USENIX Association

Proceedings of the BSDCon 2002 Conference

San Francisco, California, USA
February 11-14, 2002

Porting NetBSD to the AMD x86-64[*]: a case study in OS portability

Frank van der Linden
Wasabi Systems, Inc.
fvdl@wasabisystems.com

Abstract

NetBSD is known as a very portable operating system, currently running on 44 different architectures (12 different types of CPU). This paper takes a look at what has been done to make it portable, and how this has decreased the amount of effort needed to port NetBSD to a new architecture. The new AMD x86-64 architecture, of which the specifications were published at the end of 2000, with hardware to follow in 2002, is used as an example.

1 Portability

Supporting multiple platforms was a primary goal of the NetBSD project from the start. As NetBSD was ported to more and more platforms, the NetBSD kernel code was adapted to become more portable along the way.

1.1 General

Generally, code is shared between ports as much as possible. In NetBSD, it should always be considered if the code can be assumed to be useful on other architectures, present or future. If so, it is machine-independent and put it in an appropriate place in the source tree. When writing code that is intended to be machine-independent, and it contains conditional preprocessor statements depending on the architecture, then the code is likely wrong, or an extra abstraction layer is needed to get rid of these statements.

1.2 Types

Assumptions about the size of any type are not made. Assumptions made about type sizes on 32-bit platforms were a large problem when 64-bit platforms came around. Most of the problems of this kind had to be dealt with when NetBSD was ported to the DEC Alpha in 1994. A variation on this problem had to be dealt with with the UltraSPARC (sparc64) port in 1998, which is 64-bit, but big endian (vs. the little-endianness of the Alpha). When interacting with datastructures of a fixed size, such as on-disk metadata for filesystems, or datastructures directly interpreted by device hardware, explicitly sized types are used, such as *uint32_t, int8_t,* etc.

1.3 Device drivers

BSD originally was written with one target platform (PDP11, later VAX) in mind. Later, code for other platforms was added, and 4.4BSD contained code for 4 platforms. NetBSD is based on 4.4BSD, but has steadily expanded the number of supported platforms over the years. As more platforms were added, it became obvious that many used the same devices, only using different low-level methods to access the device registers and to handle DMA. This led to, for example, 5 different ports having 5 seperate drivers for a serial chip, containing nearly identical code. Obviously, this was not an acceptable situation, with ports to new hardware being added every few months.

To remedy this situation, the *bus_dma* and *bus_space* layers were created [1], [5]. The *bus_space* layer takes care of accessing device I/O space, and the *bus_dma* layer deals with DMA access. For each NetBSD port to a new architecture, these interfaces must be implemented for each I/O bus that the machine uses. Once that is done, all device drivers that attach to such an I/O bus should compile and work without any extra effort.

2 Machine-dependent parts

Of course, not all code can be shared. Some parts deal with platform-specific hardware, or simply need to use machine instructions that a compiler will never generate. A few userspace tools will also be platform spe-

[*] x86-64 is a trademark of Advanced Micro Devices, Inc.

cific. Here is a summary of he most important machine-dependent parts of NetBSD that need to be dealt with when porting NetBSD to a new platform.

2.1 Toolchain

First and foremost, a working cross-toolchain (compiler, assembler, linker, etc) is needed to bootstrap an operating system on to a new platform. The GNU toolchain has become the de-facto standard for open source systems, and NetBSD is no exception to that rule. Since the ELF binary format is used by almost all NetBSD ports (and should be used by any new ports), as well other operating systems such as Linux, making the GNU toolchain work for NetBSD usually involves not more work than creating/modifying a few configuration files. The exception being the case where the target CPU isn't supported at all yet, which makes for a much greater effort.

2.2 Boot code

The boot code deals with loading the kernel image in to RAM and executing it. It interacts with the firmware to load the image. The effort needed to write the boot code largely depends on the functionality offered by the firmware. Often, the limited capabilities of the firmware are only used to load a second stage bootloader, containing code that is more sophisticated in dealing with filesystems on which the kernel image may reside, and the file format(s) that the kernel may have.

