Rump File Systems
Kernel Code Reborn

Antti Kantee
pooka@cs.hut.fi
Helsinki University of Technology

USENIX Annual Technical Conference,
San Diego, USA
June 2009

Introduction

- kernel / userspace dichotomy
  - interfaces dictate environment
- make kernel file systems run in userspace in a complete and maintainable way
  - full stack, no code forks or `#ifdef`
- file system is a protocol translator
  - read(off,size,n) => blocks 001,476,711,999

Implementation status

- NetBSD kernel file system code runs unmodified in a userspace process
- total of 13 kernel file systems
  - cd9660, efs, ext2fs, fat, ffs, hfs+, lfs, nfs, ntfs, puffs, sysvbfs, tmpfs, udf
  - disk, memory, network, ”other”
- implementation shipped as source and binary with NetBSD 5.0 and later

Terminology

rump: runnable userspace meta program

1) userspace program using kernel code

2) framework which enables above

rump kernel: part of rump with kernel code

host (OS): system running the rump(1)

Talk outline

motivation

use cases

implementation

evaluation

Motivation

- original motivation: kernel development
- additional benefits:
  - security
  - code reuse in userspace tools
  - kernel code reuse on other systems

Contrasts

1) usermode OS, emulator, virtual machine, second machine, turing machine, etc.
   - acknowledge that we already have an OS
   - vm simplifications, abstraction shortcuts, etc.
   - direct host service (no additional userland)
2) userspace file systems (e.g. FUSE)
   - reuse existing code, not write new code against another interface

