Rump Device Drivers: Shine On You Kernel Diamond

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

Abstract

BSD-based operating systems implement device drivers in kernel mode for historical, performance, and simplicity reasons. In this paper we extend the Runnable Userspace Meta Program (rump) paradigm of running unmodified kernel code directly in a userspace process to kernel device drivers. Support is available for pseudo device drivers (e.g. RAID, disk encryption, and the Berkeley Packet Filter) and USB hardware drivers (e.g. mass memory, printers, and keyboards). Use cases include driver development, regression testing, safe execution of untrusted drivers, execution on foreign operating systems, and more. The design and NetBSD implementation along with the current status and future directions are discussed.

1 Introduction

The Runnable Userspace Meta Program (rump) approach divides the kernel into fine-grained functional units called components and makes it possible to run them without code modifications in a regular userspace process. This is enabled by the observation that almost all kernel code will work in unprivileged mode as such. For machine dependent parts such as memory management, a tiny amount of shim code is necessary. The components, typically device drivers in the context of this paper, may then be combined in various configurations to create a virtual kernel running in userspace. Rump has already been shown to be a working approach for the networking stack [9] and file systems [10]. This paper explores expanding the approach to device drivers.

Roughly speaking, device drivers can be divided into two categories: ones which are backed by hardware and ones which are not. The latter category is commonly known as pseudo devices. Our implementation supports drivers from both categories. USB drivers are the currently supported hardware drivers. To give an example for the pseudo device category, rump supports the NetBSD softraid implementation, RAIDframe [4]. In fact, RAIDframe was originally created for prototyping RAID systems and ran also in userspace. However, the NetBSD kernel port of RAIDframe was not able to run in userspace until recently as part of the work presented in this paper [14].

In supporting kernel drivers in userspace, there are two facets to consider. The first challenge is making the code run and work as a standalone program. The second one is seamlessly integrating the result with the host OS using a driver proxy [6]. Let us consider RAIDframe again. Having the RAID driver running inside a process does not make it possible to mount a file system residing on the RAID from the host running the process. A driver proxy can be used to make this possible. If we consider an analogy with file systems, puffs [8] is a proxy and e.g. sshfs is the driver – although in the context of this paper we want to integrate drivers which, unlike sshfs, were not originally written to be run in userspace and to be used by a proxy. Not having to rewrite existing and tested kernel device driver code for userspace and proxy interfaces is a key quality.

The most obvious application and original motivation for rump is easy kernel code development. Code can be written and tested as a userspace program and simply dropped into the kernel when it has reached required maturity. The converse also applies: if a certain driver has regressed beyond acceptable stability, it is still possible to run it as a userspace server. This makes continued service is possible without the risk of bringing down the entire host system in case of driver failure. Furthermore, in an open source context, being able ask users to debug driver failures as userspace programs greatly helps with bug reports – the set of users being able to debug an application given short instructions is far greater than the set of users willing to set up a kernel debugging environment and perform live kernel debugging.

Security implications may also apply. USB devices are generally plugged into mobile systems without much

consideration. Running the driver for an untrusted device in kernel mode risks compromising the entire system in case of maliciously crafted or misbehaving hardware. Minimizing kernel involvement by running most of the driver in an unprivileged process limits direct damage to the driver process in case the driver fails to handle an unexpected device message. This is a less likely attack than the similar one for untrusted file systems [10], but still worth considering.

Although rump code may be an unmodified kernel driver, it by no means has to be. This makes it possible to have multiple, specialized versions of drivers which sacrifice the generality of the driver for a performance benefit in a specific application. It also makes it possible to use drivers which are from a different different versions of NetBSD, or even the use of NetBSD drivers on non-NetBSD operating systems [2].

The rest of this paper is organized as follows. The remainder of the this section includes a brief introduction and review of the main features of Runnable Userspace Meta Programs. Section 2 details how applications access device drivers and how to implement a proxy. Section 3 discusses pseudo device support and applications, while Section 4 does the same for hardware drivers. Finally, Section 5 concludes and summarizes future work.

1.1 Brief Introduction to rump

We use the term *rump* in a dual sense. For one, it is the enabling technology for running unmodified kernel components in userspace. Also, we use it to describe a program which runs kernel code in this manner. The term *host* is used to describe the operating system the rump is running on.

