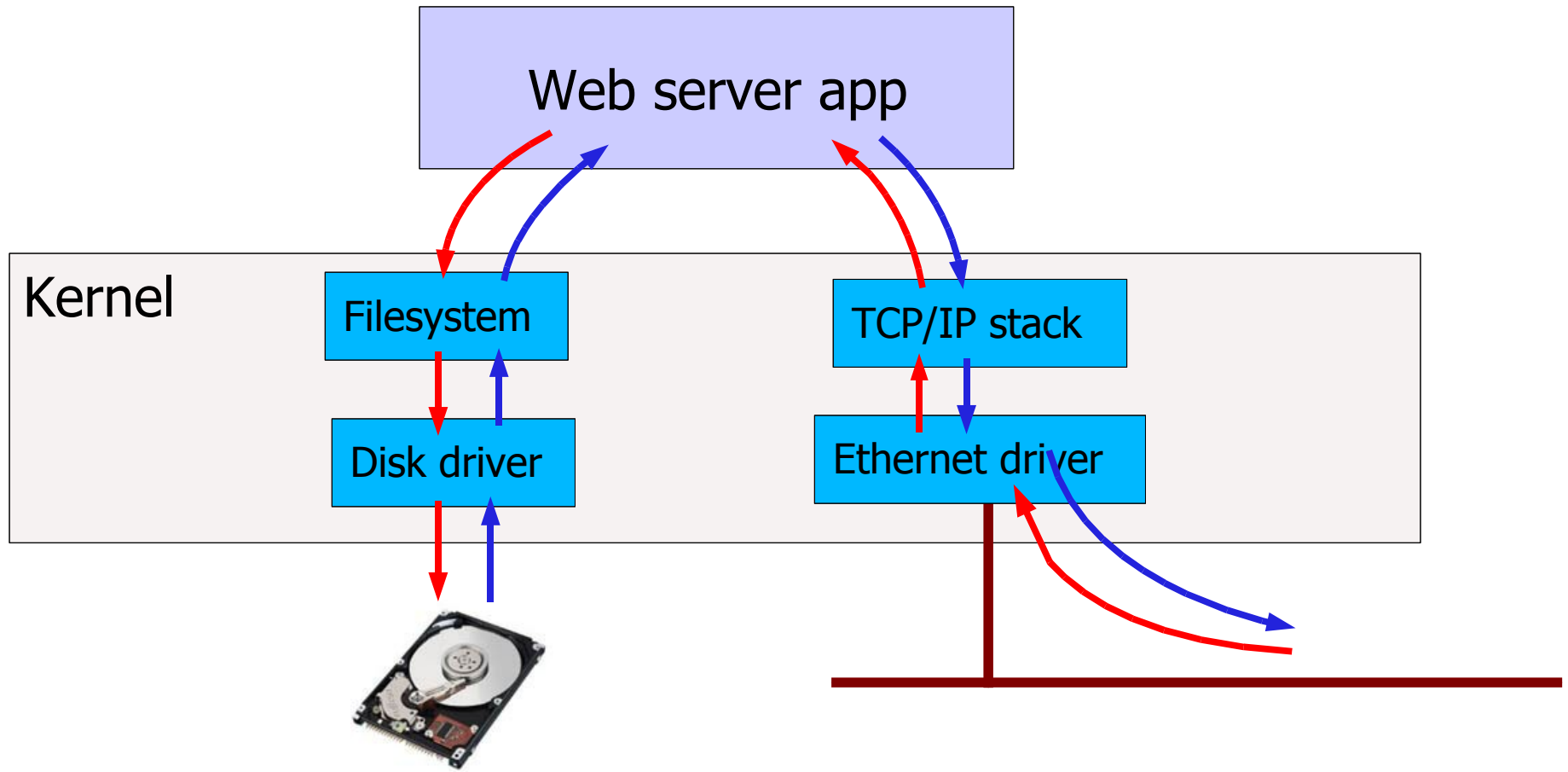
## Default implementations in the OS are not always optimal!

- Application might know more about its access patterns than the OS
- So, if the app could get control over how the OS does things, could significantly increase performance