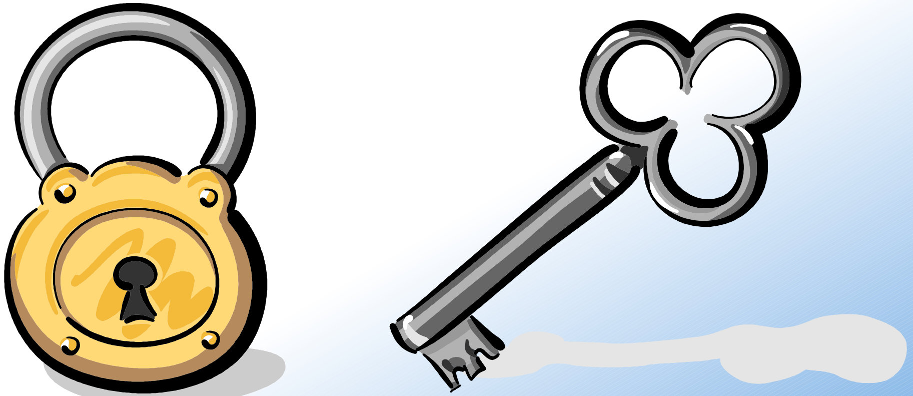
Talk outline

motivation

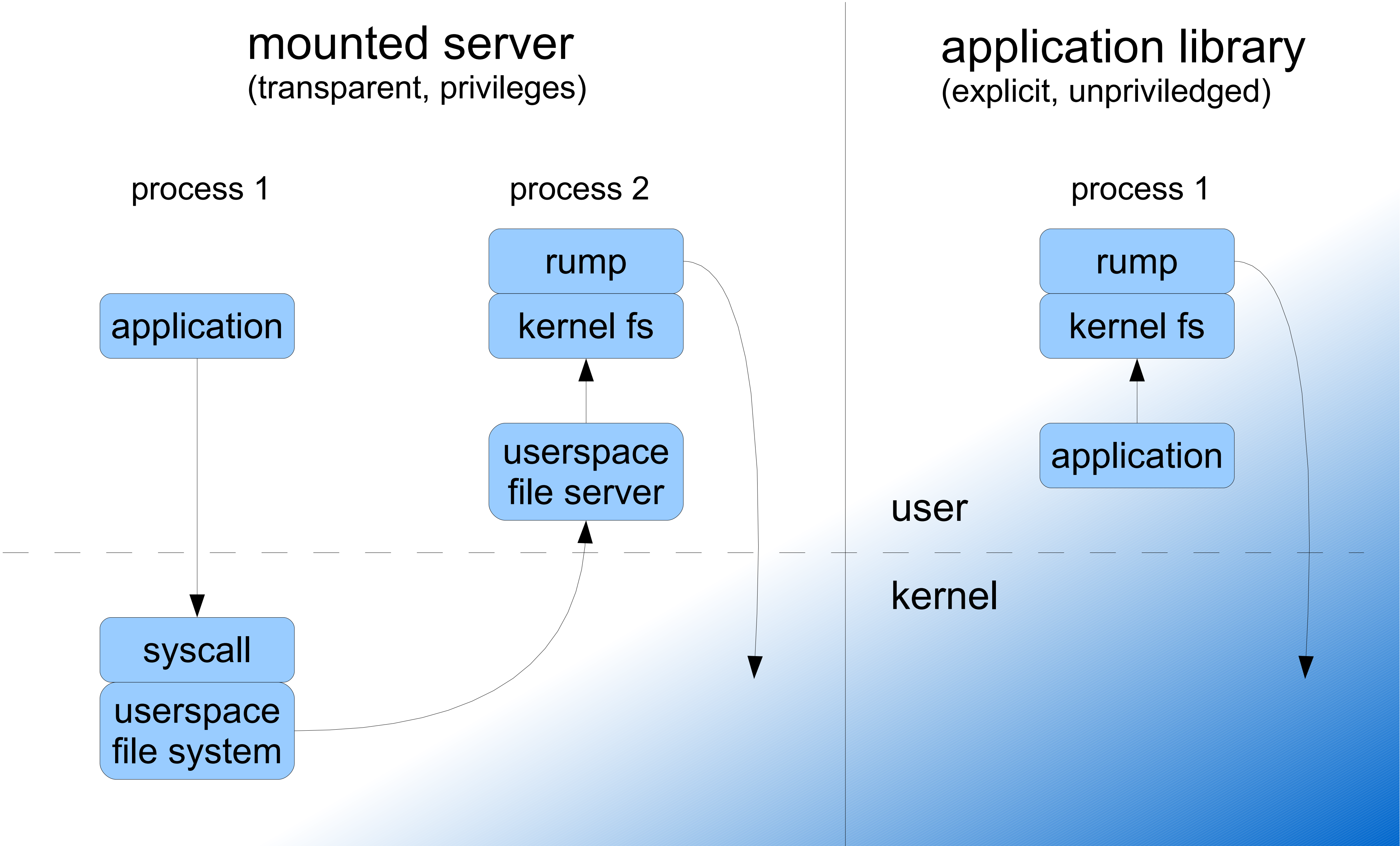
use cases

implementation

evaluation



Two modes



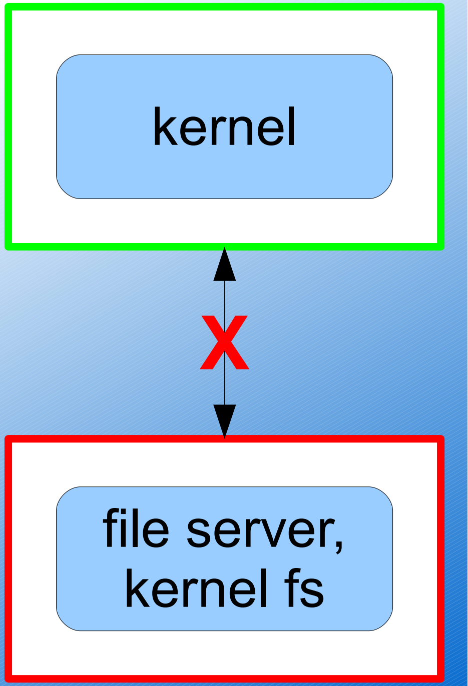
mounted server
(transparent, privileges)
application library
(explicit, unpriviledged)
process 1
process 2
process 1
rump
kernel fs
application
userspace
file server
rump
kernel fs
application
user
kernel
syscall
userspace
file system

Security

- common scenario: you get a USB stick from a 3rd party
- plug stick into your system and attempt to either read or write files
- suitably corrupt file system: crash
  - or worse
- mount as rump file system: isolate damage to a process