The rump method is midway through two extremes of making kernel code run in an unprivileged domain [7]:

1. full OS emulation

2. running small fractions of heavily `#ifdef`'d code in a process, as is a typical way to do the initial stages of kernel code development

Full OS emulation can be accomplished with techniques ranging from full machine virtualization to paravirtualization to a usermode operating system. The key difference between rump and a usermode operating system is that a rump does not contain processes or virtual memory, because they are unnecessary overhead when interest is directly related to the kernel code and its functionality. They key difference between rump and the ifdef approach is that rump does not require adding ifdef-blocks to source modules, but can utilize unmodified kernel code as-is.

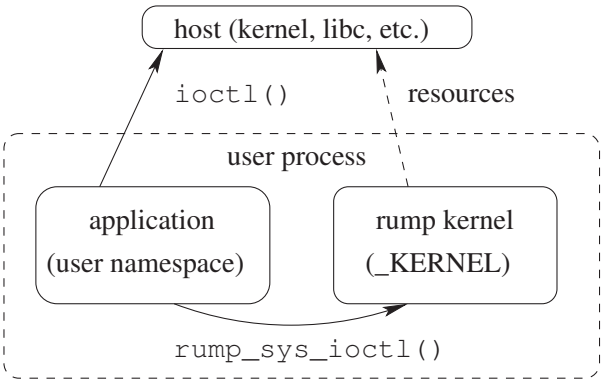

Figure 1: **rump architecture**: Architecture of a typical rump process. The application part contains the program logic (e.g. testing, userspace server, etc.). To satisfy the program logic, the application makes calls to the drivers in the rump kernel. If necessary, the application can also call host interfaces, such as the `ioctl()` system call or the `printf()` libc call.

A rump process consists of two parts: a userspace part and a kernel part. Both parts are run inside the same process, so the division is enforced in C namespace only as opposed to the common operating system situation where the user application and kernel separation is enforced by the MMU. The userspace part makes call into the kernel using exposed interfaces. By default, the standard system call interface is provided, with the difference that the calls have the `rump_sys` prefix, e.g. the `ioctl()` system call is made into the rump kernel with `rump_sys_ioctl()`. This general structure of a rump is illustrated in Figure 1.

Different rump kernel configurations are created by combining various rump components, either when linking a program or dynamically loading components at runtime. For example, a rump with RAIDframe support would consist of at least these library components:

- **rumpdev_raidframe**: RAIDframe itself
- **rumpdev_disk**: generic in-kernel disk support
- **rumpdev**: device support in rump
- **rumpvfs**: VFS support. While RAIDframe itself works on a device level and does not require file system support, any user program wishing to configure or access the raid will do so via the device files `/dev/raid*`.
- **rump**: rump kernel base
- **rumpuser**: rump host call interface (for accessing the files backing the virtual raid).

Another example is a rump supporting the *rum(4)* wireless USB interface in the inet domain. It uses the following components: *rumpdev_usbrum*, *rumpdev_ugenhc*, *rumpdev_usb*, *rumpdev*, *rumpnet_net80211*, *rumpnet_netinet*, *rumpnet_net*, *rumpnet*, *rumpvfs*, *rumpcrypto*, *rump* and *rumpuser*. The interesting ones are:

- **rumpdev_usbrum**: the rum driver itself.
- **rumpdev_ugenhc**: the rump USB host controller. We will discuss this in more detail in Section 4.
- **rumpnet_80211**: support routines for IEEE 802.11 networking.
- **rumpcrypto**: cryptographic routines required by net80211
- **rumpvfs**: VFS support. For this rump VFS is required because the rum driver loads its firmware off the disk when the driver is initialized. Note that no real file system component is included, since the firmware is actually being loaded off the host's file system by a pass-through driver included in the rumpvfs component.

As a process running in non-privileged mode, a rump kernel is indistinguishable from any other application process and as such cannot be used to crash or exploit the host it is running on. Also, a rump kernel startup time is in the typical case in the millisecond range, making it ideal for fast iteration. The only exception is slow hardware probes where the kernel drivers perform long delays to make sure hardware has initialized, but even in those cases a highly granular rump process will not take a second to start.

2 Accessing Device Drivers

Based on how a device is accessed by an application, we divide device drivers into three categories. We then discuss the categories briefly and explain their relevance with rump.

1. implicitly accessed devices
2. implicitly accessed devices configured through a file system node
3. explicitly accessed devices (via file system node)

Network interfaces are used by applications via the sockets interface. The interface is not explicitly named when sending network traffic, but rather selected by the system based on available interfaces and the peer's address. For incoming traffic, the interface is "selected" by the network. Furthermore, interface address configuration is done through a socket, although in this case the interface being configured is explicitly named.

