NetBSD/usermode

Reinoud Zandijk, MSc.

NetBSD foundation

17 March, 2013 at 12.17 p.m.

Summary

NetBSD/usermode adds a new type of system virtualisation to NetBSD. It allows one to run a NetBSD kernel and its userland as just another process on the host, complete with console, networking, audio and a virtual display. In this paper we visit some of the design challenges and our solutions for them. Some benchmarks are also produced.

Keywords: system virtualisation

1. Introduction

The goal of the project is to provide a complete NetBSD kernel and its userland running inside a POSIX compliant operating system with minimal to no kernel support.

Each way of virtualisation has its strong and weak points. NetBSD/xen is very well suited for high security applications and virtual servers, whereas Qemu is more a machine simulator and suited for testing out hardware that the developer has no access to. VirtualBox can be seen as a best-of-both-worlds solution though depends heavily on kernel support and can in conjunction with CPU support provide both performance and isolation.

Compared to these other virtualisation methods, NetBSD/usermode stands out for running at full speed without the need for a dedicated setup nor specific processor support. As expected it is limited to the same architecture of the host running it.

NetBSD/usermode provides the user a complete virtual machine that behaves just like any other machine running NetBSD. It has either a tty console where it resembles a machine console over a serial line or a graphical console displayed with a VNC client. The virtual machine can have full network access, play audio, mount NFS partitions, compile stuff etc. It can also be run multiple times in parallel making it suitable for distributed systems testing.

NetBSD/usermode is most userfull as a research platform or for regression testing. It provides excellent debugging since its just another process running on the host and can be debugged using gdb complete with all its features wich is quite a relief since all datastructures can be examined, breakpoints be set etc. For debugging it is best compared to having a dedicated machine with both a serial console and a KGDB link.

Work on NetBSD/usermode started around 2010 by Jared D. McNeill. He managed to compile it as a normal program and to ‘boot’ it until the disc had to be mounted by inserting lots of empty functions, just enough to get it going. He then abandoned the project and after some time of neglect it was picked up again by the both of us. Jared focused on most of the IO subsystems while I focussed on the internals like virtual memory, process switching, interrupts etc.

NetBSD/usermode is still in active development. Focus is now on direct support

for ATAPI/SCSI devices[1], X support[2] and of course cleaning up. Support for direct USB support is also investigated but is not yet seen as a priority.

For now, NetBSD/usermode only supports the i386 and amd64 architectures on NetBSD. Other architectures like arm32 are in the planning but haven't been started on due to lack of suitable hardware. Jared has also done some initial work on trying it to build and run on Linux. When the *build.sh* build system problems are solved so that NetBSD/usermode can be build with it this ought to become a lot easier.

2. NetBSD/usermode architecture

As NetBSD/usermode tries to run as much kernel code as possible to make it as close to a real test and develop machine. It uses all the kernel internals like the NetBSD virtual memory manager UVM/UBC. Devices are not simulated for the 'normal' hardware drivers but are implemented using the host kernels userland interfaces to provide the services needed.

The overall memory space architecture has been a point of controversy and debate inside the NetBSD community. Especially our choice to go for a unified kernel and userland virtual memory space has been the issue of a hot debate for the preferable way is to use separate overlapping memory spaces so both userland and kernel can use the full extent of virtual memory space.

This full memory separation is especially interesting for systems providing more memory than the processor can map simultaneously like i386. The separation also gives the additional security benefit that userland can never, not even by brute force, inspect suspected kernel memory ranges.

Our choice for a unified virtual memory space was a purely pragmatical one. Separating the two memory spaces entirely is nearly impossible without kernel and processor support and one of the design goals was to avoid that. Not all processor architectures support the non-unified virtual memory spaces anyway.

In the following subsections we'll highlight selected issues and how we solved them.

2.1. System calls

System calls are normally done using privileged instructions that trap the kernel. On taking this trap, the process state is saved, the system call number and its arguments are extracted[3] and the appropriate function call is taken. The Results, if any, are stored in the process state and the process continues.

One of the major goals of the project, as stated before, is that running the NetBSD/usermode kernel does not need kernel modifications nor special hardware support. This poses a serious problem since how should the kernel distinguish between a system call made by the usermode kernel process and a system call issued by its userland.

Our initial solution was to early fork the process and run the rest of the NetBSD/usermode kernel as a ktracee of its parent 'hypervisor', not unlike gdb works. This highlighted a problem with ktrace in general. We could get a nice overview of the system calls made, complete with their arguments and the result codes but the ktrace architecture does not allow system call redirection or modification. An extension was attempted by Jared but it only gave rise to more problems. In the end we abandoned this path.

Concurrently with Jareds work on

[1] like CD or DVD recorders

[2] X does run but the keyboard and mouse need work

[3] pre- and post-processed if necessary for emulations

ktrace, an alternate solution was explored by me to make use of the illegal instruction signal. For this purpose a special userland was built that replaced the normal system call instructions with undefined instructions. On receiving the signal the offending instruction is inspected and if its recognised its treated as a system call and acted upon.

To support normal precompiled userlands, a concession had to be made on kernel support. Since the process's memory space is split into a kernel part and a userland part, a kernel module was made not unlike other emulation kernel modules. It implements a single system call with a address range as argument in which subsequent system calls of the calling process are prohibited and an illegal instruction should be issued instead.