## Today: SPIN, Exokernels, and Nemesis

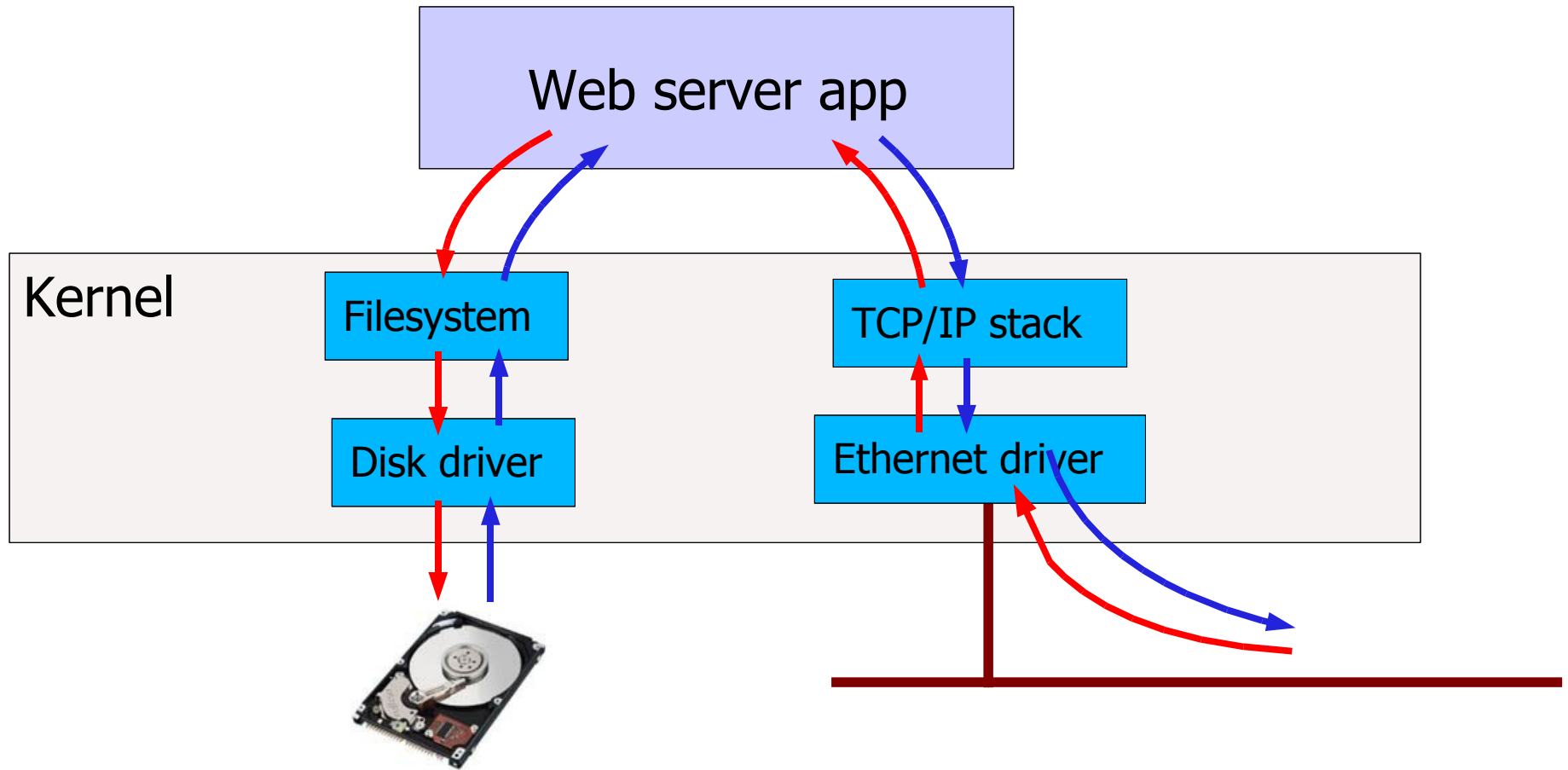
- Three “alternative” (research) OS designs meant to address these problems
- They look very different from the UNIX/Windows model that we have mostly been studying

# Web server example



What's wrong with this picture?

# Web server example



What's wrong with this picture?

- Data is copied multiple times from filesystem to Web server to network stack
- Application has no control over how buffer space is used or reclaimed
- Application cannot influence location of Web page data on disk

# Specialization

Application may wish to *specialize* the OS to achieve better performance

Examples????

- 
- 
-

# Specialization

Application may wish to *specialize* the OS to achieve better performance

## Examples:

- Specific file system cache management policy
- Specialized network protocol for video transport
- Eliminate extra copies of data from kernel to user and back (Web server example)

## Existing OS designs are very difficult to extend!

- One must typically write a new device driver or kernel module
- Kernel does not always provide extensible internal interfaces
  - *e.g., The page replacement policy might be hard coded ...*
- Once code is inside the kernel, it runs with full protection and can do essentially anything to the system

# Safe code downloading

One approach: Allow app to download *safe extensions* into the kernel

- Code should not be able to access resources that it is not allowed to touch
- Code should not be able to hang or crash the system
- Code should not be able to open up security holes

How to ensure that application-supplied code is “safe?”

# Safe code downloading

One approach: Allow app to download *safe extensions* into the kernel

- Code should not be able to access resources that it is not allowed to touch
- Code should not be able to hang or crash the system
- Code should not be able to open up security holes

How to ensure that application-supplied code is “safe?”

