Kernel Development in Userspace - The Rump Approach

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

Abstract

In this paper we explore the use of the NetBSD Runnable Userspace Meta Program (rump) framework for kernel development. Rump enables running kernel components in userspace without code modification. Examples include file systems, the networking stack, and the kernel entry points for a large number of system calls. This makes it possible to develop, debug and test kernel code as a normal userspace application with all the associated benefits over work done directly in the kernel.

This paper describes how, why and when to use rump for kernel development. It does not delve into the secrets of the implementation of the framework or evaluate the framework in any other sense apart from its usefulness in kernel development. The contribution of this paper is to give useful information and examples on how to make kernel development a less daunting task.

1 Introduction

Apart from device drivers and low level machine dependent code, kernel code does not generally care if it runs in the privileged CPU mode or not. The only constraint keeping kernel code from running in userspace is the fact that kernel source modules depend on other kernel modules. A small fraction of the kernel source modules depend on code which does require the privileged mode. This means that running the standard kernel binary in userspace is not possible. Furthermore, a strategy of picking single source modules from the kernel and attempting to include them in userspace programs is likely to run into missing dependencies.

The Runnable Userspace Meta Program, or *rump*, framework provides support for running code from the kernel in userspace applications. This is done by grouping existing kernel source modules into components according to their functionality and dependencies. These components are then provided as userspace libraries. Where the dependencies for desired source code involve kernel code which requires the privileged CPU mode, a reimplementation suitable for userspace is provided.

The term ”rump” is also used to describe a userlevel program which uses the rump framework to run kernel code in userspace. Additionally, we use the term ”rump kernel” to describe the kernel source code and the dependencies reimplemented for userspace.

Doing kernel development in userspace has numerous advantages. The main benefit is the light speed testing cycle: bootstrapping the rump kernel takes just microseconds, so testing changes is typically extremely fast. Additionally, a kernel panic always means an application core dump on the file system, so examining the state after a crash is straightforward. Unmodified userspace tools such as gdb, gprof and Valgrind may be used on a rump kernel. Furthermore, the setup and maintenance of a full OS installation is not required.

This paper represents current best practices in NetBSD-current as of May 2009. Some solutions presented are not optimal and are merely a way for rumps to work around system or tool problems. The solutions are also not foolproof recipes for problems in kernel development. Nevertheless, the techniques presented have proved useful in attacking and defeating many real NetBSD kernel bugs during the past 18 months.

The remainder of this paper is organized as follows. In Section 1.1 we go over the typical approaches to kernel debugging. Section 2 introduces rump from a practical perspective and Section 3 discusses the programming interfaces. Section 4 explains how to use rump for debugging the kernel and Section 5 demonstrates how to handle kernel regression testing. Finally, Section 6 concludes and lists pointers to further resources.

1.1 Problems with traditional methods

There are numerous traditional methods for doing kernel development.

1. **direct approach**[8]. The most straightforward approach is to do development directly in the kernel on real hardware. Commonly two machines are used: one for code development and one for testing. After a crash two reboots may be required: one for uploading a fix attempt and a second one for re-testing. Booting from the network or kernel modules may be used to reduce the number of reboots. Typically, a serial console and gdb are used for debugging. Alternatively, Ethernet or FireWire may be used for remote memory access debugging[1]. Another option is to examine a post-reboot kernel coredump in the debugger. Notably, in some cases of hardware device driver development the direct approach may be the only available approach.

   The drawbacks of using the same machine for development and testing are obvious: development will stall whenever rebooting. Two machines require two separate installations. Especially when doing casual development, the cost of updating the test installation before testing may be high. Furthermore, this model does not isolate the component under development. This can, to give a random although fairly unlikely example, cause networking stack development to overwrite file system memory and in turn corrupt the root file system. Finally, in case kernel debugging has to be done when the product is already deployed on the field, setting up a development machine might not be feasible.

2. **emulator/vm approach**. A full operating system is run, but instead of using real hardware, an emulator or virtual machine is used. The popular choices available on NetBSD are qemu [4] and Xen [3]. A full usermode OS also falls into this category, although NetBSD does not currently support one.

   The drawbacks listed for the direct approach apply here also. Apart from the fact that a second unit of hardware is not required and the development host is untouched by crashes during testing, this option may be worse: without KVM support, qemu is fairly slow and the setup cost of Xen is quite high if the development machine has not been installed with Xen in mind and a Xen Dom0 is not available.

3. **userland approach**. The component under development is isolated to a self-contained userspace program. It is written against a pseudo-kernel interface and e.g. locking might simply be defined away with `#ifdef` when running in userspace.

   The main problem with this approach is that kernel environment emulation is often very lazy. For example, if locking is not properly dealt with in the development phase, it will have to be sorted out when the code is moved into the kernel. Keeping the necessary pseudo-kernel emulation alive after moving it to the kernel may also prove challenging; without constant use the `#ifdef` portion of the code has a tendency to bitrot. Finally, if this technique is used in multiple projects, effort duplication may result.

[1]The author has only used a serial console for remote kernel debugging of NetBSD and cannot comment on the status of Ethernet or FireWire debugging.

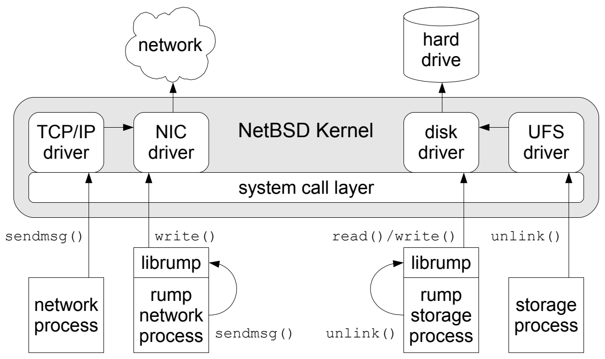


Figure 1: **rump architecture**. The kernel components are run in userspace. For comparison, regular processes are depicted on the outer edge and the rump versions in the middle.

