Kernel Development in Userspace
The Rump Approach

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

BSDCan 2009,
Ottawa, Canada
May 2009
Introduction / Motivation

- computers are difficult
  - otherwise this would be a boring conference
- kernel hacking is even more difficult
  - very unforgiving
  - lots can go wrong
  - everything can touch anything
- **motivation**: make this easier
  - at least on NetBSD ;-)


Talk outline

- survey of kernel development techniques
- introduce Runnable Userspace Meta Programs (rump)
- explain why, how, when and when not
- go into some details, provide tips
- introduce some useful tools
- **goal**: give ideas for alternative and easier approach on kernel development
Traditional development vectors: directly on hardware

- the hardcore approach
- typically two machines are used: one for development, one for testing
- environment setup may take a while
  - installing *and* maintaining two systems
- typically serial console/firewire + gdb
- sometimes the only feasible option
  - some stages of device driver development
Traditional development vectors: emulator or virtual machine

- two popular examples: qemu and Xen
- fundamentally same as direct approach
- no Xen dom0 already available?
- no KVM support => qemu is slow'ish
- benefit over hardware approach: no cables necessary
Traditional development vectors: ad-hoc userspace techniques

- massage component under development to run as part of a userspace program
- compile and run in userspace
- simplify/ignore some kernel interfaces
- apply `#ifdef` liberally
- file systems are well-known employers of this strategy, e.g. FFS and ZFS
Runnable Userspace Meta Programs

• **observation**: most kernel code does not need to run in the kernel (cf. microkernels)
• make kernel code runnable in userspace
• kernel source module x still depends on interfaces provided by modules y, z and å
• some code depends on hardware access
• **solution**: split code into components to handle dependencies, reimplement code unnecessary for userspace (e.g. pmap)
The (big) picture

TCP/IP driver → NIC driver \(\xrightarrow{\text{sendmsg()}}\) network process

NetBSD Kernel

NIC driver \(\xrightarrow{\text{write()}}\) rump network process

disk driver \(\xrightarrow{\text{read()}/\text{write()}}\) rump storage process

UFS driver

system call layer

network

hard drive
A different view

In Front

kernel

rump

function call

application

proc 1

Behind

kernel

service loopback

syscall

application

proc 1

server

rump

proc 2

user

kernel

function call

application

proc 1
Available components

- kernel core (rumpkern, -lrump)
- most file systems (rumpvfs, -lrumpvfs)
  - FFS (-lrumpfs_ffs), NFS (-lrumpfs_nfs),
    tmpfs (-lrumpfs_tmpfs) etc.
- networking (rumpnet, -lrumpnet)
  - networking subroutines (-lrumpnet_net)
  - TCP/IP (-lrumpnet_INET)
- system calls for each component
When to use

- debugging a supported component
- developing new code
- testing
- playing around

- application uses
  - beyond the scope of this presentation
When to not use

- rump *complements* existing methods
  - not a general solution
- desired component not available
  - you might attempt to add support, though
- desired component depends on interaction with unsupported component
  - e.g. virtual memory and page remapping
- desired component uses hardware directly
gdb and rump

• gdb can be used on a rump like on any userspace program
• backtrace, break, single-step, examine data, examine core dump, ...
• gdb on threaded programs currently suboptimal on NetBSD
  – env variable RUMP_THREADS set to 0
     disallows threads creation in rump
Examples of other tools

- gprof
- valgrind
- eletricfence, dmalloc
  - depends on which implementation of the memory allocator you use
File system tools

rump\_\$fs

- userspace mount
  - uses puffs

- *any* application

fs-utils (by Arnaud Ysmal)

- use rump syscalls to implement POSIX file system utils (ls, cp, etc.)
- does not require mounting
Demystifying *rumpns*

- kernel and application linker namespaces are normally disjoint
  - e.g. `malloc()` can exist in both without conflicts
- rump stuffs both under the same roof
  - possibility of conflicts
- => kernel symbols prefixed with "rumpns"
- linker complains: no `rumpns_garven_deh`
  - missing `garven_deh`, not `rumpns_garven_deh`
Interfacing with the "kernel"

- kernel function prototypes not available directly in userspace
  - they would be wrong anyway due to rumpns
- rump interfaces (e.g. `rump_init()`)  
- syscalls (e.g. `rump_sys_open()`)  
- vfs/vnode (e.g. `RUMP_VOP_SETATTR()`)  
- user-defined, must provide "rumpns" prototype for the compiler
ABI mix&match

- possible to run rump on non-matching NetBSD system version
- also possible on non-matching OS
- problem is interfacing
  - types with different size
Link sets

- entries placed in a certain section of the object file are unified by the static linker
- the kernel can traverse the entries runtime
- **problem:** scheme not fully compatible with dynamic shared objects
- **effect:** link set entries only from first DSO on linker line are visible
- **solution:** traverse link sets manually
Networking stack testing

• generate complex routed networks within a single machine
  – scales to thousands of nodes
  – script to generate&configure routing tables, interface addresses, etc.

• convert test applications to use rump
  – e.g. Apache took an hour or so
  – no preexisting tools yet
Tests and regression tests

- kernel tests usually run against live kernel
- test crash can crash the system
  - bad for batch testing
  - even worse for fault injection
- no need to: 1) compile kernel 2) update target environment 3) boot kernel 4) boot userland 5) run test program
  - very rapid incremental development
Repeating problems in rump

• most kernel problems easily repeatable
  – based on experience
• really sensitive timing problems might be problematic
  – or they might not be
• kernel bug or rump bug?
Example: real life fs problem

- mkdir returned ENOSPC with >4TB free
- solution:
  - mount with rump_ffs
  - put a breakpoint into ufs_mkdir
  - single-step and locate problem, fix
- rump enabled debugging the problem on a production system by a non-fs developer
Conclusions

- rump helps kernel development in its target cases
  - complements traditional methods
- short test cycle
- userspace tools
- makes the kernel more approachable
  - allows users to submit better bug reports
More info

- `src/sys/rump` in NetBSD source tree
- NetBSD mailing lists
- BSDCan 2009 paper
  - and other papers

- questions?