2.3 Traps and interrupts

Trap and interrupt handling is obviously a highly machine-dependent part of the kernel. At the very least, the entry points for these must have machine-dependent code to save and restore CPU registers. Furthermore, CPUs will have different sets of traps, needing specific care.

2.4 Low-level VM / MMU handling

Memory management units (MMUs) tend to be quite different from CPU to CPU. They may even be different within one family of processors (for example, the PowerPC 4xx series has an MMU that differs significantly from the 6xx series). They may also be very different in what they expose to the hardware. MMUs may for example have a fixed page table structure that they walk in hardware, or leave many more operations up to the software, exposing translation lookaside buffer (TLB) misses. The low-level virtual memory code in all 4.4BSD-derived systems is called the *pmap* module, a name taken from the Mach operating system, whose virtual memory (VM) system was used in 4.4BSD.

2.5 Port-specific devices

Some platforms will have devices that are highly unlikely to appear on any other platform. These may include on-chip serial devices and clocks. Porting NetBSD to a new platform often involves writing a driver for at least one such device.

2.6 Bus-layer backend code

As mentioned above, the device code uses a machine-independent interface for I/O and DMA operations. The differences per platform are hidden underneath this interface, and the implementation of this interface deals with the platform specifics. This interface is heavily used in device drivers, so a small memory footprint and speed are important. Often, the interface is implemented as a set of macros or inline functions.

2.7 Libraries

In userspace, the C startup code and a few libraries will contain machine-dependent code. The library parts in question are mainly the system call interface in the C library, the optimized string functions in the same library, and specific floating point handling in the math library. There are a few other, lesser used areas, like the KVM library which deals with reading kernel memory. Lastly, shared library handling (relocation types) will likely be different from other platforms.

3 The x86-64 hardware

Before going into the specifics, a brief introduction to the AMD x86-64 architecture[2]is in order. The AMD x86-64 (codenamed "Hammer") architecture specification was released end of 2000. As of December 2001, no hardware implentation was publicly available yet (NetBSD/x86-64 was exclusively developed on the Simics x86-64 simulator made by VirtuTech). Since x86-64 is essentially an extension to the IA32 (or *i386*, as it is known in NetBSD) architecture, its predecessor will be introduced first below.