Disk backed file systems, such as FFS, are "configured" by mounting them. The disk partition on which the file system resides is identified in this step. This establishes a relationship between between a directory tree hierarchy and the backing device. Applications are not aware of the backing device, and simply access contents through the file system hierarchy.

However, with e.g. a printer or the Berkeley Packet Filter [12], the driver is explicitly named the by application. In this case, the application addresses the driver through a device special node in the file system, usually located in `/dev`. In the cases mentioned above, the device nodes would be `/dev/ulpt<n>` and `/dev/bpf`, respectively. The file system node contains a (*major*, *minor*) identifier tuple. The major number tells the kernel which driver to direct access to, and the minor number can be used by the driver for internal partitioning. For example, the *ulpt* printer driver uses the minor number to distinguish between multiple printers attached to the system.

If the application part of the rump is to be able to configure a driver through a device node, the device node must exist in the file system namespace of the rump. It is important to note that this namespace is (and should be) different from the host's file system namespace, since the major number space between a rump kernel and the host kernel is not necessarily the same. Finally, it is equally important to note that if there are multiple rumps running on a host, they all have their unique file system namespaces.

Since NetBSD does not yet support devfs, we have augmented the appropriate device components to create /dev nodes on the rump in-memory root file system namespace during attach. This allows the application part to behave like an application and assume that the necessary system nodes are already present. Now, for example when examining bpf, the application can open a file descriptor for the rump bpf driver by calling `rump_sys_open()` on `/dev/bpf`, configure it with `rump_sys_ioctl()` and read captured packets with `rump_sys_read()`.

2.1 Integration with the host (proxy)

When a device driver is running inside a rump kernel, it is directly accessible only from the rump kernel. In some cases access to this driver from the host system may be desired, such as when wanting to mount a file system on the host with the mass media driver being run in a rump kernel. The simplest reason for wanting to do this is the host not supporting the device in question.

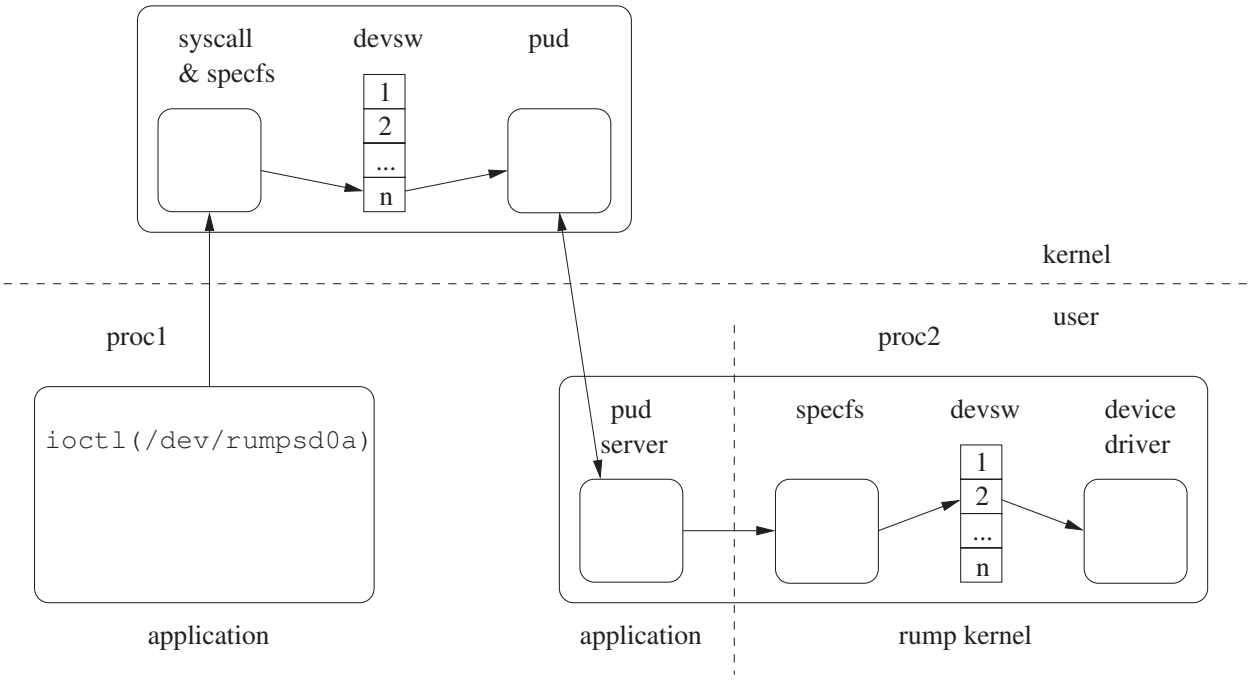

Figure 2: **rump device integration with the host**: access to a device registered to pud is queued to be handled by the pud userspace server which corresponds to the device major being accessed. The server calls the respective vnode operation of the rump kernel to deliver the request to the device driver in rump. Access to the driver in rump works also if the request is initiated by the host kernel instead of a process.