2 Runnable Userspace Meta Programs

As mentioned in the introduction, rumps enable running kernel code components in userspace without any code modification. To function, the rump kernel must still communicate with the host OS kernel. For example, file systems must still be able to access the block device used for backing storage. However, the communication protocol with the kernel is very low level. Continuing the file system example, the protocol consists of reading and writing device blocks. The general architecture of rumps is illustrated in Figure 1.

In a way, rump is similar to the userland approach (3). However, a more complete emulation of kernel semantics, including locking and synchronization, is provided. Additionally, the implementation is centralized, so maintenance penalty has to be paid only once. Finally, rump provides a way to reuse kernel code in applications, but going into details on it is beyond the scope of this paper.

2.1 Available components

Next, we introduce the current sets of available rump components.

Base components: These components provide the base functionality for rump. They must be included in all rumps regardless of function.

- **rump**: provides base kernel functionality such as libkern, memory allocation and file descriptor management.
- **rumpuser**: used by the rump kernel code to access resources available in userspace, e.g. open a file, read from a socket or create a thread via libpthread. This is the only part of a rump kernel not compiled with `-D_KERNEL`. In a very loose sense, rumpuser is the machine dependent portion of the kernel.

Networking components: These components provide support for networking in a rump. There are essentially two choices [7]: an emulated TCP/IP stack and full networking support. The emulated stack uses the configuration of the host machine and can be applied in situations where the development interest is not directly in networking, e.g. when debugging NFS. The full networking stack requires configuration of interfaces and addresses, but is the only choice when work is directly related to networking, e.g. when tweaking the TCP subroutines.

- **rumpnet**: basic networking support. This component provides for example mbufs and sockets and must be included in all rumps using networking.
- **rumpnet_sockin**: emulates the TCP/IP networking stack using socket system calls. This component is mutually exclusive with all the components introduced next [7].
- **rumpnet_net**: interface and routing support.
- **rumpnet_inet**: IP support, includes the UDP and TCP protocols.
- **rumpnet_local**: UNIX domain protocol support.
- **rumpnet_virtif**: virtual networking interface, communicates with the world using a tap device node.
- **rumpnet_shmif**: virtual networking interface, communicates with other rumps using shared memory.

File system components: These components provide support for kernel file systems:

- **rumpvfs**: virtual file system support and file system subroutines. Additional functionality includes, for convenience, a block device driver because it is required by most file systems (although not all, e.g. nfs). This component must be included in all file system rumps.
- **rumpfs_cd9660**: ISO9660
- **rumpfs_efs**: SGI's Extent File System
- **rumpfs_ext2fs**: ext2
- **rumpfs_ffs**: Berkeley FFS

```
struct modinfo **mi;
void *handle;
int rv;

handle = dlopen("librumpfs_tmpfs.so", RTLD_GLOBAL);
mi = dlsym(handle, "__start_link_set_modules");
rv = rump_module_init(*mi, NULL);
```

Figure 2: **Example of dynamically loading a component in rump** (error handling omitted to save space).

- **rumpfs_hfs**: Apple HFS+
- **rumpfs_lfs**: Berkeley LFS
- **rumpfs_msdos**: FAT
- **rumpfs_nfs**: NFS client
- **rumpfs_nfsserver**: NFS server[2]
- **rumpfs_ntfs**: Microsoft NTFS
- **rumpfs_syspuffs**: puffs in userspace
- **rumpfs_sysvbfs**: SysV Boot File System
- **rumpfs_tmpfs**: tmpfs memory file system
- **rumpfs_udf**: Universal Disk Format

2.2 Including optional components

Components may be included in two different ways: linking them in at compile time or dynamically loading them at runtime. In both cases the dependencies much be available. For dynamic loading the dependencies must be loaded before the components that depend on them.

Compile time linking is done by supplying the linker with the necessary set of `-lcomponents`, e.g. `-lrumpfs_tmpfs -lrumpvfs`.

Dynamic loading uses functionality provided by kernel modules and is limited to components which are currently available in NetBSD as modules. Therefore, only file systems are currently loadable dynamically. Dynamic loading is a two stage process which consists of first loading and linking the shared library and then informing the rump kernel about its existence. Loading is performed by calling `dlopen()` on the library. The module information of the component is then located and passed to `rump_module_init()`. An example of this is presented in Figure 2.

The link set approach used by the kernel causes additional trouble in userspace. A link set is a mechanism which can be used by source modules to add an entry to a specific section in the compiled binary object. These sections are then coalesced by the static kernel linker into a single section which can be traversed at runtime

[2]A fully functional rump kernel NFS server depends on additional patches to mountd. These are not currently in the NetBSD CVS repository. The short version is that the exports list must be loaded to the rump kernel instead of the real kernel.

by the kernel. However, the scheme is incompatible with shared libraries[3] due to the fact that the dynamic linker does not create storage, only resolves symbols; only the first linker command line component where a link set entry is provided will be stored in a link set. The effective result is that only the first file system or networking component from the linker command line will be valid in a rump kernel. For example, a rump linked with "`-lrumpfs_ffs -lrumpfs_efs`" will have support only for FFS. There has been discussion to work around the problem in the dynamic linker, but so far the only option is to load subsequent components dynamically, as described in the previous paragraph.