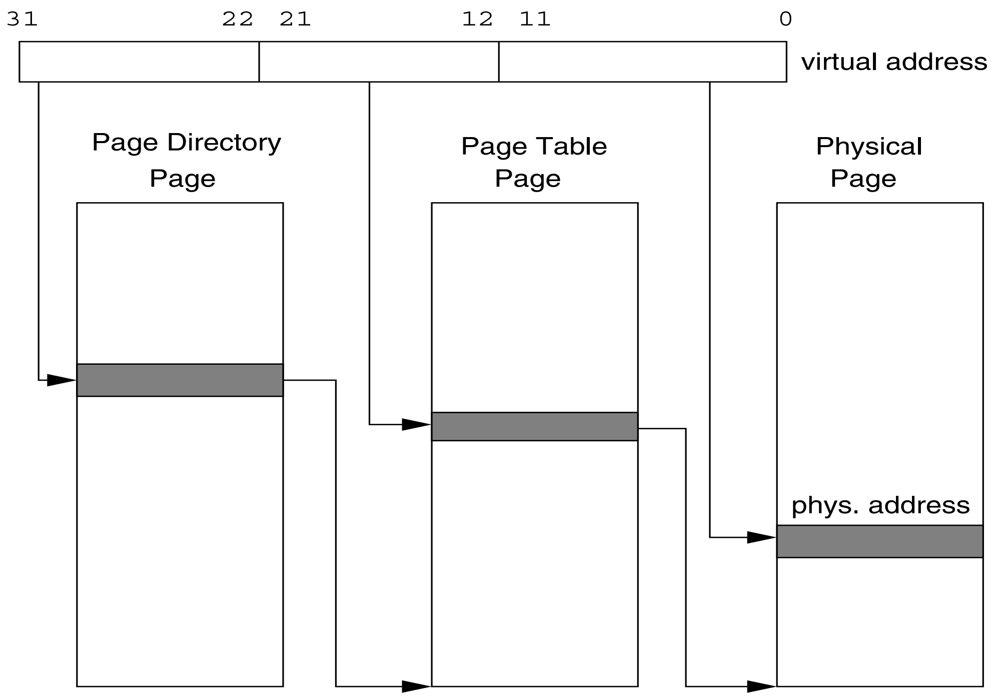

Figure 1: IA32 virtual address translation (4K pagesize). The entries in the page table and page directory are 32 bits wide.

3.1 The IA32 architecture

IA32 is the name that Intel gave to the 32-bit architecture that was originally introduced with its 80386 CPU, and has, through its application in the PC world, become the most popular CPU architecture today.

IA32 CPUs have the following features [4]:

- Seven 32-bit wide general purpose registers, 4 of which can also be used in 16-bit and 8-bit chunks
- A large set of instructions.
- From the 80486 and up, an on-chip floating point unit.
- From the Pentium III and up, Streaming SIMD Extensions (SSE), a class of instructions dealing with parallelized load/store and computation, targeted at graphics applications. SSE also adds a set of new registers, 64 bits wide, for use with SSE instructions. The Pentium 4 added yet more SSE instructions, and widended the SSE registers to 128 bits. These newer SSE instructions are known as SSE2.
- 32-bit wide addressing
- An MMU supporting the usual page protection schemes, and four gigabytes of virtual memory through a three-level pagetable. Different parts of a virtual address are used as indices into page table structures. These structures contain some information on the protection of the page, and point to the physical address of the page in which the structure to be indexed at the next (lower) level is contained. The lowest level of page tables contains the actual physical address to be referenced. Later CPUs also support an extended version of this scheme called Physical Address Extensions (PAE), enabling the use of more than 4G of physical memory. Virtual address translation for the IA32 architecture is shown in figure 1.
- Memory segmentation through descriptors, describing the type, base address and length of a section of memory. Memory descriptors (and other types) are stored in special tables.
- Segment registers that specify which descriptor is used to determine the location of loads/stores/execution.
- Trap/interrupt handling through a special array of descriptors, called the Interrupt Descriptor Table (IDT).
- Four different execution modes for programs: plain 16-bit mode (“real mode”, backward compatible to the 8086); 16-bit “protected mode” (16-bit mode

with protection); 32-bit “protected mode” (this is the mode that all modern systems use); and finally “virtual 8086 mode”, which runs old-style 16-bit programs in a virtual “real mode”, while the operating system is actually running in 32-bit protected mode.

3.2 General x86-64 extensions

The x86-64 architecture is essentially a 64-bit extension of the IA32 architecture. In addition to the legacy “real” (16-bit) and “protected” (32-bit) modes, it defines a “long” (64-bit) mode. In real mode and protected mode, it is fully compatible with the IA32 architecture. In long mode, it is capable of running 32-bit binaries without modification, and contains a number of extensions. This paper only discusses long mode, except where explicitly noted otherwise.

3.3 Registers

The general-purpose registers that already were present in the IA32 architecture were extended to 64 bits. Eight general purpose registers were added, yielding a total of fifteen (not counting the *esp* register, see figure 2). This addresses an often heard complaint about the IA32 architecture: it has very few general purpose registers. Additionally, eight SSE2 registers were added. For consistency, all general purpose registers can have their lower 16 bits or lower 8 bits addressed specifically in instructions, something that was only possible for four of the registers in the IA32 architecture.