The well-known NetBSD method for mounting a file system using a userspace server is puffs [8]. The Pass-to-Userspace Device, or *pud*, driver in NetBSD uses the same principle, but instead of providing a file server, it attaches to a device major number. A block or character device request made to a pud-registered device is proxied to the server in userspace, handled, and the result of the operation is passed back to the original caller. The caller is not aware if the request was handled internally by the kernel or by a userspace server.

By constructing a server which registers device majors with the host kernel and maps the incoming requests to driver running inside the rump kernel, we can achieve integration with the host. This approach is similar to the rump file system p2k layer [10], which maps incoming puffs operations to rump kernel vnode operations for running kernel file system servers in userspace. For the server to pass the request from pud to the driver in the rump kernel, it calls the respective vnode operation for the device special vnode in the rump kernel, and this causes specfs to call the device entry point. For example, if we got a devsw open request, we call `RUMP_VOP_OPEN()`, for a read request `RUMP_VOP_READ()`, and so forth. The reason the rump kernel driver is called via the vnode interface instead of the device switch directly is of a pragmatic nature: the interfaces required to do so were readily exported by the rump infrastructure, so no extra effort was needed. The proxy is illustrated in Figure 2.

Most drivers can be integrated in this manner, but for drivers without a device node this is not possible. Examples of drivers in this class are network interface drivers. Still, ad hoc integration solutions, like the use of `/dev/tap`, may be attempted. We have not currently implemented such an approach.

A problem with the current implementation is the lack of devfs support in NetBSD. Each server must register the device majors it supports and then create the appropriate device nodes. Even though the rump kernel might be providing a ”well-known” driver instance, such as *sd0*, the device numbers for sd0 may already be bound in the kernel, even though there might be no sd0 configured in the system. Now, since it is possible for the host kernel to attach sd0 while the pud server is running, the well-known `/dev/sd0x` device nodes cannot be used. Instead, the server must use pseudo-names such as `/dev/rumpsd0x`. Another implication is that currently it is not possible to write a generic pud-to-rump proxy, but rather each rump driver must be supported by a pud server which is aware of the device name and minor number mapping conventions. These issues will hopefully be addressed when NetBSD grows devfs support.

3 Pseudo Devices

As mentioned in the introduction, pseudo devices are drivers which are not backed by hardware devices.

The current list of supported pseudo device drivers is:

- Berkeley Packet Filter (bpf)
- Cryptographic Disk Driver (cgd)
- Device Mapper / LVM (dm)
- netsmb (used by the SMB file system client)
- RAIDframe (raidframe)
- Entropy Collector / RNG (rnd)
- System Monitor (sysmon, framework for e.g. watchdogs and environmental sensors)

The curious case in the list is netsmb. It would seem to belong to networking instead of devices. However, we classify it as a device for several reasons. First, it is not a networking domain, i.e. it does not supply `DOMAIN_DEFINE()` like true domains such as inet (`inetdomain`) or unix local (`unixdomain`). Second, it is accessed by userspace programs via entries in `/dev` instead of socket system calls. Therefore, we classify it as a device.

More pseudo devices can be easily supported by adding them to the build infrastructure. So far, this has been done for the above pseudo devices on a need basis.

3.1 Implementation

Since pseudo devices do not require direct access to hardware, supporting them in rump did not essentially extend support from what was already required for file systems or networking. The only exception was having to include the kernel autoconf code in rump. However, that worked without problems.

3.2 Uses

Beyond the obvious use in kernel development and standalone regression tests, pseudo devices in userspace have a number of application uses, depending a little on the device in question. For example, the cgd crypto driver [5] can be used to write applications which do encryption and decryption. Since the encryption policy is the one used in the kernel, it is possible to access and modify encrypted partitions been written by the kernel. It is also possible to write applications which produce an encrypted kernel-readable partition. One possible use is to include support for this in *makefs* [13] to produce encrypted installation file system images with a portable userspace application.