## Software Fault Isolation (SFI)

- Write OS extensions in any language (such as C)
- When downloading into the kernel, rewrite the machine code to insert checks
- Check for pointer access, jump instructions, etc. as they run
  - *Make sure program does not access memory it is not allowed to!*
- *Problem: High runtime overhead*

## Safe programming languages

- Write OS extensions in languages that cannot forge pointers, access arbitrary memory, or jump to arbitrary addresses
  - *Examples: Modula-3 or Java*
- *Moves safety check to compile time rather than run time!*

# SPIN Operating System

Research project at Univ. Washington, '95-'96

Kernel provides a large number of “hooks” for app extensions to use

- e.g., Each time a page fault occurs, a disk I/O completes, a thread is scheduled, etc.
- App extension can provide an *event handler* for each of these hooks

App extensions written in Modula 3

- Strongly typed, garbage collected language somewhat like Java
- All kernel objects accessed through type-checked *interfaces*

# Example: App specific thread scheduling

Say I have an app that wants to provide its own scheduler.

- Example: I'd like to give some user level threads higher priority than others.
- But, instead of using user-level threading, want the kernel to manage the threads
  - *e.g., to run on multiple CPUs.*
- But still want application control over the **scheduling policy**

The user-level scheduler needs to know when:

- A thread blocks (e.g., disk I/O or page fault)
- A thread becomes runnable (e.g., disk I/O completes)

The user-level scheduler is also responsible for:

- Picking the next thread to run
- Saving and restoring the CPU state of each thread

In SPIN, user would implement the “strand” interface

- Single global scheduling policy determines when a given app's **strand** gets to run
  - *Round-robin, preemptive, priority-based scheme*
- The app then decides which thread to run while its strand is active!

# Virtual Memory in SPIN

SPIN also allows apps to implement their own paging policy!

- Hook into physical page, virtual address, and translation interfaces
- **Physical page**: Allows physical frames to be allocated/deallocated
- **Virtual address**: Allows virtual address range to be allocated
- **Translation**: Controls mapping from virtual to physical address

Example: Implementing demand paging

- **Translation.BadAddress** and **Translation.PageNotPresent** events raised when an unmapped virtual address is accessed by the application
- **User-supplied page fault handler** can allocate new physical page and map it into the virtual address space
- User can decide how to pool memory, when to reuse pages, when to kick physical pages out, etc.

What happens if OS needs to reclaim a physical page from an app?

- User-supplied **PhysAddr.Reclaim** event is called
- Code can propose an alternate physical page to kick out (e.g., one that has not been used in a while)

# Issues with SPIN

Multiple apps can register “competing” event handlers!

Have to ensure that an event handler doesn't hog the system

- Requires timeout mechanism and ability to kill running handler

Have to guarantee fair resource allocation and behavior

- If every app has a different scheduling policy, what do you do?
  - *SPIN approach: Punt. Use a single global scheduler but let apps “do their own thing” while they are running*

Most performance benefit derived from tests with single event handler

- Could optimize the heck out of things by assuming a single handler, no need for setting timeout, or dispatching to separate thread
- When multiple event handlers are installed, overhead should go up considerably?

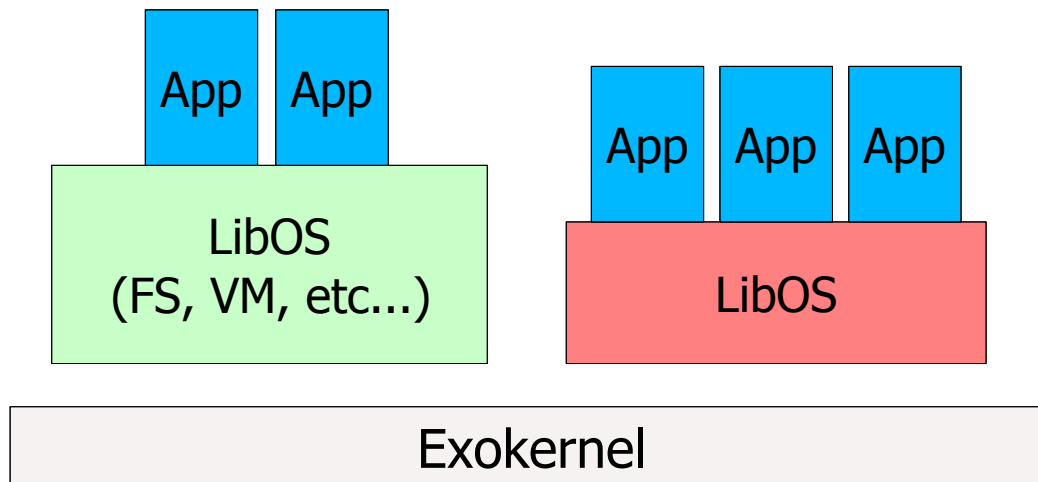
# Exokernels

## MIT project around '96-97

- Trying to solve similar problems as SPIN:
- Allowing applications to customize OS behavior