This concession does not mean NetBSD/usermode won't run on generic POSIX-like systems like Linux anymore since the undefined instruction trap solution is still valid and used and the modified NetBSD userland solution still works.

2.2. Virtual memory

NetBSD can only run on systems with virtual memory support. Normal processors translate a virtual address to a physical address trough a cache, the translation lookaside buffer. When this gives a cache miss the processor then either generates an interrupt or walks tables to lookup the translation itself to add to this lookup cache. Since page faulting on NetBSD/usermode needs to be handled in software anyway, a TLB-only solution was chosen with the SIGSEGV signal acting as the generated interrupt.

On receiving this signal the NetBSD/usermode kernel checks the processes pmap[1] to see what it should become and then maps a piece of physical memory substitute on the offending place using mmap.

As there is no real physical memory to map around, a substitution has to be created. This substitute must be able to be mapped multiple times in the virtual memory space using specified credentials for each mapping. Since it must be mappable multiple times, sources like anonymous memory fall off since on each new mapping it would be created anew. Using /dev/mem is an option but was abandoned since it would mean that the physical memory needed to be completely mapped in the host processes memory space too. This would significantly reduce the amount of virtual memory space of the virtual machine. This could be avoided by using the anonymous memory of a parent forked processes but this was seen as too freaky and possibly non-portable.

A more pragmatic and sane solution was found in using a disc file as virtual memory.

2.3. LWP switching

In NetBSD, each process, including the kernel, is a collection of LWPs[2]. Switching between these threads is machine dependent. Next to preserving the LWP stack and registers some operating systems also dictate other measures like calling a system call to signal for signal mask changes etc. To cater for this, the choice was made to use the POSIX getcontext/setcontext/switchcontext calls. This interface predates pthread and allows for multi threading. Converting the code to use pthread could be done but it would kind of messy since we explicitly need to switch between threads and not let the host choose one thread for us. It might be worth investigating though.

Contexts are created using makecontext

[1] a pmap holds the architecture specific memory mapping administration for a process including the lookup tables if the processor can look it up itself

[2] light weight processes, also more commonly known as threads

and are patched up in a cpu architecture dependent way to make sure the stack is setup correctly and no information is leaked to the new process trough the other registers.

One problem we encountered with switching with setcontex/getcontext is the lack of thread specific variables like the often used ‘curcpu’. To set this atomically i deviced a dedicated trampoline that has all signals switched blocked. This way there can’t be a signal occuring in the small time between setting curcpu and switching the context. Unlikely as it might seen it actually happened a lot.

2.4. Interrupts and I/O

Just like in a physical computer, NetBSD/usermode needs interrupts for basic things like preemptive sheduling, keeping time, page faults and I/O operations. All interrupts are implemented using signal handlers and the sigalt stack.

Since this sigalt stack is shared by all signals it needs to be switched from as soon as possible. Since NetBSD/usermode can’t rely on the running context’s stack space either[1], each lwp gets assigned an extra stack space. This allows for recursive signal handling.

Since devices in NetBSD/usermode use the host operating system’s userland interfaces, all files and host devices are preferably opened asyncronously and set to generate an SIGIO signal when possible. When generating a SIGIO signal is not possible, the drivers have to fall back to polling and/or piggy-back on other SIGIOs.

On a SIGIO signal reception, all the SIGIO capable devices are visited so they can check if they were the cause and take action. Since multiple SIGIO signals can be folded and issued as just one signal, all drivers are visited twice regardless of a driver recognizing it.

3. Performance

The performance of NetBSD/usermode hasn’t been stated as a distinct development goal but care has been taken to get the performance on par with the host machine. Various tests have been performed on NetBSD/amd64 (i7-920) with 12 Gb of memory and the results are as to be expected.

The number of syscalls per second is a factor 22 lower as on the native machine[2]. This drop is expected and is a direct result of the signal generating code overhead in the host OS. The original signal(3) code was never designed to be high-volume and is thus not implemented as such. A good amount of speed increase could be gained if this code is tackeled.

Disc IO has been tested by using dd(1) to copy a file on the host OS to /dev/null with bs=64k. In usermode the raw disc device was used to directly access the file without inducing a 2nd file system layer. The results were as expected the same, a rough 80 Mb/sec.

As an overall performance test a medium sided program was compiled. For comparisation, the timing is split in a ‘configure’ phase and a real compilation phase. In NetBSD/usermode, the ‘configure’ phase takes significantly more time to run, a whopping 11 seconds compared to 3 seconds natively. The compilation phases take an equal time of 10 seconds on both.

The difference in time during the ‘configure’ phase is more complex. After a long discussion with other NetBSD developers we came to the conclusion that it is most likely due to a higher page fault latency and thus an effectively lower memory bandwidth on new

[1] Especially ld.so’s stack space is nearly nothing.

[2] 13.67 million/sec versus 0.62 million/sec system calls

starting applications that starts to play up. This same behaviour can be seen on memory bandwith stricken machines like some cheap ARM machines that despite their megahertz figures have to cope with a 16 bit RAM bus.

4. Conclusions

NetBSD/usermode has proven to be feasable and met the expectations on performance we set out. It proved usefull for debugging kernel internals. With some additional work on providing bus-level access to busses like USB, ATAPI/SCSI and maybe one day even PCI, NetBSD/usermode can also prove to be a great platform for device driver development.