4 Hardware Drivers: A Case Of USB

Hardware devices differ from pseudo devices in the sense that they must be able to access the device hardware. This access is typically available only when the CPU is operating in privileged (kernel) mode. USB devices are an excellent candidate for userspace support, since the kernel USB stack readily exports USB device access to userspace via the USB generic driver, or *ugen*. After ugen attaches to a USB bus node, it provides access to the particular piece of hardware from userspace via the `/dev/ugen<n>` device nodes. What is important is that the protocol spoken from userspace to the kernel is by all major accounts the same as what the kernel uses internally. This allows preexisting kernel drivers to talk to ugen without any complex protocol translation.

Our main goal with USB support was to show it is possible to do kernel device driver development safely and conveniently in a process. The two subproblems we had to solve were being able to access the hardware and to integrate with the device autoconfiguration subsystem and be able to supply a meaningful device configuration.

4.1 Other Passthrough Solutions

Various device driver ”passthrough” mechanisms to unprivileged domains have been attempted and already exist. For example, Xen on NetBSD supports PCI passthrough to DomU's. Doing so gives the unprivileged guest access to the machines physical memory space, and may be undesirable [1], as programming errors will bring the machine down.

The qemu [3] machine emulator includes support for USB hardware. Not only does it emulate some popular hardware devices such as a mouse, it also includes a host device mode, where devices from the host are presented to the guest operating system running inside qemu. On BSD systems the passthrough happens using ugen. For purposes of being able to use the device in another OS in case your host OS does not support it, this is convenient. However, for the purposes of driver development, this adds complexity including but not limited to long restart cycles [7].

4.2 Structure of USB

At the root of USB topology is a USB host controller. It controls all traffic on the USB bus. All devices access the bus through the host controller using an interface called USBDI, or USB Driver Interface. This, along with ugen, is a detail which makes USB suitable for userspace drivers: we merely need to implement a userspace host controller which maps USBDI to ugen instead of having to care about all bus details.

```
pain-rustique> dmesg | grep ugen2

ugen2 at uhub4 port 3
ugen2: Apple iPod, rev 2.00/1.00, addr 3

pain-rustique> ./sdread probe

Copyright (c) 1996, 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005,
    2006, 2007, 2008, 2009, 2010
    The NetBSD Foundation, Inc.  All rights reserved.
Copyright (c) 1982, 1986, 1989, 1991, 1993
    The Regents of the University of California.  All rights reserved.

NetBSD 5.99.24 (RUMP-ROAST) #0: Wed Feb 10 13:28:37 EET 2010
        pooka@pain-rustique.localhost:/usr/allsrc/src/sys/rump/librump/rumpkern
root file system type: rumpfs
mainbus0 (root)
ugenhc2 at mainbus0
usb0 at ugenhc2: USB revision 2.0
uhub0 at usb0: vendor 0x0000 product 0x0000, class 9/0, rev 0.00/0.00, addr 1
uhub0: 1 port with 1 removable, self powered
umass0 at uhub0 port 1 configuration 1 interface 0
umass0: Apple Computer product 0x1301, rev 2.00/1.00, addr 2
umass0: using SCSI over Bulk-Only
scsibus0 at umass0: 2 targets, 1 lun per target
sd0 at scsibus0 target 0 lun 0: <Apple, iPod, 2.70> disk removable
sd0: 968 MB, 30 cyl, 255 head, 63 sec, 2048 bytes/sect x 495616 sectors

pain-rustique>
```

Figure 3: **USB mass storage autoconfiguration output**. We see *ugenhc2* (the number "2" denotes `/dev/ugen2`) attach to mainbus. sd@scsibus@umass is probed to be found in ugen2.

The rump host controller is called *ugenhc*. Calling the autoconf match routine of the device ugenhc<n> results in the ugenhc driver trying to open `/dev/ugen<n>` on the host. If the open is successful, the kernel has attached a device to the respective ugen instance and ugenhc can return a successful match. After this, the ugenhc driver is attached, along with a usb bus and a usb root hub. The root hub then explores which devices are connected to it, causing the probes to be delivered first to ugenhc and through /dev/ugen to the kernel to the actual hardware. Figure 3 contains a "dmesg" of the process.

By default, ugen attaches at a low priority if no other driver claims the hardware, but this can be changed to make ugen attach at the highest priority claiming all USB hardware, or it can be made to claim hardware attaching to specific ports using the standard kernel configuration specification language.

4.3 Defining Device Relations with Config

Device autoconfiguration [16] is a central part of the NetBSD kernel. The device configuration determines the relationship of device drivers in the system. This relationship is expressed in a domain specific language (DSL). The language is divided into two parts: a global set of descriptions on which drivers can attach to which busses, and a system-specific configuration of what hardware is expected to be present and how this particular configuration should allow devices to attach. For example, even though the USB bus allows a USB hub to be attached to another hub, the device configuration might allow a USB hub to be attached only to the host controller root hub and not other hubs.