## Very different approach: *Separate protection from management*

- OS is only responsible for enforcing protection
  - *e.g., Making sure one app can't corrupt memory page of another*
- Applications implement *all* functionality to manage resources themselves!
  - *Generally done in a “Library Operating System” (LibOS)*



# Exokernels and LibOS's

## Exokernel itself just handles protection

- Hands out physical resources to applications
- Examples: Physical memory pages, disk blocks, CPU time
- Kernel does not attempt to *manage* these resources
  - *That is, decide how and when they are used*

## Most functionality implemented by LibOS

- Each application can have a different LibOS
- e.g., One LibOS for traditional UNIX apps, another for a specialized Web server

# LibOS Examples

## LibOS-based filesystem

- Uses exokernel low-level disk block read/write calls to perform I/O

## LibOS-based virtual memory management

- Exokernel provides calls to allocate and map physical pages
- When a page fault occurs, OS traps to user-supplied page fault handler
- Need to ensure that page fault handler does not run forever...

# Exokernel file systems

Would like to implement a filesystem as a library.

- What issues come up?

## Problem #1: Enforcing Protection

- Say an application running as user U tries to access file F.
- Should we trust the LibOS to do the protection check?
  - *No! The LibOS is part of the application!*

## Problem #2: Buffer management

- What if every LibOS/application had its own buffer cache?
- Lots of memory overhead for commonly-cached files.