To be backward compatible, computations and moves involving the lower 16- and 8-bit parts of the registers do not affect the upper bits. However, 32-bit operations are zero-extended. A special 64-bit immediate register move instruction was added to conveniently use 64-bit constants; 64-bit immediate values are not allowed in other instructions.

3.4 Memory management

The idea of the x86-64 being an extended IA32 architecture is also reflected in its memory management unit (MMU). The x86-64 has a 64-bit virtual address space, however, initial implementations of the architecture are specified to only use 48 out of these 64 bits, and 40 bits of physical address space. Addresses are specified to be sign-extended, essentially leading to a “hole” in virtual memory space that cannot be addressed. To enable the MMU to handle the translation of 48 bits of virtual address space, an extra page table level was added, in addition to the extra level that already had been added in later implementations of the IA32 architecture to support PAE (see figure 3, the dotted line inside the virtual address shows where PAE ends). So basically, the x86-64 page table scheme is the IA32 PAE scheme, with 512 instead of 4 page table pointer directory entries, plus a fourth level, called PML4. This similarity goes so far that the PAE feature must actually be specifically enabled as part of the steps to get the chip into long mode.

3.5 Other

Other notable features of the x86-64 architecture include:

- The possibility of addressing relative to the instruction pointer.
- Flat address space (the memory offsets specified through the code and data segment registers are ignored).
- Most hardware (system) data structures were extended to hold 64-bit addresses where needed.
- Fast, special cased instructions for system calls into the operating system. AMD had already introduced these with the K6 CPU, and they were extended to 64 bits.
- A few special registers were added, such as the registers that hold the entry point for the SYSCALL/SYSRET instructions, and the EFER, the Extended Feature Enable Register, which, amongst other things, contains a bit that enables long mode.

4 The actual port

Using the list of machine-dependent operating system parts in section 2, the work that was needed to port NetBSD to the x86-64 architecture will now be discussed.

4.1 Toolchain

When the work on NetBSD/x86_64 was started, the GNU toolchain already had some basic x86-64 support in it, which had been developed for Linux at SuSe, Inc. Shared library support wasn’t working yet, but