2.3 Compiling a new rump kernel

Compiling a new version of the rump kernel means simply compiling the necessary libraries and installing them for consumers. For example, to compile the standard rump components, `make dependall` in src/sys/rump followed by `make install` should be used. Standard make syntax is valid and adding `DBG=-g` to the make command line will compile the rump kernel with debugging symbols. The author frequently uses also `MKSTATICLIB=no` and `MKPROFILE=no` in the makefile configuration to speed up compilation when doing testing and development.

Notably, the above differs slightly from a normal userland build. In the base system build the rump core libraries are built from reachover makefiles in src/lib and shared libraries are linked to their dependencies already at build time. When building from src/sys/rump it is not so. In practice this means that while in a regularly built system the whole set of component dependencies (e.g. `-lrump -lrumpuser -lpthread`) does not have to specified when linking a dynamic rump, for a development build from src/sys/rump they must be given. It is good practice to do this always anyway because of static libraries.

2.4 Defining additional components

Additional rump kernel components can be defined by creating makefiles which include the appropriate source modules. All of the common definitions for a rump component are available in `sys/rump/Makefile.rump`. Before inclusion the variable `RUMPTOP` designating the top of the rump source tree should be set. The path can be either relative or absolute. An example of a fictional rump component makefile is presented in Figure 3.

[3]It is by large also incompatible with dynamically loading kernel modules, since the kernel will not re-traverse link set entries every time a module is loaded.

```
RUMPTOP= /usr/src/sys/rump
.include "${RUMPTOP}/Makefile.rump"

.PATH:   ${RUMPTOP}/../myfoo/bar

LIB=     rumpfoo_bar

SRCS=    foo_dothis.c foo_dothat.c foo_dotdot.c

.include <bsd.lib.mk>
.include <bsd.klinks.mk>
```

Figure 3: **Example of a rump component Makefile.**

2.5 Version mix & match

Theoretically, and in some limited cases in the real world, rumps are portable and work across different NetBSD versions and even on different operating systems. This means that the host running a rump does not have to be the same OS or OS revision as the rump itself. For example, a rump using NetBSD-current kernel code can be run on 5.0 if issues described in the following paragraphs are kept in mind.

The real world limitation is that binary types passed from the application to the rump kernel and back must match. In case they do not match, there is no general case solution as of now, and compatibility code must be manually provided. For example, NetBSD increased the size of `time_t` from 32 bits to 64 bits after the NetBSD 5.0 release branch. Current base system file system utilities address this in compatibility code which translates the incompatible binary types and makes running possible.

For compiling an arbitrary version rump kernel on NetBSD the instructions provided in Section 2.3 are valid apart from the make command name. Instead of using `make`, build.sh [9] should be used to build a toolchain for the target rump kernel version. The `nbmake-$arch` script created by build.sh should then be used to build rump. This ensures that the toolchain can handle everything in the source tree. Compiling (and using) rump on a non-NetBSD platform is possible, but the process is currently convoluted, and the details are beyond the scope of this paper.

2.6 Tools for file systems

Next we introduce tools frequently useful in debugging file system problems.

- **rump_$fs**: The rump_$fs daemons use the puffs [6] userspace file system framework to provide mountable services of the kernel code. These daemons translate the puffs protocol to the kernel vfs protocol by using a library called p2k, or puffs-to-kernel.

  For example, the `rump_ffs` daemon does an FFS mount. The resulting mount behaves exactly like a kernel file system from an application's perspective, so standard applications such as OpenOffice can be used against them. The servers, on the other hand, behave exactly like normal userspace daemons and a debugger can be attached to them or they may be otherwise profiled.

The rump kernel file system servers are shipped in the standard installation of NetBSD starting from NetBSD 5.0.

- **fs-utils**: The fs-utils [12] suite provides rump workalikes for standard POSIX file system utilities such as ls, cp and rm. Instead of having to mount a file system through the kernel, the utilities access the file system contents fully in userspace. This has the benefit that no kernel support or mounting privileges are required. For example, `fsu_ls -lR` can be used as a quick means to verify that file system operations necessary for a recursive long directory listing still function after changes to a kernel file system have been made.

  fs-utils is currently available as source code from the NetBSD *othersrc* CVS module, or from pkgsrc.

Currently there exist no out-of-the-box rump tools for debugging networking problems and such programs must be written for the occasion using the programming interfaces described next in Section 3.

3 Rump programming

In case tools for the purpose are not available, a new program must be written. There are only two common steps across all rumps. First, all the necessary components must be linked and loaded before use is attempted. Second, the rump kernel must be initialized by calling `rump_init()` before use is attempted. The rest depends on what the rump in question does. The subject examined in this Section is interfacing with the rump kernel. Use of some of the interfaces presented in this Section is demonstrated later in Section 5.

3.1 Exported function interfaces

A number of function interface classes are exported from rump to the application. This is comparable to any library. The available sets of interfaces are discussed next.

System calls

A rump kernel exports a subset of the system calls available on a regular NetBSD system. These rump system calls have the same semantics as regular system calls. Since the rump kernel runs in the same address space as the application, there is no technical need for a system call mechanism – function calls suffice. In fact, the raison d'être for rump system calls is to provide interface sugar: having known interfaces available makes rumps easier to program. Also, converting an existing application to use rump for example when trying to reproduce a problem is easier, since it can be done by replacing the call points for syscalls in the source code. However, this method does not deal with libraries making system calls and they must be addressed separately. A general solution for libraries is not yet available, although the rump network paper [7] discusses some options.