NetBSD is still in the process of transforming from the completely monolithic kernel to one which supports kernel modules. Currently the practice is to write the device configuration, have it translated into C tables by an utility called *config*, compile the generated tables into the kernel, and use the configuration information at runtime. However, in its current state, config takes care of all issues related to building a kernel, including generating the necessary makefiles from the configuration file. This is not granular enough for a single device driver.

Any device drivers loadable as kernel modules have their own ad-hoc way of specifying the compilation result of the config DSL in C. Handcoding the relationships in C lacks the sanity checks performed by the config utility. Additionally, it is very easy to make mistakes when

```
ioconf ums

include "conf/files"
include "dev/usb/files.usb"
include "rump/dev/files.rump"

# USB HID device
uhidev* at uhub? port ?
    configuration ? interface ?

# USB Mice
ums*    at uhidev? reportid ?
wsmouse* at ums? mux 0
```

Figure 4: **USB mouse device configuration**

manually describing the relationships. And ultimately, debugging any errors at runtime is extremely cumbersome and code reading is the best way of debugging, although it is by no means simple either.

We added an *ioconf* keyword to the config DSL. This keyword instructs config to generate the runtime loadable device configuration description. This in turn makes it possible to describe rump component device configurations in the familiar config DSL and preserve the pre-runtime safety checks. The created tables must still be loaded manually in the component constructor, but in the future we hope to improve config to produce the necessary code to load the tables automatically. Also, each rump is currently limited to including one ioconf statement. This is because the devices are described starting from root, and multiple ioconf's would result in duplicate definitions being loaded at runtime. This is another issue we hope to address soon.

An example of a configuration file describing a system configuration attaching a USB mouse is presented in Figure 4 and a similar one for USB mass memory disks is shown in Figure 5.

4.4 DMA and USB

As is well-known, direct memory access (DMA) allows devices to be programmed to access memory directly without involving the CPU. Being able to freely program the DMA controller to read or write any physical memory address from an application is of course a security and stability issue and should not be allowed.

Due to USBDI, we do not have to worry about USB drivers attempting to perform actual DMA operations, since they are done by the host controller. However, drivers will still allocate DMA-suitable memory to pass to the host controller in hopes of the host controller being able to perform the DMA operations. We must be able to correctly emulate the allocation of DMA-safe memory.

```
ioconf umass

include "conf/files"
include "dev/usb/files.usb"
include "dev/scsipi/files.scsipi"
include "rump/dev/files.rump"

# USB Mass Storage
umass*  at uhub? port ?
    configuration ? interface ?

# SCSI support
scsibus* at scsi?
sd*     at scsibus? target ? lun ?
cd*     at scsibus? target ? lun ?

# ATAPI support
atapibus* at atapi?
sd*     at atapibus? drive ? flags 0x0000
cd*     at atapibus? drive ? flags 0x0000
```

Figure 5: **USB umass storage configuration**. This configuration supports both SCSI and ATAPI attached disk drives and CD/DVD devices. Technically, the actual drives (sd & cd) should be in a different component so they could be loaded independently.

In NetBSD, all modern device drivers use the machine independent *bus dma* [15] framework for managing DMA memory. bus dma specifies a set of interfaces and structures which different architectures and busses must implement for machine independent drivers to be able to manage DMA memory. We can plug into the bus dma interface in rump to provide our userspace DMA memory management routines. As device driver I/O is currently the ugen host controller driver doing read/write on /dev/ugen, there is no DMA happening and a bus dma implementation can be handled simply with virtual memory and malloc.

A problem for us in the bus dma interface specification is that it allows the use of macros in the `bus.h` header for implementing the interface specification on a given CPU architecture. As long as the leaked implementation does not contain privileged CPU instructions, this is not a show stopper for rump. However, this leaks the implementation of the interface to the caller which in our case is an unmodified kernel driver. To make the driver link against rump, the rump implementation must match the leaked interface. One option would be to provide stubs with the correct linkage for all the bus dma interface implementations. However, this would be a considerable amount of work given the amount of CPU architectures supported by NetBSD. An easier way is to introduce a blanket `machine/bus.h` which provides a pure