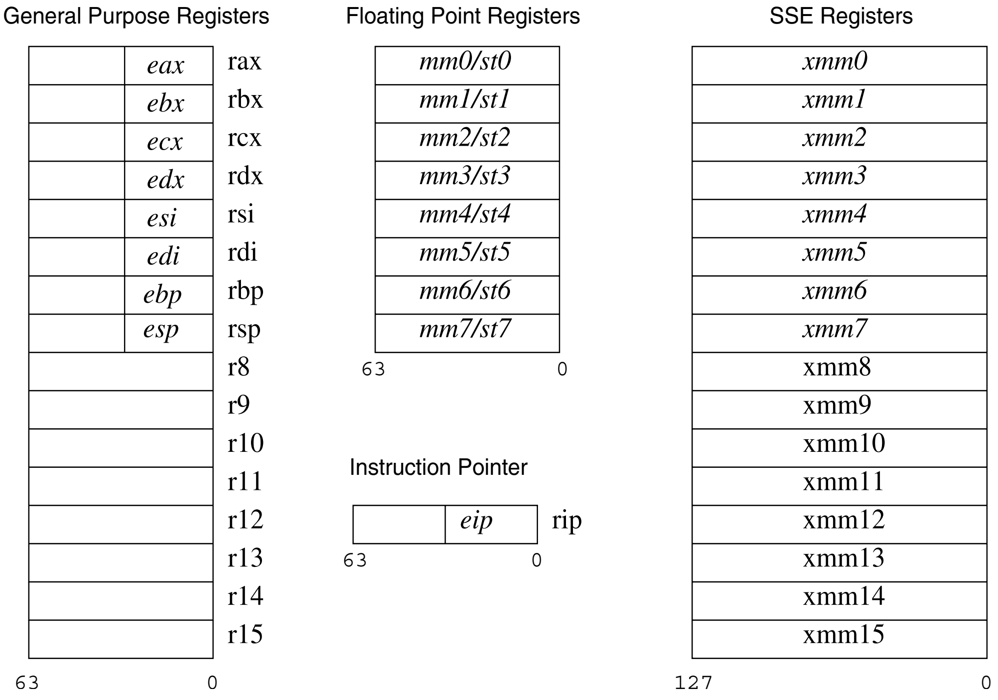

Figure 2: x86-64 registers. IA32 compatible registers are shown in italic

compiling, assembling and linking applications mostly worked. The application binary interface (ABI) had also been defined[3]. Adapting this code for NetBSD came down to just modifying/creating some configuration header files. Naturally, a few compiler and linker bugs were present in the OS-independent code of the toolchain, but this could be expected as the x86-64 code was quite young.

There are some ABI issues to consider. The x86-64 ABI defines four non-PIC code models:

- Small. All symbols in the program are assumed to be at virtual addresses in 32-bit range.
- Kernel. As above, but instead, all addresses are expected to be in negative 32-bit range, i.e. in the upper 32 bits of 64-bit memory. Kernels are often run in the upper region of virtual memory, and this code model was added to map a kernel in that area, without having to specify full 64-bit offsets in the code.
- Medium. Size and address of the code segment are expected to be in 32-bit range, but data can be the full 64-bit range
- Large. No restrictions on data or code addresses. The compiler has to generate code to only use indirect addressing via registers to be sure that the full 64-bit range is addressable.

There are similar models for position-independent code. By default, userspace programs are expected to use the "small" code model. Other code models weren't yet completely supported when the port was done, although at least the "large" model turned out to be stable after a few small modifications, and was used for the kernel.

4.2 Bootcode and bootstrap

Since there is no actual x86-64 hardware available yet, and no definitive firmware interface for the upcoming x86-64 machines has been specified yet, the bootcode had to deal with whatever the simulator provided. The simulator provided a normal PC BIOS interface, meaning that the NetBSD/i386 bootcode could almost be used as-is. However, the kernel image to be loaded is a 64-bit ELF binary for the x86-64 case. Making this work was a trivial modification, since the code to load 64-bit ELF binaries had already been made machine-independent and placed into the stand-alone library used by the bootcode of various NetBSD platforms.

The initial bootstrap code in the kernel (i.e. the first code to be executed in the kernel) naturally had