The distinguishing factor between rump and regular system calls is in their naming. All rump system calls are prefixed with "rump_sys". For example, the call `rump_sys_open("/my/fisu",O_CREAT,0777)` will, barring an error, create the file "fisu" in the directory "/my". The resulting file descriptor is valid only in other rump system calls and passing it to a regular system call will cause undefined effects and vice versa. The system calls are declared in the header `<rump/rump_syscalls.h>`. The header also acts as the list of currently supported rump system calls. Since the call semantics are equal to regular system calls, documentation is available from regular manual pages.

rump interfaces

Over the course of using and developing rump, various interfaces into the kernel bypassing the syscall layer have proved to be of use. This set of rump interfaces is under constant development with new ones being added as a need arises. The current interfaces are available in the `rump.h` header. Examples of these interfaces are found in Figure 4.

process/lwp:

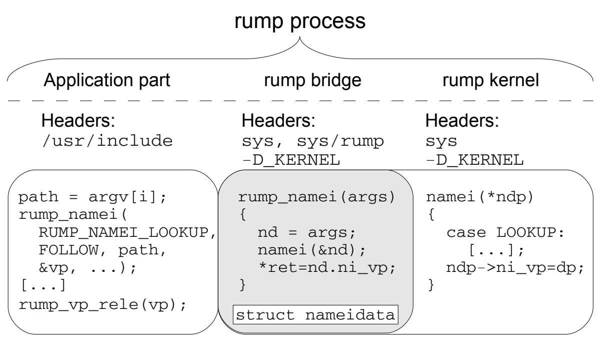
```
struct lwp *rump_setup_curlwp(pid_t, lwpid_t, int);
struct lwp *rump_get_curlwp(void);
void        rump_set_curlwp(struct lwp *);
void        rump_clear_curlwp(void);
```

credentials:

```
kauth_cred_t  rump_cred_create(uid_t, gid_t,
                                   size_t, gid_t *);
kauth_cred_t  rump_cred_suserget(void);
void          rump_cred_put(kauth_cred_t);

#define rump_cred_suserput(c) rump_cred_destroy(c)
/* COMPAT_NETHACK */
#define WizardMode()  rump_cred_suserget()
#define YASD(cred)    rump_cred_suserput(cred)
```

Figure 4: **Examples of misc. rump interfaces**



```
path = argv[i];
rump_namei(
  RUMP_NAMEI_LOOKUP,
  FOLLOW, path,
  &vp, ...);
[...]
rump_vp_rele(vp);
```

```
rump_namei(args)
{
  nd = args;
  namei(&nd);
  *ret=nd.ni_vp;
}
```

```
namei(*ndp)
{
  case LOOKUP:
    [...];
  ndp->ni_vp=dp;
}
```

Figure 5: **rump namespaces**. A rump is divided into three conceptual zones. The rump bridge provides storage for `struct nameidata`, since the definition is not available in the application namespace.

VFS interfaces

The original purpose of rump was to run kernel file system code as a userspace server. This required access to the virtual file system interfaces. The complete vnode interface is available from userspace plus additional miscellaneous vfs interfaces interfaces.

The vnode routines are available after including the header `rump/rumpvnode_if.h`. They behave exactly like the kernel interfaces: when called with a given vnode, they call the file system underlying the vnode. The only difference is that while kernel operations have the name `VOP_OPER()`, the rump exported names are `RUMP_VOP_OPER()`.

To get a reference to a vnode, the vfs interfaces must generally be used. They, however, are not introduced by `rumpvnode_if.h`, but rather by `rump.h`. This is due to the fact that the vnode interface prototypes are autogenerated from the kernel's `vnode_if.src`, while the vfs interfaces are handcrafted. Specifically, namei is available as `rump_namei()`.

3.2 The kernel namespace

Most system headers contain `#ifdef _KERNEL` sections and do not provide function prototypes, all structure definitions or all macros if the header is included from a userland application, i.e. one which does not define `_KERNEL` for its cppflags. While it is possible for a userspace application to define `_KERNEL` for its build and access the values meant for kernel consumers, this generally results in a good amount of problems due to name collisions and implicitly included headers. The `_KERNEL` namespace division is illustrated in Figure 5.

Even though a rump kernel operates in the kernel namespace and is compiled with `_KERNEL`, the rest of the rump program does not. However, accessing the kernel even where the standard rump framework does not support it may be useful. We discuss using kernel structures, macros and functions in the following subsections.

Structures

Some kernel structures are exported to userland. Typically, they are either structures shared between the kernel and userland (e.g. `struct timespec`) or structures exposed for the benefit of kmem grovelers[4] (e.g. `struct vattr`). While both subclasses of exported structures can be examined and modified directly through pointers from the application part of a rump, it is critical to make sure that both structure definitions are the same. An excellent example once again is the recent `time_t` size change to 64bits, which caused a lot of structures to become binary incompatible with their old versions. As mentioned already in Section 2.5, there is no general solution for mixing incompatible versions at this point.

If a structure is not exported to userspace, it cannot be directly accessed from the application part of a rump. If it must be accessed directly, there are two strategies: either provide a local copy of the structure for the application part or provide an interface into the rump kernel which does the necessary handling. The decision on which approach to choose is highly dependent on the situation, and no general recipe can be given.

Preprocessor macros

Preprocessor macros are commonly used throughout the kernel for bitmasks and magic values passed to interfaces. As with structures, some are exported to userland, but others are hidden behind `_KERNEL` or in headers not installed to userland at all. Strategies for use are similar to structures.