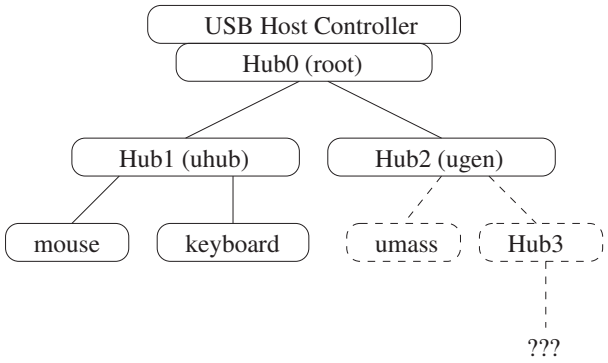

Figure 6: **Attaching USB Hubs**: hub1 on the left is attached properly as uhub and probing the rest of the bus is possible. However, on the right Hub2 is attached as ugen. This makes attaching the direct children and further probing of the bus impossible.

function-style implementation of bus dma and is found from the include path before the real `machine/bus.h`. We use this blanket header for all architectures that have macros in their implementation (currently all except i386 and amd64).

4.5 USB Hubs

A feature of the ugen interface is that an instance of ugen can access only the specific USB device it is attached to; this prevents for example security issues by not allowing access to any device on the USB bus. For USB functions, such as mass memory or audio, this does not have any implications, as we are only interested in accessing the particular device. However, USB hubs expose other USB devices (functions or other USB hubs) further down the bus. If a USB hub is attached as ugen, it is possible to detect that devices are attached to the hub, but it is not possible to access any devices after the USB hub, including ones directly attached to it – all ugen access will be directed at the hub. Figure 6 illustrates ugen concealing all nodes on the bus after it.

In practice hub-as-ugen is not an issue, but rather one needs to be aware of it. Already, the host ugen driver attaches to USB hubs, so hub support must be present in the host kernel for rump USB access to work at all. The only implication is that for the full bus to be probed, the hub driver must attach at a higher priority than ugen. The kernel variable `uhub_ubermatch` takes care of this when set to non-zero and forces hubs to attach at a higher priority than any other drivers. The difference between correct and incorrect kernel hub match is illustrated in Figures 8 and 7 (output has been trimmed for presentation purposes).

kernel probe:

```
uhub5 at uhub2 port 1: OnSpec Generic Hub
uhub5: 2 ports with 0 removable
ugen2 at uhub5 port 1
ugen2: Alcor Micro FD7in1
ugen3 at uhub5 port 2
ugen3: CITIZEN X1DE-USB
```

rump probe:

```
ugenhc2 at mainbus0
usb2 at ugenhc2: USB revision 2.0
uhub2 at usb2
umass0 at uhub2
umass0: Alcor Micro
umass0: using SCSI over Bulk-Only
scsibus0 at umass0
sd0 at scsibus0
sd0: 93696 KB, 91 cyl, 64 head, 32 sec,
    512 bytes/sect x 187392 sectors
sd1 at scsibus0
sd1: drive offline
ugenhc3 at mainbus0
usb3 at ugenhc3: USB revision 2.0
uhub3 at usb3
umass1 at uhub3
umass1: using UFI over CBI with CCI
atapibus0 at umass1
sd2 at atapibus0 drive 0
sd2: 1440 KB, 80 cyl, 2 head, 18 sec,
    512 bytes/sect x 2880 sectors
```

Figure 7: **Attaching a USB device with an internal hub *with* uhub_ubermatch**. sd1 does not have media inserted and is therefore offline. sd2 is a USB floppy driver.

---

kernel probe:

```
ugen2 at uhub1 port 1
ugen2: OnSpec Generic USB Hub
```

rump probe:

```
ugenhc2 at mainbus0
usb2 at ugenhc2: USB revision 2.0
uhub2 at usb2
uhub2: 1 port with 1 removable
uhub3 at uhub2 port 1: OnSpec Inc.
uhub3: 2 ports with 0 removable
uhub4 at uhub3 port 1: OnSpec Inc.
uhub5 at uhub3 port 2: OnSpec Inc.
```

Figure 8: **Attaching a USB device with an internal hub *without* uhub_ubermatch**.

4.6 Other Hardware Families

In this section we speculate on how easy it is to extend the concept of rump hardware device driver support to other hardware families. The main challenge is in exporting kernel interfaces which are general enough to be useful, but safe enough to not lead to a host system compromise or crash.

Some other driver families beyond USB export ugen-like interfaces to userspace. An example of this is the *uk* (unknown) driver for SCSI bus. The potential of these driver families should be analyzed and root drivers similar to the ugen host controller can be created.

For device families such as PCI support is centrally a question of DMA. Device drivers running in kernel mode have full access to system memory and can program any address they like on the DMA controller. However, when programmed from unprivileged mode, the DMA controller should be programmed to access only physical memory belonging to the process in question.