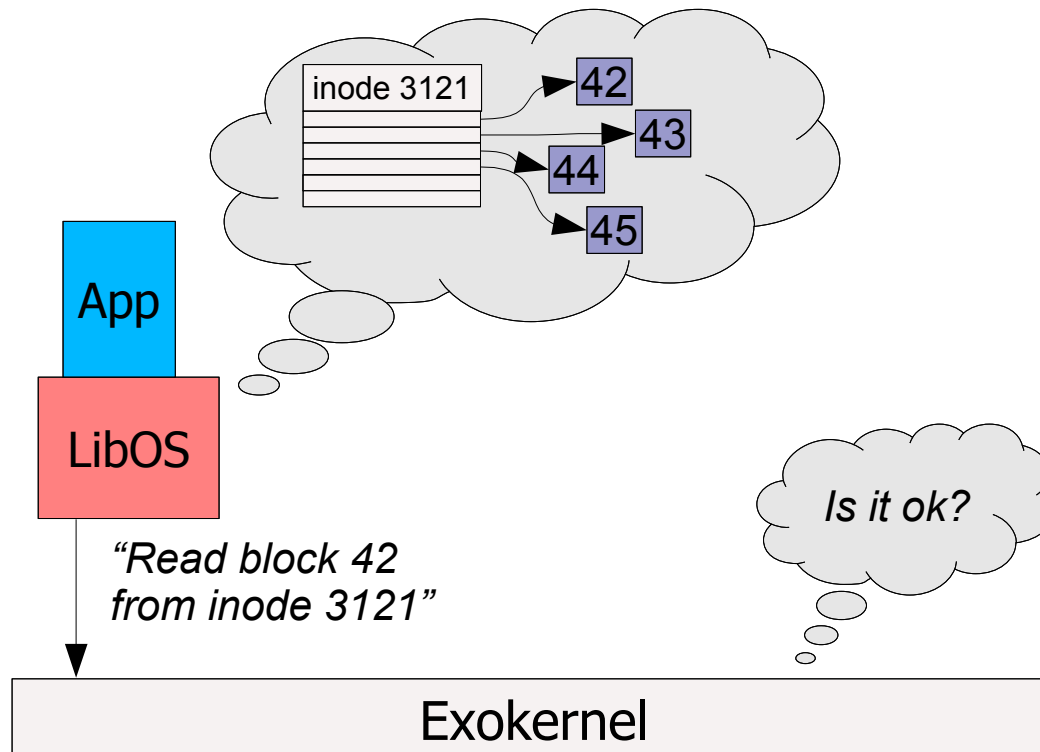
# Exokernel “XN” Storage System

Maintain a single, shared buffer cache in the kernel

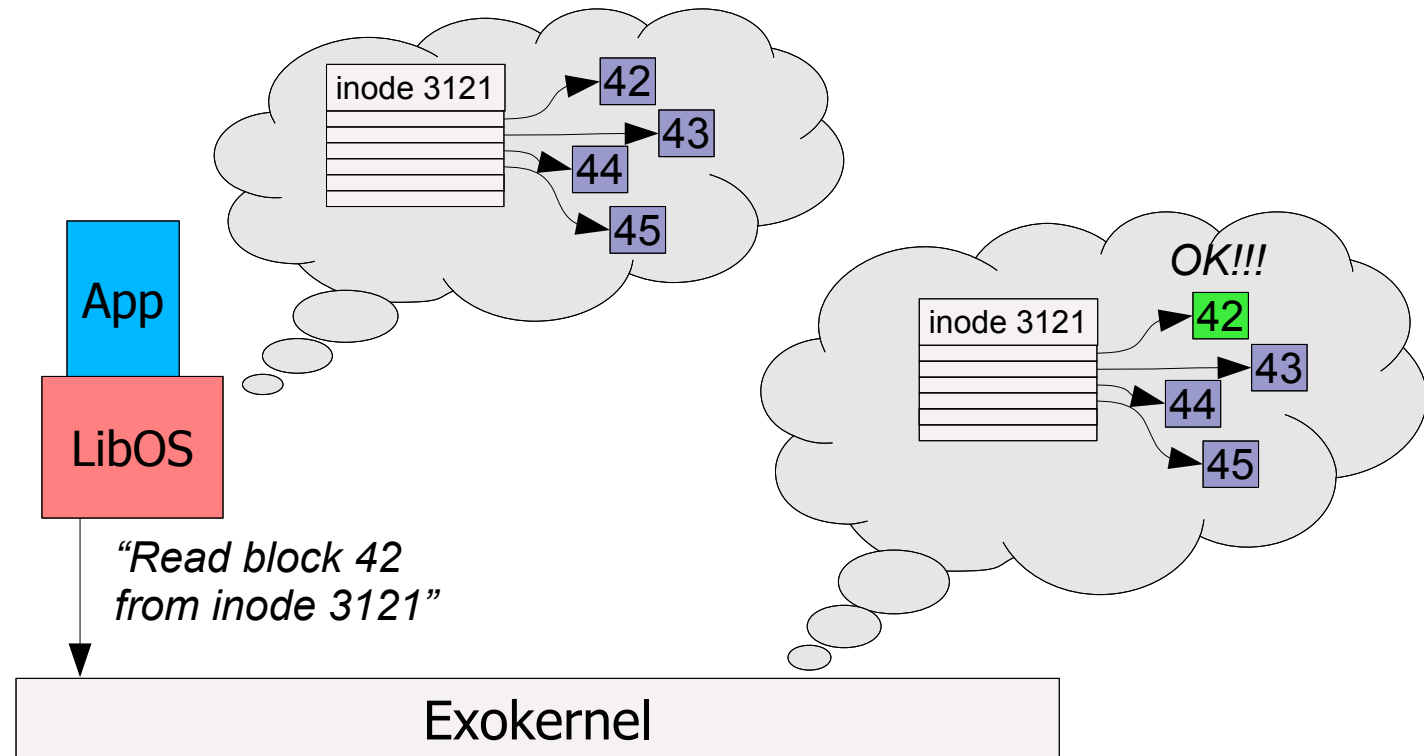
- Allows multiple LibOS's and users to share the same cache

Problem: How to enforce protection across multiple users?

- XN needs to know whether a given user is allowed to access a given block
- But ... the kernel doesn't know anything about the filesystem format!



# Exokernel “XN” Storage System



One approach: tell the kernel something about the filesystem

- If XN understands the inode format it can do the protection check itself

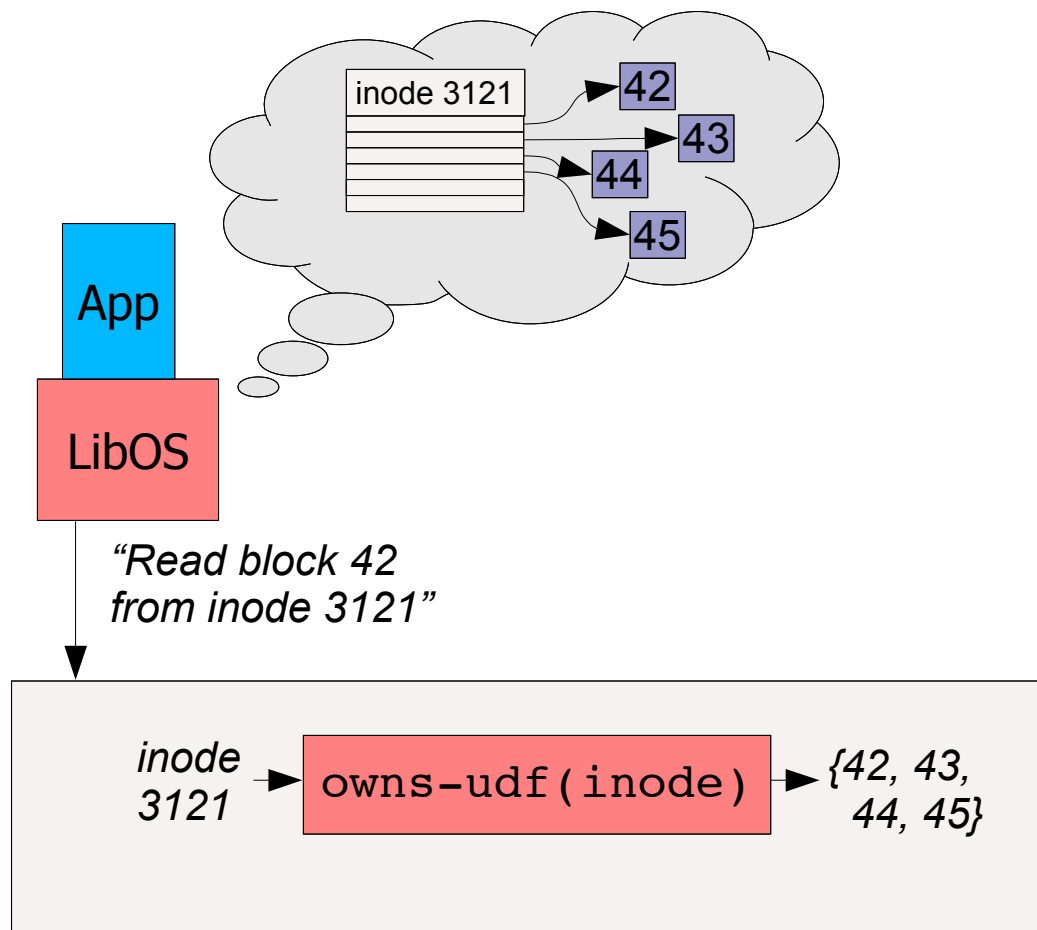
However, this is pretty restrictive!