Rump provides definitions for a set of useful values in the header `rumpdefs.h`. The values are generally prefixed to prevent application namespace pollution. For example, the namei symbol `LOOKUP` is available as `RUMP_NAMEI_LOOKUP`, as used in Figure 5. It should be noted that while these helper values are provided by the system, the same disclaimer as for hand-copied structures apply: the values from the headers installed to userland may be out-of-sync with the rump kernel.

Kernel functions

Making a function call from the application part to the rump kernel is possible for any kernel interface. How-

[4] A *kmem groveler* is an application which accesses kernel memory directly, typically through the `kvm(3)` interfaces.

```
          source              object             library              final library
prop_array_add -> prop_array_add -> prop_array_add -> rumpns_prop_array_add
             compile             link          rename (objcopy)
```

Figure 6: **Illustration of symbol renaming**. The name of the symbol, a function in this case, is displayed for each step. The diagram applies to both the function definition and in-kernel callers, but not the application part.

ever, this requires access to the function prototype. Generally, kernel interfaces are not exported to userland, and therefore not available in the application part except via the interfaces examined in Section 3.1.

With rumps, there is also another issue when calling kernel functions directly. In a normal operating system scenario, the kernel and user namespaces are disjoint. This means that a user program and the kernel can contain symbols with the same name. For example, a standard program using *proplib* requires the function `prop_array_add()` both in the kernel and in the libprop userspace library. Since the rump kernel is linked directly against the application, this would result in a conflict where the same symbol is defined twice. To address the issue, the symbols in the rump kernel are renamed en masse *after* compilation. All symbols that do not begin with the string ”rump” or ”RUMP” are prefixed with ”rumpns”. This means that `prop_array_add()` will be renamed to `rumpns_prop_array_add()`. Since both the kernel function definition and the kernel callers are renamed, this does not affect the runtime behavior inside the rump kernel. Since the name is now different, there are no symbol collisions with userland applications, and linking can safely be done. An illustration of the different stages of compilation and symbol renaming is provided in Figure 6.

The renaming means that even if we could access a function prototype by defining `_KERNEL` for our C preprocessor, the function prototype would now be wrong, since it is lacking the ”rumpns” prefix – userland components do not undergo any symbol renaming.

A method for accessing arbitrary kernel routines directly from the application part is to declare the properly namespaced prototypes locally in the application. The downside is that this requires manual work and information duplication. An example illustrating this is found in Figure 7.

As manually exporting numerous routines from the kernel can get taxing, an alternate strategy may be applied. This consists of: identify kernel space functionality, implement it in the rump kernel, export only one function and call that. This is illustrated, for comparison with the previous method, in Figure 8. Unless the bridge routine fits into an existing rump component, the cost of this approach is quite high due to having to introduce a

```
void rumpns_vfs_mount_print(struct mount *,
    int, void (*pr)(const char *, ....));
void rumpns_printf(const char *, ...);

func()
{
    struct mount *mp;

    rump_sys_mount(MOUNT_THEFS, "/mymnt", ...);
    /* do other stuff */
    rump_vfs_getmp("/mymnt", &mp);
    rumpns_vfs_mount_print(mp, 1, rumpns_printf);
}
```

Figure 7: **Calling a kernel function directly from userspace code**. This simplified listing shows how to call a kernel diagnostic routine to print information about a mount point in the rump kernel. It is worth noting that the printf function pointer passed as the third argument is from the rumpns namespace, i.e. the kernel printf function. If plain `printf` were used, the libc printf function would be used as a callback. It might or might not have the correct semantics.

```
/* file1: rump bridge */
#include <headers>
rump_myprintfunc(const char *path)
{
    struct nameidata nd;
    int rv;

    NDINIT(&nd, LOOKUP, FOLLOW, UIO_USERSPACE, path);
    if ((rv = namei(&nd)) != 0)
        return rv;
    vfs_mount_print(nd.ni_vp->v_mount, 1, printf);
    vrele(nd.ni_vp);
    return 0;
}

/* file2: application part */
#include <otherheaders>
func()
{
    rump_sys_mount(MOUNT_THEFS, "/mymnt", ...);
    /* do other stuff */
    rump_myprintfunc(path);
}
```

Figure 8: **Calling kernel namespace code through a self-defined kernel routine**. This is another alternative for implementing the code displayed in Figure 7.

```
golem> env RUMP_BLKFAIL=500 fsu_ls ffs2.img -l ps
rumpblk: FAULT INJECTION ACTIVE! fail 500/10000. seed 2668677932
-r-xr-xr-x  1 pooka  wheel  41675 Apr 29 20:02 ps
golem> env RUMP_BLKFAIL=500 fsu_ls ffs2.img -l ps
rumpblk: FAULT INJECTION ACTIVE! fail 500/10000. seed 4121636030
block fault injection: failing I/O on block 1408
fsu_ls: ps: Input/output error
```

Figure 9: **rump block device fault injection**. Fault injection with the above parameters will cause 5% of block I/O requests to fail. We see two consecutive invocations of the fsu_ls tool. The first succeeds and the latter fails because a failure was randomly injected.

separate library in the process. Since the illustrated case is very simple, taking this approach is not worth it. However, especially in case the code needs to inspect structures declared only in the kernel namespace, the payoff may be significant. Specifically, if rump would not provide something like `rump_vfs_getmp()`, support in kernel namespace would likely be a good idea.

4 Debugging

In this section we examine some strategies on how to use rumps to locate and solve kernel problems.

4.1 Fault injection

Fault injection allows to test code error paths by returning fictional faults from routines. In a case where the full operating system is used, care must be taken not to inject faults into the wrong components, for example the root file system. Since a rump is self-contained, very little care is needed to apply fault injection.