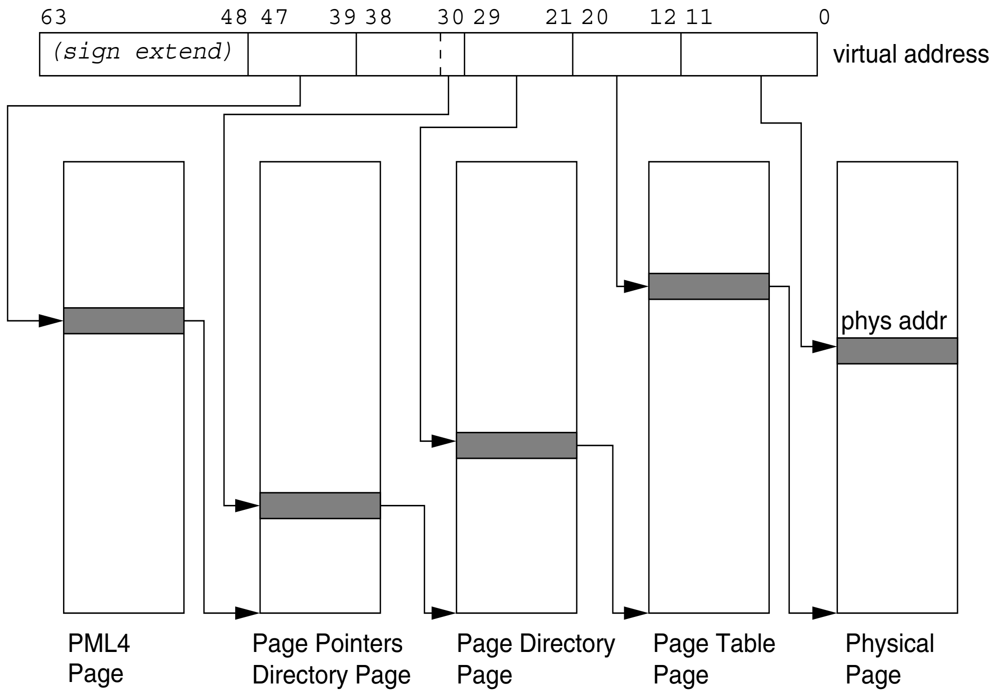

Figure 3: Virtual address translation in the x86-64 (4K pagesize). The entries in the page table structures are 64 bits wide.

to be written from scratch for this new NetBSD port, although it can be viewed as an extended version of that in NetBSD/i386. Since the x86-64 is fully IA32 compatible at power-on, it also needs to deal with getting the CPU out of 16-bit mode, in to 32-bit mode, and this time a couple of steps further, into 64-bit mode. The actual steps are:

1. Enable Physical Address Extensions
2. Set the LME (Long Mode Enable) bit in the EFER register
3. Point the %cr3 register at a prefabricated initial 4-level page table structure
4. Enable paging
5. The CPU is now running in a 32-bit compatibility segment. Fabricate a temporary Global Descriptor Table with a Long Mode memory segment, and jump to that segment
6. Because the kernel is mapped in the upper regions of memory, we could previously not address that region, as it lies well out of 32-bit range. But since we are now finally running in long mode, it is within range of the jump instructions, so use one to finally start executing the actual kernel code.

4.3 Traps and interrupts

The structure of the low-level trap and interrupt code is similar to that of NetBSD/i386, although no code could be shared. The x86-64 also uses an Interrupt Descriptor Table (IDT) to set up the vectors for traps and interrupts. The work done at the entry points for the traps was the normal save registers/dispatch/restore registers. Although the higher-level trap code can probably be shared between NetBSD/i386 and NetBSD/x86-64, since the set of traps is the same between the architectures, this has not yet been done.

Another set of traps is the system call entry points. The x86-64 supports the same mechanisms that were already present in the IA32 architecture: entering the kernel via a software interrupt, or doing so by issuing a call to a special type of structure (a *call gate*), which automatically switches to the kernel environment. These instructions perform some actions that are usually not needed in an environment where the address space is flat (i.e. the full 4G of virtual memory is available to programs in one chunk, with the kernel usually occupying the upper region). The SYSCALL and SYSRET instructions are optimized for this case, and can be used to implement a faster system call path, which can be an important factor in the performance of applications. The code to handle this was written, but not yet integrated;

currently the old-style entry points are still used, but this will change in the near future.

4.4 Low-level VM / MMU handling

The x86-64 MMU uses a page table structure that is very similar to the IA32. It basically is the IA32 with a Physical Address Extensions page table, extrapolated to have 4 levels, to deal with the 48 bits of virtual memory that the initial family of x86-64 processors will have. Because of this similarity, the i386 pmap module was taken, and abstracted to implement a generic N-level IA32 page table, with either 32- or 64-bit wide entries. The resulting code has been tested on both the x86-64 and i386 ports of NetBSD, with success. The differences between the IA32 and x86-64 code were hidden in C preprocessor macros and type definitions. The advantage of this approach is that it is now easy to optionally support PAE on the NetBSD/i386 as well, because this only means conditionally compiling in a few other macros and type definitions.