- Means the kernel needs to know a lot about the filesystem semantics
- Ideally, kernel doesn't know **ANYTHING** about the specific filesystem being used!

# Untrusted Deterministic Functions

Idea: Push code into kernel that kernel can use to check that user accesses are safe.

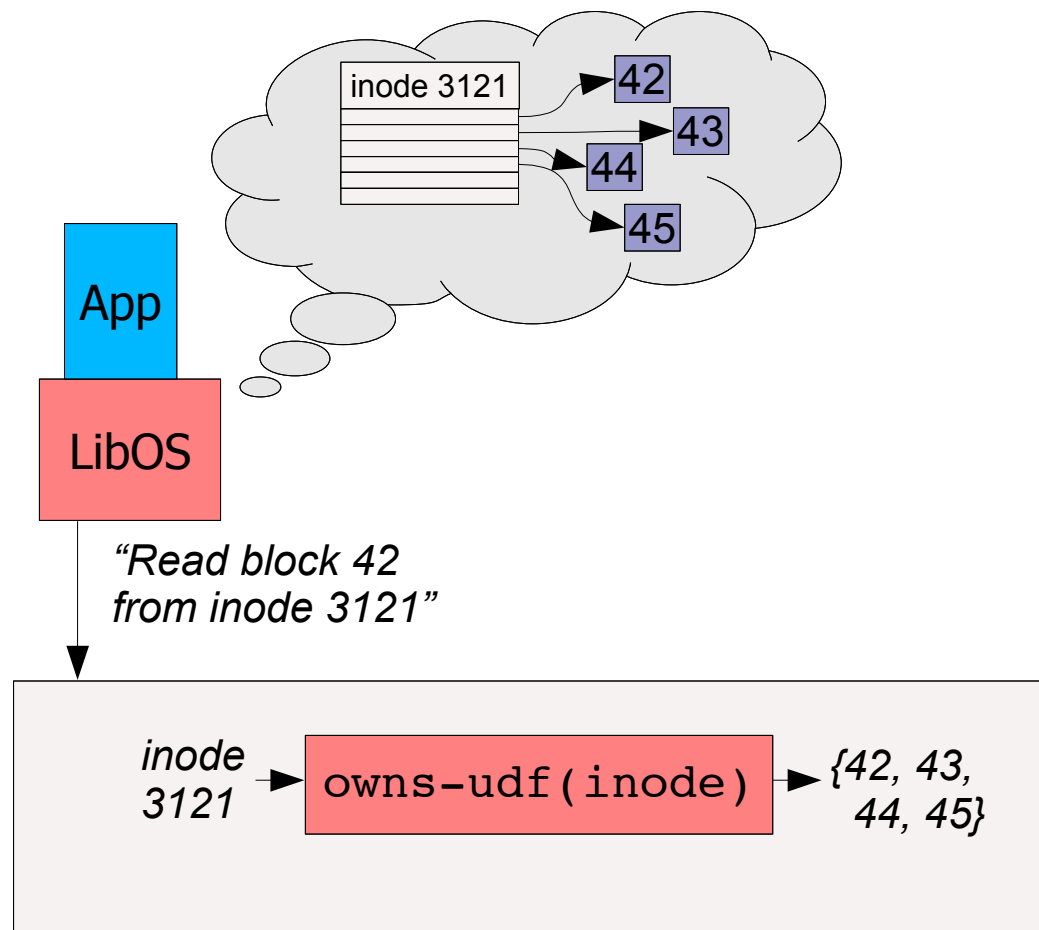
- But how do we trust that the function does the right thing (i.e., does not lie)?