The rump block device driver offers two tunables for fault injection: the percentage of failed I/O and the seed for the pseudo random number generator used for deciding which I/O requests fail. Setting both gives repeatable faults for the same set of operations performed on the same file system. The environment variable `RUMP_BLKFAIL` controls how many of 10,000 requests fail on average. The variable `RUMP_BLKFAIL_SEED` sets the PRNG seed and can be any integer value. If not set, a random value will be used.

Figure 9 illustrates how to use fault injection. If the latter failed case were interesting, it could be repeated by setting the `RUMP_BLKFAIL_SEED` environment variable to 4121636030.

While not currently implemented, if would not be difficult to make the rump network interfaces support a similar scheme to introduce packet loss and jitter into the simulated network.

4.2 Multithreading strategies

Not all development tools work perfectly with threads. The best example on NetBSD is gdb, which cannot currently reliably single-step multithreaded programs. Breakpoints and memory examination work even in multithreaded programs and all the information is available from core dumps. Even so, sometimes single stepping is the best approach to locating a problem.

- While rump makes use of threads and allows components to create them, it is possible to attempt to run without threads. This is done by setting the environment variable `RUMP_THREADS` to 0 before running a rump. It will cause the rump kernel thread creation to simply ignore a kthread creation request for a thread it knows it can safely ignore. For example, in short debug runs it is not critical that the namecache purging thread is run. If creation of any unknown thread is attempted in non-threaded mode, the rump kernel will panic. Most file systems can safely be run in non-threaded mode, but for example nfs cannot, since it depends on the networking stack, which is heavily dependent on threads.
- Other utilities such as gprof seem to exhibit even worse behavior and are sometimes completely unwilling to work if the application is even linked against the pthread library. If necessary, the rumpuser component may be compiled without any threading support. This is done by editing `rump/librump/rumpuser/Makefile`, commenting out `rumpuser_pth.c` and uncommenting `rumpuser_pth_dummy.c`. With the stubs in place, threaded programs cannot be run at all, but some profiling and development tools may give the results desired.
- In some cases it is possible to fake threads for debugging purposes. For example, the sockin network component has unthreaded mode debugging code which assumes that for every sent packet and only a

sent packet there will be an incoming packet. This causes all network traffic to be tied to the sender and therefore a receiver thread is not required to handle asynchronously incoming data. Clearly, this is not a valid generalization, but sometimes it allows to debug special cases.

- If the problem zone is known, most faults can be identified by examining the data present and looking at how the code uses the data. The data can easily be secured by placing a call to `panic()` at a suitable location. The core dump can then be examined for the data.

In case all else fails, the following observation from Brian Kernighan should be kept in mind: "*The most effective debugging tool is still careful thought, coupled with judiciously placed print statements*". With rumps even slightly less judicious print statements may be a valid option due to the short test cycle, followed by a judicious application of grep.

4.3 Valgrind

An extremely useful dynamic analysis tool for userspace development is Valgrind [10]. It is perhaps best known for its ability to detect use of uninitialized memory, wide and dangling pointers, and memory leaks.

There is an old and incomplete but still semifunctional port of Valgrind to NetBSD, called vg4nsd [2]. While not all system calls are supported, rumps can occasionally be run under Valgrind with good results due to the fact that a rump does not use many host kernel system calls. The exception is that the current implementation of vg4nbsd does not support threads and pthread use must be completely disabled by using the dummy stubs, as described in Section 4.2.

4.4 Example: single stepping in FFS

Recently one NetBSD developer, who does not usually work on file systems, encountered a problem on a deployed system where creating a directory on an FFS mount with over 4TB of free space always returned `ENOSPC`. Code reading did not provide any useful guesses on where the problem was.

Normally debugging the issue would have required either setting up a kernel development environment or slowly closing in on the problem through a series of added print statements and reboots. The use of rump_ffs allowed to single step the problematic code without any reboots and quickly locate the problem. This is demonstrated in the reconstruction below.

First, we set the necessary environment variables. The variable `P2K_NODETACH` prevents the rump_ffs server from detaching from the terminal via fork and therefore not requiring gdb fork following modes to be set. A breakpoint is then set to the mkdir vnode operation implementation, `ufs_mkdir`.

Notice that we do not use `rumpns_ufs_mkdir`, but rather are able to use the original name. This is because symbol renaming does not touch the debugging information generated by a `-g` compile. If the compiled binary does not contain debugging symbols, the breakpoint should be set to the rumpns symbol instead.

```
golem> setenv RUMP_THREADS 0
golem> setenv P2K_NODETACH
golem> gdb rump_ffs
GNU gdb 6.5
...
(gdb) break ufs_mkdir
Function "ufs_mkdir" not defined.
Make breakpoint pending on future
    shared library load? (y or [n]) y
Breakpoint 1 (ufs_mkdir) pending.
```

The program is started. This mounts the file system on wd0e to /mnt. The command line parameters for the server are described on the rump_ffs manual page, and are actually exactly the same as for mount_ffs. The breakpoint is resolved, which means that the necessary dynamic library was loaded at this stage.

```
(gdb) run /dev/wd0e /mnt
...
Pending breakpoint "ufs_mkdir" resolved
rump warning: threads not enabled,
    not starting vrele thread
rump warning: threads not enabled,
    not starting namecache g/c thread
```

Meanwhile, a directory is created under /mnt using `mkdir`. This causes the breakpoint to trigger. Now the actual debugging in the form of single stepping begins. The mkdir command blocks while debugging takes place.

```
Breakpoint 2, ufs_mkdir (v=0xbfbfd918)
    at /usr/allsrc/src/sys/rump/fs/lib/
    libffs/../../../../ufs/ufs/ufs_vnops.c:1326
1326            struct vop_mkdir_args /* {
(gdb)
```