Security

- common scenario: you get a USB stick from a 3rd party
- plug stick into your system and attempt to either read or write files
- suitably corrupt file system: crash
  - or worse
- mount as rump file system: isolate damage to a process



Development & Debugging

- kernel hacking is ... convoluted
  - kernel hackers already know this
- rump allows more details from non-expert user submitting bug report
  - very important in an open source context
  - users are rarely willing or in a situation to setup a kernel development environment

Tools: fs-utils

- mtools, ntfsprogs, ltools, etc.
  - self-contained apps to access fs in userspace
  - different syntax
  - file system driver reimplementation
- fs-utils (Ysmal 2008)
  - standard POSIX utilities, rump fs driver
  - examples: `fsu_ls /dev/sd0e -ltr`
  - `fsu_mkdir fs.img -p /my/hier`

Tools: makefs

- problem: create an installation image
  - crossbuild => cannot use in-kernel fs & mount
- solution: makefs (Mewburn 2001)
  - application which creates a file system image from a directory tree
  - modified copy of the FFS driver, >100h effort
- rump makefs uses fs-utils
  - more supported file systems, much less effort

Talk outline

motivation

use cases

implementation

evaluation

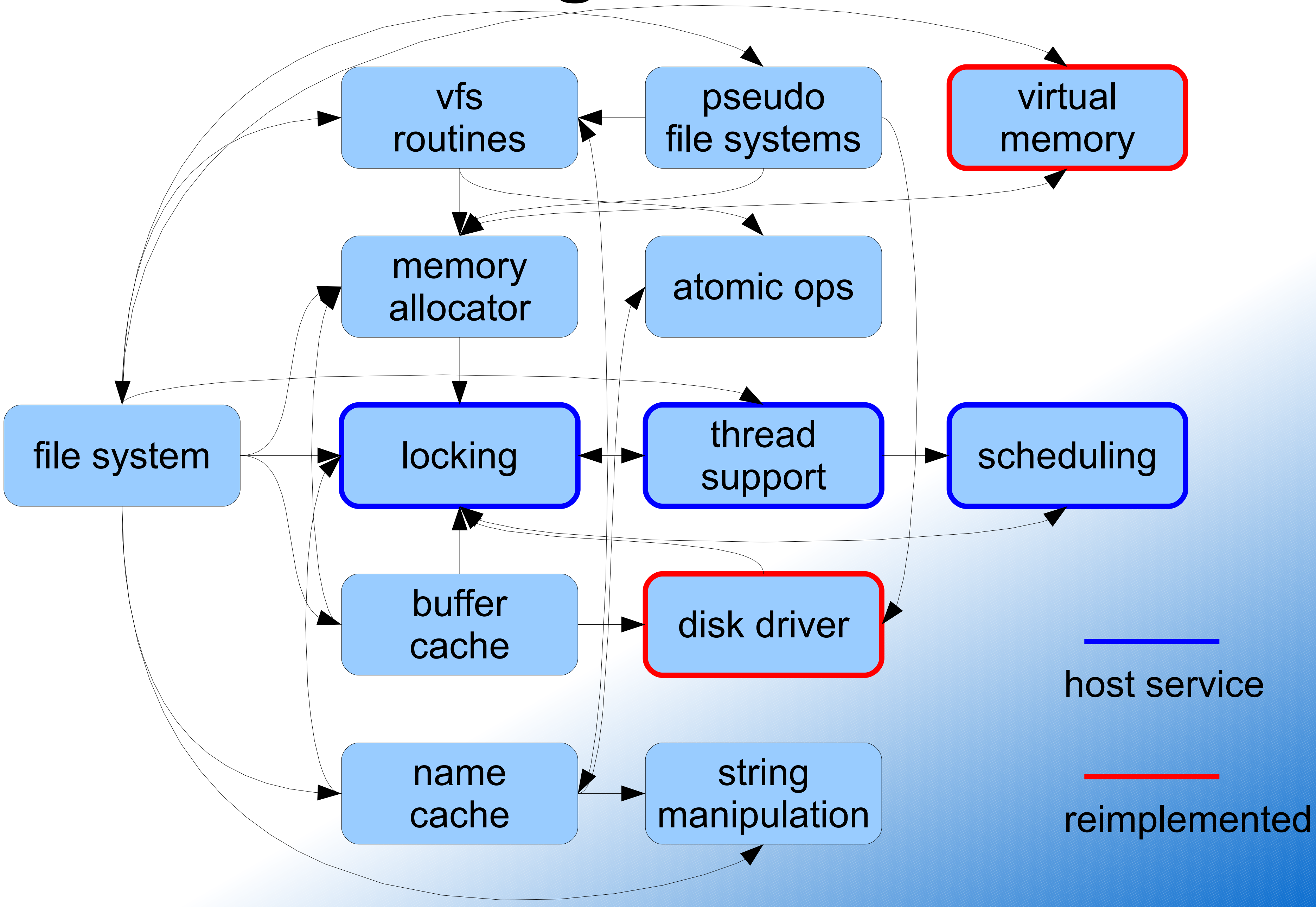
Kernel Code

- almost all kernel code can run anywhere
- set of interfaces code depends on determines default environment
  - `malloc()` vs. `kmem_alloc()`
- dependees have own set of dependencies
- need to find dependency closure starting from file system code

Implementation Strategy

- use as much kernel code directly as is practical
- use high-level services provided by host
  - threading, synchronization, sockets, etc.
- alter system structure: regroup source modules to minimize dependency hazards
  - but avoid getting yelled at