# Untrusted Deterministic Functions

The UDF function is **deterministic**

- Implemented in a restricted language that guarantees its output is only a function of its inputs (and no other “external” state).
- So, given the same input inode, `owns-udf(inode)` always returns the same set of blocks.



# Why do UDFs work?

The UDF can't lie about which blocks are in a file. Why not?

Couldn't you write a UDF that just returns every block in the system regardless of the inode?

Answer: Kernel also checks that UDF does the “right thing” when LibOS requests to *add blocks* to an inode.

- Before adding a new block B, XN calls `owns-udf(inode)` and gets the set of current blocks
- LibOS is then allowed to muck with the inode and add a new block to the file
- XN calls `owns-udf(inode)` again, and checks that block B has been added to the list
  - *Net result: LibOS can't add arbitrary blocks to the set that it's allowed to access!*

Why does this work?

- Because the UDF is **deterministic**: its output only depends on the input file (and nothing else!)
- This is like a proof by induction.

# SPIN and Exokernel retrospective

Both SPIN and Exokernel are about improving performance by permitting OS customization.

But, they take very different approaches:

- SPIN allows user to customize kernel, but requires safe language
- Exokernel is about stripping all functionality out of kernel, moving it to user level

This work is great, but shows how hard it is to open up OS internals in a safe and efficient manner.

- Example: Exokernel needed a shared buffer cache **in the kernel** after all

# Nemesis: OS for multimedia apps

Research project at Univ. Cambridge, '95-'98

Main goal: Make it possible to run multiple competing real-time multimedia apps on the same hardware!

- e.g., A server sending multiple real time video feeds to clients
- A client watching multiple video/audio streams on the display

## Quality of Service (QoS)

- General term used to define the requirements of an application

## QoS Examples:

- *Minimum frame rate should be 30 frames a second*
- *Application should run once every 50 ms for at least 10 ms*
- *The jitter between consecutive scheduling opportunities should be less than 5 ms*

# QoS Crosstalk

Would like to “reserve” some amount of resource (e.g., CPU time) for each application so it can meet its QoS requirements.

Problem: Not all CPU time is properly accounted for!

- e.g., When TCP stack is processing incoming packets, the CPU time is being used by the kernel, not any specific application
- Same goes for handling file system requests, disk I/O, page faults, etc.
  - *It is sometimes very hard to tell which app the OS is doing work for!*

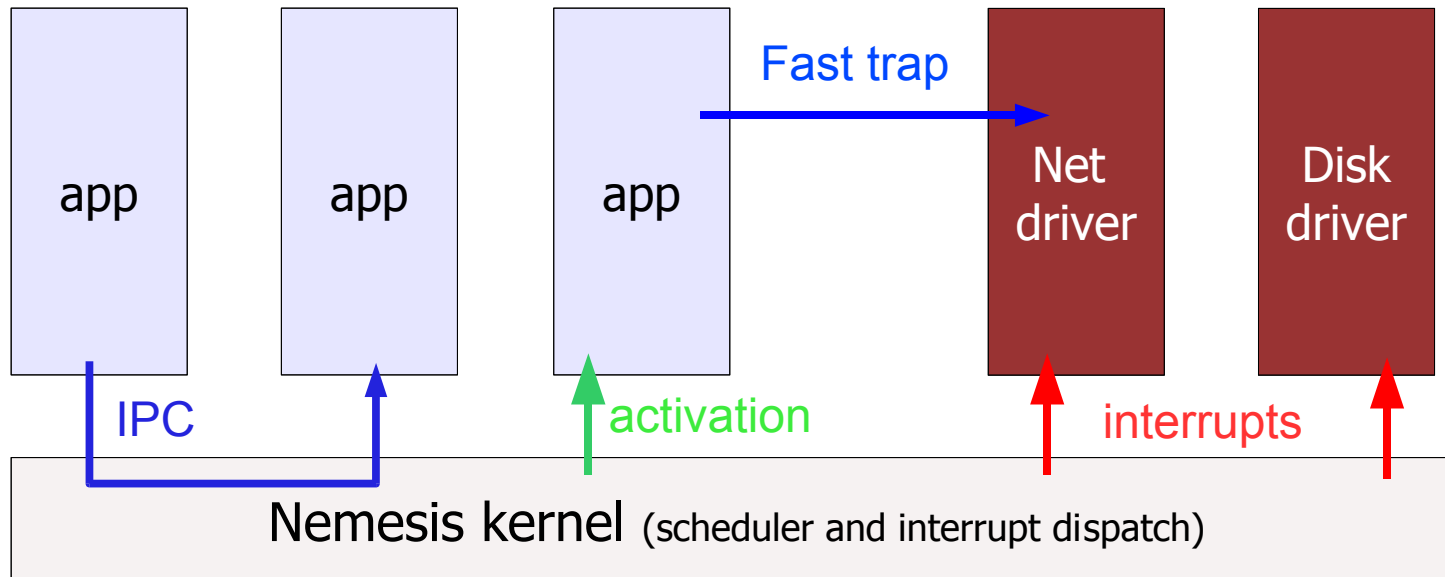
Result: Activity of one application can impact performance of another