The problem is located in the block allocator. The allocator is supposed to return a positive result if a block is successfully allocated. However, due to a signed integer overflow problem, the block number returned was negative. This caused the calling code to think that allocation failed when it in fact did not. Further examination revealed that this class of overflow problems was fixed already months earlier, but the original fix omitted two code paths.

```
1381            blkno = cg * fs->fs_fpg + bno;
(gdb) print cg * fs->fs_fpg + bno
$5 = -2049440479
(gdb) print (daddr_t)cg * fs->fs_fpg
$6 = 2245526808
```

5 Test cases

Kernel test cases written against rump have the advantage of not crashing the test host kernel in case the test results in a kernel panic. This means it is possible to easily extract a useful test report out of test which resulted in a crash. A test run which ends in a kernel panic is illustrated in Figure 10 and described in more detail later in this Section.

Occasionally, it is desirable to be able to call kernel interfaces with parameter values that are hard to impose using system calls, especially in unit testing. One way to address this in a standard setup is to create a kernel module which provides a special system call that performs the unit test when executed. However, test code tends to be more error-prone than standard kernel code and will likely result in unwanted crashes more frequently. With rump it is possible call arbitrary kernel interfaces by using techniques described in Section 3. This makes it possible to generate unit tests with no parameter restrictions and yet protect against test host kernel crashes.

The Automated Testing Framework (*atf*) [11] provides an infrastructure for creating tests and producing unified reports. Tests can be written either as C programs or shell scripts. Both options can be used with rumps. C programs can be used to interface with the rump kernel as was demonstrated earlier in this paper. Shell scripts can be used with rump tools, such as fs-utils.

5.1 Example: file descriptor passing

File descriptor passing enables passing open file descriptors from one process to another. This is done by constructing a specific IPC message which is sent over a socket. Recently, NetBSD-current had an off-by-one error which caused the file descriptor array to be indexed by an invalid file descriptor. If the passed file descriptor was a large number, the array index would point to hyperspace and result in a kernel panic.

The rump regression test set for kernel file descriptor passing consists of two separate test cases. The first case tests that file descriptor passing works as expected. The second case attempts to pass an invalid file descriptor and checks that it gets a proper error message as the result.

The test cases are found in the NetBSD source tree in `src/tests/syscall/t_cmsg.c`. We will go over them step-by-step to demonstrate how to unify atf and rump for writing test cases. Some of the presented code has been modified for presentation purposes by leaving out for example error handling. Please refer to the source tree for the full version. We begin our demonstration with the case verifying that out-of-bounds access is not allowed, since out of the two it is simpler.

```
t_cmsg (1/1): 2 test cases
    cmsg_sendfd: Passed.
    cmsg_sendfd_bounds: Failed: Test case did not
        exit cleanly: Segmentation fault (core dumped)

    Failed test cases:
        t_cmsg:cmsg_sendfd_bounds
```

Figure 10: **Example of a test resulting in a kernel panic**. If run against a live kernel, the failure would have caused the system to crash and an incomplete test report.

Out-of-bounds case

The test case body and local variables are declared.

```
ATF_TC_BODY(cmsg_sendfd_bounds, tc)
{
    struct cmsghdr *cmp;
    struct msghdr msg;
    struct iovec iov;
    int s[2];
    int fd;
```

rump is initialized and a socketpair to be used for descriptor passing is opened.

```
    rump_init();
    if (rump_sys_socketpair(AF_LOCAL,
      SOCK_STREAM, 0, s) == -1)
        atf_tc_fail("rump_sys_socketpair");
```

The control structure to be used for file descriptor passing is initialized.

```
    cmp = malloc(CMSG_LEN(sizeof(int)));

    iov.iov_base = &fd;
    iov.iov_len = sizeof(int);

    cmp->cmsg_level = SOL_SOCKET;
    cmp->cmsg_type = SCM_RIGHTS;
    cmp->cmsg_len = CMSG_LEN(sizeof(int));

    msg.msg_iov = &iov;
    msg.msg_iovlen = 1;
    msg.msg_name = NULL;
    msg.msg_namelen = 0;
    msg.msg_control = cmp;
    msg.msg_controllen = CMSG_LEN(sizeof(int));
```

An invalid value is inserted as the descriptor to be passed.

```
    *(int *)CMSG_DATA(cmp) = 0x12345678;
```

Descriptor passing is attempted. The rump kernel should return an error indicating that an invalid file descriptor was being passed.

```
    rump_sys_sendmsg(s[0], &msg, 0);
    if (errno != EBADF)
        atf_tc_fail("descriptor passing failed: "
          "expected EBADF (9), got %d\n(%s)",
          errno, strerror(errno));
}
```

In Figure 10, the fix to this problem had been reverted for demonstration purposes. The call to sendmsg did not return at all because the process got a segmentation violation signal for out-of-bounds indexing of the array.

Valid case

The test for file descriptor passing from one process to another is more complex than the previous, since we have to simulate two different processes in the rump kernel – passing a file descriptor within a single process is not interesting, as it is the same as `dup()`. Since rump does not support `fork()`, we need to manage multiple processes in another fashion using a rump interface. The lack of fork also rules out the use of socketpair for creating the communication channel. Instead, we use a bound Unix domain socket, which is available to both processes in the file system namespace.