The easiest solution is to ignore the DMA problem and allow it for root processes. This is a ”95%” solution, which makes userspace development available in the very near future and makes development almost crashproof. The developer must just be aware to exercise extra caution when doing device-to-host DMA write. However, since there is no guaranteed protection, this approach cannot be used for running untrusted drivers.

One approach would be to create a new language which could be passed from userspace to a kernel driver. The language would contain instructions on how to program the DMA controller. However, since the DMA controllers are programmed by NetBSD drivers in a non-semantic manner as just a set of device accesses, it is difficult to extract which part is actually programming the host address and should be put under scrutiny. Changing DMA programming would require very heavy modifications to the driver base, and is not currently on our list of things to attempt.

As a third option, the problem can be solved with the use of an IO-MMU which can enforce memory access permissions [11]. This requires hardware capabilities not present in every system, but would be an effective measure where available.

5 Conclusions

We presented rump device drivers, unmodified kernel device drivers running in userspace, and discussed their design and implementation. We found that non-hardware device drivers (”pseudo devices”) are directly usable in userspace. Hardware device drivers require kernel assistance for hardware access. For the currently supported USB driver family, this was readily available in NetBSD with the *ugen* driver. We adapted the config tool used for configuring the kernel to be able to describe the driver relationships in a rump device component.

The work done on rump also benefits other uses in NetBSD. For example, the new features of config can benefit kernel modules. On the USB front, the hub issue was likewise a problem for qemu USB driver passthrough mode, and once NetBSD grows support for ”usermode OS” operation, the ugen host controller will be usable for attaching host USB devices.

In the future we wish to extend hardware driver support beyond USB, and already outlined and evaluated the possible steps for this. Additionally, more pseudo devices should be supported. Since in most of the cases the only task is to write a Makefile for producing the component library (and possibly code to autogenerate the file system device nodes), we plan to look into making this automatic with the help of config(1).

Availability

Rump device drivers are available in NetBSD-current and will be a part of NetBSD 6.0. Device components are installed in binary form as libraries in `/usr/lib` with the prefix *rumpdev* (e.g. `librumpdev_usb` or `librumpdev_wscons`).

Acknowledgments

Thanks go to Roland Dowdeswell for the idea with cgd and Quentin Garnier for help with config(1). Special thanks go to anyone who has ever submitted a bug report for rump.

References

[1] NetBSD/xen Howto, Referenced Feb 2nd, 2010. http://www.NetBSD.org/ports/xen/howto.html.

[2] Rump on non-NetBSD Operating Systems, Referenced Feb 2nd, 2010. http://www.NetBSD.org/~stacktic/rumpabroad.html.

[3] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proc. of USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[4] W. V. Courtright II, G. Gibson, M. Holland, L. N. Reilly, and J. Zelenka. Raidframe: A rapid prototyping tool for raid systems, 1997.

[5] R. C. Dowdeswell and J. Ioannidis. The cryptographic disk driver. In *Proc of. USENIX Annual Technical Conference, FREENIX Track*, pages 179–186, 2003.

[6] G. C. Hunt. Creating user-mode device drivers with a proxy. In *Proc. of the USENIX Windows NT Workshop*, 1997.

[7] A. Kantee. Kernel development in userspace - the rump approach. In *BSDCan 2009*.

[8] A. Kantee. puffs - Pass-to-Userspace Framework File System. In *Proc. of AsiaBSDCon*, pages 29–42, 2007.

[9] A. Kantee. Environmental Independence: BSD Kernel TCP/IP in Userspace. In *Proc. of AsiaBSDCon*, pages 71–80, 2009.

[10] A. Kantee. Rump file systems: Kernel code reborn. In *Proc of. USENIX Annual Technical Conference*, pages 201–214, 2009.

[11] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proc. of the 6th OSDI*, pages 17–30, 2004.

[12] S. McCanne and V. Jacobson. The BSD packet filter: a new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference*, pages 259–269, 1993.

[13] L. Mewburn and M. Green. build.sh: Cross-building NetBSD. In *Proc. of USENIX BSD Conference*, pages 47–56, 2003.

[14] G. Oster. Porting RAIDframe to NetBSD. private communication, January 2010.

[15] J. Thorpe. A machine-independent DMA framework for NetBSD. In *Proc. of USENIX Annual Technical Conference (FREENIX track)*, pages 1–12, 1998.

[16] C. Torek. Device Configuration in 4.4BSD, December 1992.