- This is called **QoS Crosstalk**
- Applications are not properly isolated from one another

# Nemesis Approach

Very much like Exokernel: Implement all functionality at user level!

- Applications, device drivers, etc. all run at user level
- **All** CPU time is accounted to the appropriate **domain**



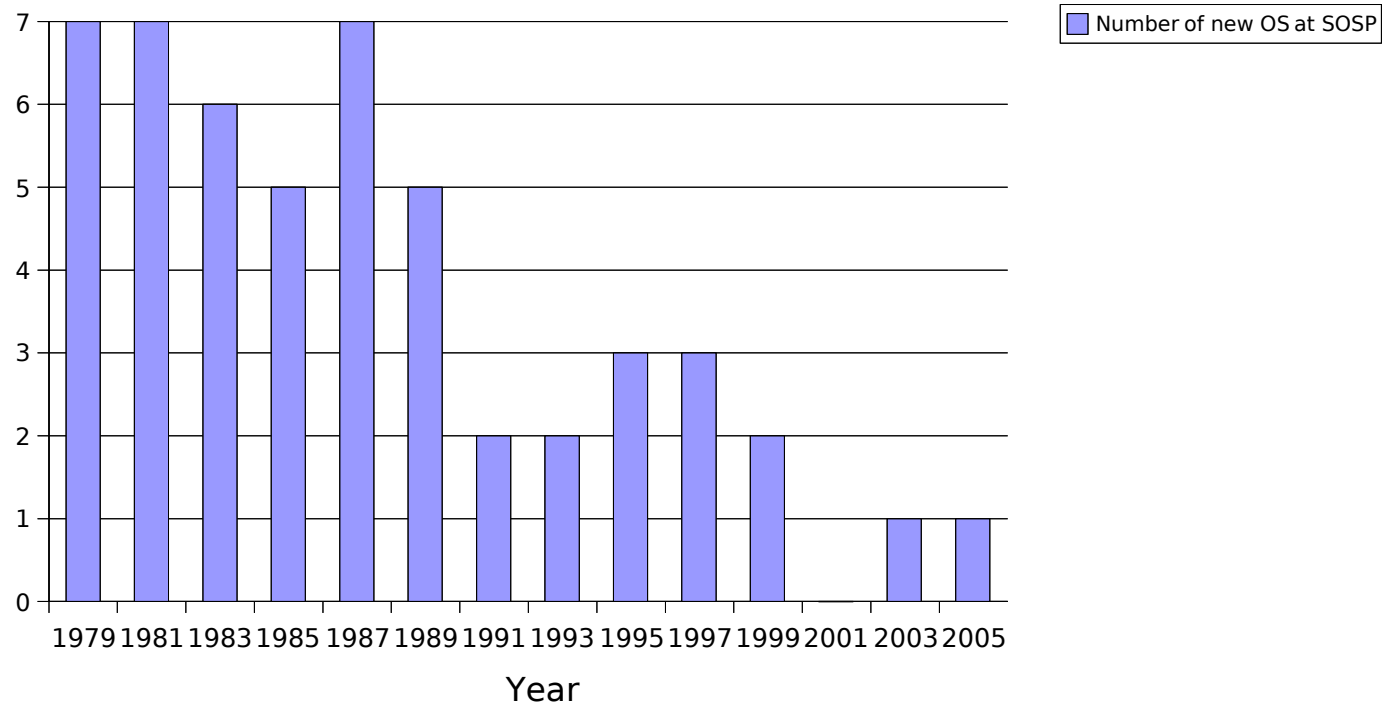
- Domain is scheduled by the kernel with a *processor activation*
- Domains can communicate using (shared memory) *IPC* through kernel

IPC may be expensive, and charges CPU time to the receiving domain.

- How do we allow a user app to access state in a protected (“kernel”) domain?
- Solution: *Fast trap* (Kind of a hack...)

# Where is OS Research Going?

Number of new OS designs presented at SOSOP, last 27 years:



Systems research no longer focused on new OS designs!

- Main focus not even on kernel features or performance
- Lots of papers on network-based systems: P2P, distributed applications, etc.
- Lots of work on new services, e.g., automatic performance tuning, finding bugs in OS code, stopping worms and viruses, etc.

Partly due to prevalence of Linux as an OS research platform