First, we mount tmpfs as the root file system over the current root. This is done because the rump root file is very barebones and does not support most operations. In this test we want to create a Unix domain socket on the file system.

```
rump_init();
rump_sys_mount(MOUNT_TMPFS, "/", 0,
  &args, sizeof(args));
```

Next, we create the file system socket at the location `SOCKPATH` and start listening for incoming connection requests.

```
memset(&sun, 0, sizeof(sun));
sun.sun_family = AF_LOCAL;
strncpy(sun.sun_path, SOCKPATH, sizeof(SOCKPATH));
s1 = rump_sys_socket(AF_LOCAL, SOCK_STREAM, 0);
rump_sys_bind(s1, (struct sockaddr *)&sun,
  SUN_LEN(&sun));
rump_sys_listen(s1, 1);
```

Then we store our current *lwp* and create a new process and a new lwp. We connect our new process to the socket opened by the initial process.

```
l1 = rump_get_curlwp();
l2 = rump_newproc_switch();
[...]
rump_sys_connect(s2, (struct sockaddr *)&sun,
  SUN_LEN(&sun));
```

For testing purposes, we create a file onto our root tmpfs file system and write a magic string into the file. We then rewind the file descriptor to the beginning of the file and pass the file descriptor in the usual way.

```
#define MAGICSTRING "duam xnaht"
fd = rump_sys_open("/foobie",
  O_RDWR|O_CREAT, 0777);
rump_sys_write(fd, MAGICSTRING,
  sizeof(MAGICSTRING));
rump_sys_lseek(fd, 0, SEEK_SET);
[...]
*(int *)CMSG_DATA(cmp) = fd;
rump_sys_sendmsg(s2, &msg, 0);
```

Finally, we switch back the original process, accept the incoming connection, get the passed file descriptor and verify we can read the magic string from it. If the contents read match the ones written, we can conclude that we were able to pass a file descriptor from one process to another and use it with expected results in the process it was passed to.

```
rump_set_curlwp(l1);
sgot = rump_sys_accept(s1,
  (struct sockaddr *)&sun, &sl);
rump_sys_recvmsg(sgot, &msg, 0);
rfd = *(int *)CMSG_DATA(cmp);

memset(buf, 0, sizeof(buf));
rump_sys_read(rfd, buf, sizeof(buf));
if (strcmp(buf, MAGICSTRING) != 0)
    atf_tc_fail("expected \"%s\", "
      "got \"%s\"", MAGICSTRING, buf);
```

Additionally, to cope with the possibility of the test case hanging due to blocking socket I/O, we set an atf timeout. If the test case does not finish in two seconds, it is deemed to have failed.

```
atf_tc_set_md_var(tc, "timeout", "2");
```

Ultimately, a Makefile is necessary for building the test case. Apart from additional rump component libraries we must specify, the Makefile is like any other.

```
TESTS_C=        t_cmsg

LDADD.t_cmsg+=  -lrumpnet_local -lrumpnet_net
LDADD.t_cmsg+=  -lrumpnet -lrumpfs_tmpfs -lrumpvfs
LDADD.t_cmsg+=  -lrump -lrumpuser -lpthread
```

It is worth noting that even without tmpfs this test case would depend on the vfs component. This is because the Unix domain local socket component uses the file system code for creating bound sockets in the file system namespace.

6 Conclusions

This paper described the uses of the Runnable Userspace Meta Program (rump) framework for kernel development and debugging and provided comparisons to currently popular approaches. We started by explaining that rump consists of kernel subsystems grouped into components and is provided as shared libraries in userspace. We then went into details on how to use rump for kernel development, including information about the programming interfaces, how to compile, how to debug and how to write regression tests.

Rump provides a fast way for doing incremental kernel development. It supplies a safe environment for learning about how the kernel works and enables non-expert users to provide more data in bug reports due to the fact that extracting it no longer requires special knowledge on how to do kernel debugging – regular userspace debugging skills suffice.

Future work includes complete and accurate support for more kernel subsystems. Additionally, more automated regression tests and tools for no-setup execution are being planned.

6.1 Further resources

- The rump web page [5] provides information on the current status of development.
- The rump.3, p2k.3 and rump_$fs.8 NetBSD manual pages [1] provide information about programming interfaces and use.
- The ukfs library, as documented in the ukfs.3 manual page, provides an alternative programming interface for rump file systems (not discussed in this paper).
- The Automated Testing Framework [11] provides an infrastructure for testing. As was demonstrated in this paper, it integrates well with rump for convenient kernel regression testing.

Acknowledgments

This part of the ongoing rump project was funded by The Nokia Foundation. Stephen Borrill helped with proofreading.

References

[1] NetBSD Manual Pages. http://man.NetBSD.org/.

[2] Valgrind for NetBSD. http://vg4nbsd.berlios.de/.

[3] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles* (2003), pp. 164–177.

[4] BELLARD, F. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track* (2005), pp. 41–46.

[5] KANTEE, A. Runnable Userspace Meta Programs. http://www.NetBSD.org/docs/rump/.

[6] KANTEE, A. puffs - Pass-to-Userspace Framework File System. In *Proc. of AsiaBSDCon* (2007), pp. 29–42.

[7] KANTEE, A. Environmental Independence: BSD Kernel TCP/IP in Userspace. In *Proc. of AsiaBSDCon* (2009), pp. 71–80.

[8] LEHEY, G. Debugging kernel problems, 2006.

[9] MEWBURN, L., AND GREEN, M. build.sh: Cross-building NetBSD. In *Proc. of BSDCon* (2003), pp. 47–56.

[10] NETHERCOTE, N., AND SEWARD, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proc. of PLDI* (2007), pp. 89–100.

[11] VIDAL, J. M. M. Automated testing framework. http://www.NetBSD.org/ jmmv/atf/.

[12] YSMAL, A. FS Utils. http://NetBSD.org/˜stacktic/fs-utils.html.