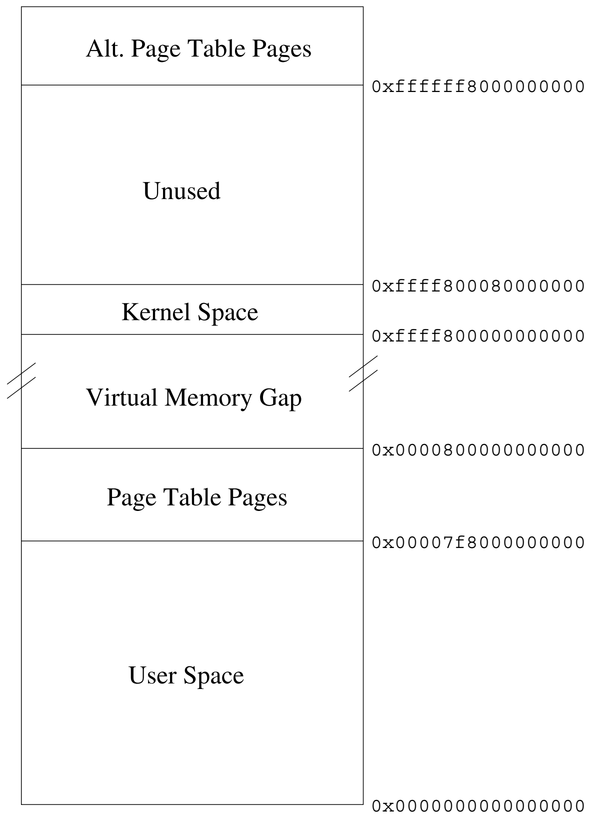

Figure 4: NetBSD/x86-64 virtual memory layout.

The implemented virtual memory layout is shown in figure 4. Because of the sign-extension that the CPU performs on virtual addresses, an unaddressable gap exists between $2^{47}$and $2^{64} - 2^{47}$. This is not unusual; such a gap is also found on e.g. SPARCv9 and Alpha processors. The memory layout is more or less a stretched-out version of the IA32 memory map, as was to be expected in a merged pmap module. A user process runs in the bottom half of virtual memory, while the kernel is always mapped in the top half. Part of the top half is unused, because the kernel doesn't need the huge amount of virtual memory available there, and it turned out that using such an amount of space blew up some data structures in a disproportional way. The upper part of the bottom half of virtual memory is taken up by recursively mapped page table pages, as is the the upper part of the top half of virtual memory (used if the page tables of a process other than the current one need to be changed).

This layout meant that the kernel was out of range of the "kernel" code model, specified in the ABI. The "large" model was needed, but not yet supported by gcc. Fortunately, it turned out that it did work, with a few small modifications. The NetBSD/x86_64 kernel will likely be changed to use the kernel ABI model, once real hardware is available, and a speed assessment will be made. This layout is being used for now because it is a consistent extension of the IA32 model, making pmap code sharing easier.

4.5 Bus-layer backend code

Much of the bus-layer backend code could be reused from the i386 port. For the PIO case, the instructions remained the same, so no changes were needed, except for some modifications to make it fit the extended 64-bit register set. The same goes for memory-mapped I/O. The DMA framework needed to deal with possibly having 32-bit PCI, which would be unable to do DMA access into memory above the 4G limit. For now, a simple solution was picked of having DMA memory for 32-bit PCI always come from below the 4G limit. This needs to be revisited later; for machines with more than 4G of memory conditions may occur where RAM is available, but not below the 4G limit. To avoid this problem, *bounce buffers* can be introduced; they are temporary buffers on which the DMA is done, and to or from which the data must be copied to its actual location.

4.6 Port-specific devices

So far, the NetBSD x86-64 port does not deal with any platform-specific devices. The simulator simulates a number of hardware components that is known from the PC world (like the host-PCI bridge, etc). These components "just worked", and no modifications were needed, after the *bus_space* and *bus_dma* layers were implemented.