Finding Closure



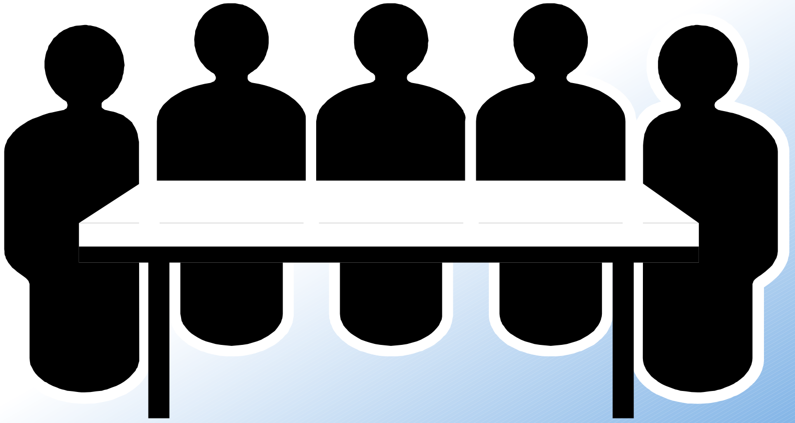
Talk outline

motivation

use cases

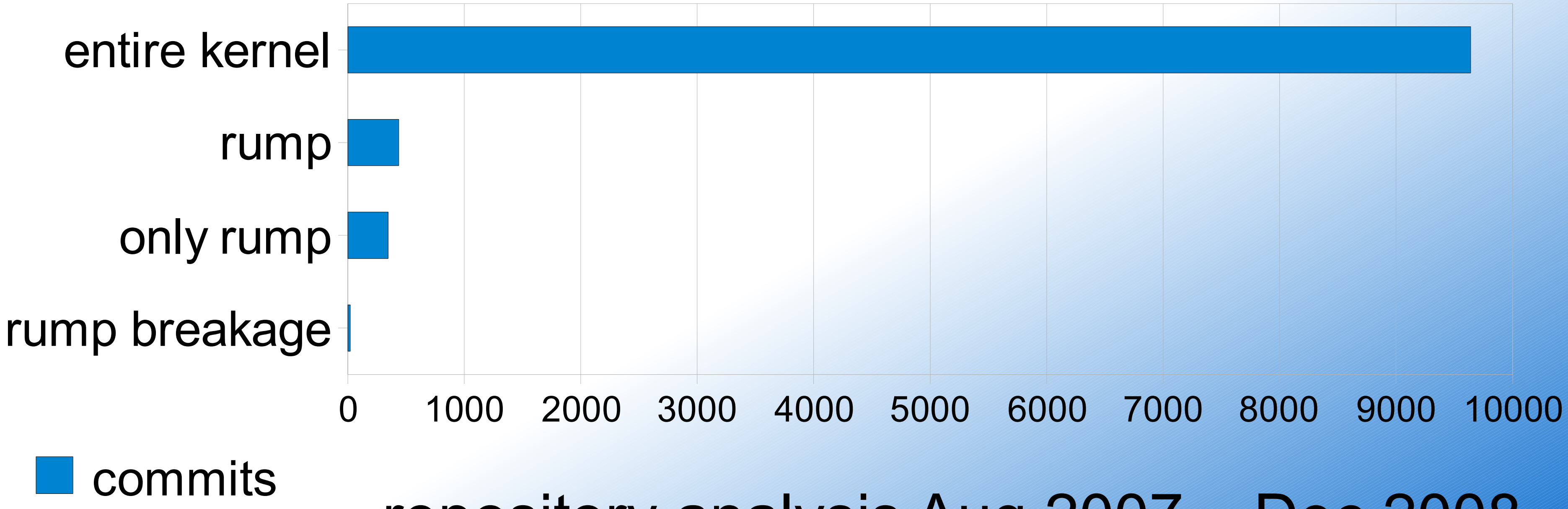
implementation

evaluation



Maintenance

- implementation duplicates interfaces and relies on module boundaries
- how often does it break?



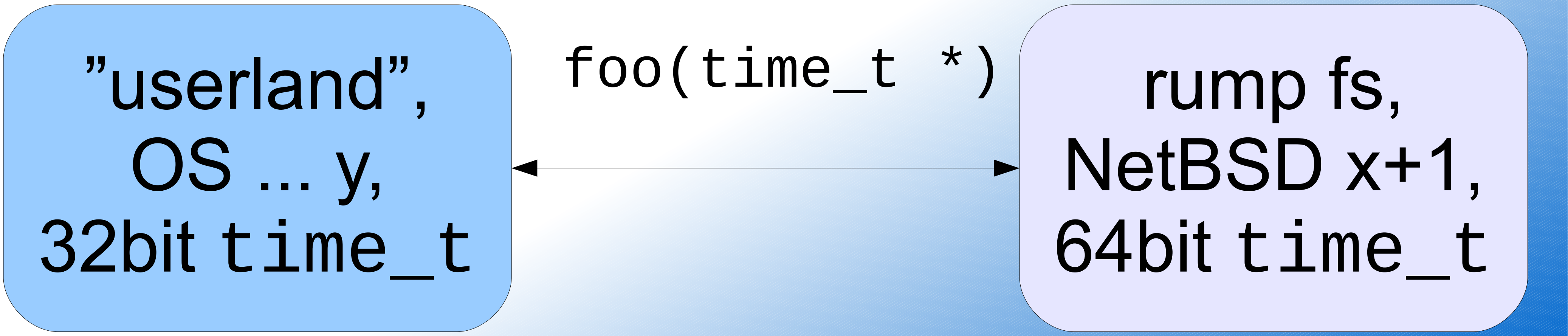
repository analysis Aug 2007 – Dec 2008

Portability

”C is portable”

”userspace programs are portable”

- NetBSD and Linux fs, mix&match
- there are details to take into account
  - data types need to match



”userland”,
OS ... y,
32bit `time_t`

`foo(time_t *)`

rump fs,
NetBSD x+1,
64bit `time_t`

Performance

- current approach: enhance performance only *inside* rump
  - do not modify the host system to provide non-standard interfaces
- for ultraperformance, use in-kernel mount
- common rump performance for FFS is ±5% of kernel mount performance
  - depends on backend

Conclusions

- possible to run kernel fs code of a general purpose OS in userspace
  - alter system structure
- benefits
  - avoid reimplementations
  - security
  - and kernel development
- implement it on $YourOS ;-)

Try it out!

- go to http://www.NetBSD.org/
- download 5.0 or -current & install
  - or use LiveCD
- man rump
- submit bug reports

Try it out!

- go to http://www.NetBSD.org/
- download 5.0 or -current & install
  - or use LiveCD
- man rump
- submit bug reports

Questions?