4.7 Libraries

The main work in userspace was porting the libraries and C startup code. The C startup code and C library were fairly trivial to port. Most of the work that had to be done was to write the system call stubs and the optimized string functions, keeping in mind that the x86-64 ABI passes most arguments in registers, instead of always on the stack as the i386 ABI specifies. The math library was a bit more work. It could share a lot of code with the i386 (really i387) code, since the FPU has the same instructions, but there was an ABI difference. The x86-64 ABI specifies that floating point arguments are passed in SSE registers, but the i386 ABI passes these on the stack. A few macros had to be written to extract the arguments to the various (mostly trigonometry) functions, prepare them, and then use a common bit of code for both the i386 and x86-64 ports. Lastly, the dynamic linker had to be adapted to deal with the types of relocation that x86-64 shared libraries may use.

4.8 Compatibility code

The x86-64 offers the option to run 32-bit i386 applications without modification. This is a useful option, as it enabled operating systems to run older applications out of the box. Some support for this is needed in the kernel, though. The basic item that is needed to run a 32-bit application is to install 32-bit compatibility memory segments in the various descriptor tables of the CPU. The CPU will execute instructions from such a segment in a 32-bit environment. However, traps to the kernel will switch the CPU to 64-bit mode. This has the advantage that there doesn't need to be any special kernel entry/exit code for 32-bit applications. 32-bit programs do have a different interface to the kernel; they pass arguments to system calls in different ways, and in 32-bit quantities. Also, structures passed to the kernels (or rather, the pointers to them) will have a different alignment. These issues needed to be addressed to enable running old NetBSD/i386 binaries.

Compatibility code to run binaries from various platforms (Linux, Tru64, Solaris, etc.) has been a part of NetBSD for a long time. So, not surprisingly, this issue had already been tackled once before, when NetBSD/sparc64 needed to deal with running 32-bit SPARC binaries. This code, the *compat_netbsd32* module, implements a small layer which translates (if needed) 32-bit arguments to system calls to their 64-bit counterparts.

5 Conclusions and future work

The port of NetBSD to AMD's x86-64 architecture was done in six weeks, which confirms NetBSD's reputation as being a very portable operating system. One week was spent setting up the cross-toolchain and reading the x86-64 specifications, three weeks were spent writing the kernel code, one week was spent writing the userspace code, and one week testing and debugging it all. No problems were observed in any of the machine-independent parts of the kernel during test runs; all (simulated) device drivers, file systems, etc, worked without modification.

The porting effort went smoothly. Table 1 shows the amount of new code written. In the area of sharing

Area	Assembly Lines	C lines
libc	310	2772
C startup	0	104
libm	52	0
kernel	3314	17392
dynamic linker	59	172

Table 1: New lines of C/assembly code per area of the NetBSD source tree

code between "related" CPUs (such as the x86-64 and the IA32), some more work can be done. Currently, the x86-64 pmap isn't shared between the 2 ports, though it is known to work for both architectures. The pmap code is counted as "new code" in table 1, but most of its 3500 lines of code was in some form or another based on the i386 code. Some descriptor table code can also be shared. The number of new C code lines will drop well below 10,000 when the code is properly shared.

Acknowledgments

This work was paid for by my employer, Wasabi Systems, Inc. I'd also like to thank AMD for their support, Virtutech for the simulator, and the folks at SuSe for doing the Linux toolchain work.

References

[1] Jason Thorpe: A Machine-Independent DMA Framework for NetBSD, Usenix 1998 Annual technical conference.

[2] Advanced Micro Devices, Inc: The AMD x86-64 Architecture Programmers Overview, http://www.amd.com/products/cpg/64bit/pdf/x86-64_overview.pdf

[3] Hubicka, Jaeger, Mitchell: x86-64 draft ABI, http://x86-64.org/abi.pdf

[4] Intel Corporation: Pentium 4 manuals, http://developer.intel.com/design/Pentium4/manuals/

[5] Chris Demetriou: NetBSD bus_space(9) manual page, originally in NetBSD 1.3, 1997.