A NIX Terminal

Erik Quanstrom
quanstro@quanstro.net

ABSTRACT

Starting just after the initial announcement of NIX in September, 2011[1], a project was started to replace the Plan 9 32-bit port completely with a 64-bit port based on NIX. Replacing cpu servers was relatively easy; most of the work was already done. Replacing terminals was quite a bit more involved. There was more chore than innovation. But it required enhancements to physical address mapping. Using the VESA interface required a way to execute or emulate BIOS calls. In addition, many new devices were added to support terminal hardware. A few subtle but significant bugs were encountered. The result is a Plan 9 terminal kernel that runs on real hardware and under VMWare Fusion.

Introduction

In September, 2011, NIX was announced on the 9 fans mailing list. By this time, many of the accomodations for ancient PC hardware such as 8259 interrupts, the 8254 timer, were weighing quite a bit on the 32-bit Intel kernel. The kernel also suffered from type confusion; ulong was used for virtual addresses, physical addresses, 32-bit registers and marshaled values. Padding appropriate for 32-bit machines was assumed. A large number of ancient devices were being dragged around. There were a total of 99,524 lines of code in the pc directory. Unfortunately, some of this old code is useful for embedded or old systems, so simply discarding it seemed unwise.

The approach taken was to try to duplicate all currently-used functionality in the 64-bit kernel. The number of 32-bit systems is fading rapidly. It seems feasible to run on only 64-bit hardware most of the time, so using 64-bits as an excuse to drop old code seemed like a viable strategy.

The announced NIX kernel while suitable as a cpu kernel, was missing a number of required features for terminals. Real mode emulation was missing, as were the proper sort of virtual memory mappings. The VGA and draw infrastructure were missing, as were support for usb and for terminal keyboards and mice. These have all been addressed, and this paper was composed on two native NIX terminals. Additionally, some work was put into the networking subsystem to allow booting of terminals under VMWare Fusion.

Address Map and Virtual Mapping

Most video hardware handled by plan 9 works by mapping a physical frame buffer into virtual address space. The frame buffer is mapped after the machine has been otherwise fully initialized, and all cores have been started. Draw operations read and write
from this frame buffer. To speed drawing, the caching subsystem is often instructed to
relax coherency or turning write-combining on for this buffer via (deprecated) MTRR
registers or page-table attributes (PAT).

At the start of this project, none of these requirements were met by the NIX kernel. Vi-

tual memory mapping via *vmap* could only be done on the boot processor early in
startup, and there was no way to modify the the cacheability of a memory range from
the default. The reason for both limitations was similar. There was no facility for
remembering the mapping. So the architectural requirement that all mappings of the
same physical range have the same cacheability could not be met. Likewise, faults on a
*vmap'd* range could not be resolved on other processors. *Vmap* works without tracking
early in boot because the boot processor's (BSP) page tables are copied for each addi-
tional processor (AP).

To track memory cachability, it was decided to create new type of physical memory map
called *adr(9nix)*. The map is low duty-cycle for entry manipulations, but high duty-
cycle for lookups so a simple insertion sorting scheme was used. Map entries consist of
a base, length address type (e.g. memory, mmio, etc.), current use and caching flags.
The memory type is a superset of ACPI memory types. In-use entries remain in the
table. A subset of the programming interface is detailed here.

```c
#include "adr.h"
enum {
  Anone,
  Amemory,
  Areserved,
  Aacpireclaim,
  Aacpinvs,
  Aunusable,
  Adisable,
  Aapic,
  Apcibar,
  Ammio,
  Alast   = Ammio,
};
enum {
  Mfree,
  Mktext,
  Mkpage,
  Mupage,
  Mmmap,
  Mlast   = Mmap,
};
void    adrmapinit(uintmem base, uintmem len, int type, int use)
void    adrfree(uintmem base, uintmem len)
uintmem adrmemtype(uintmem pa, uintmem *len, int *type, int *use)
```

The map is initialized from the *e820* configuration variable. The kernel assumes that
this is populated by the bootloader, typically by calls to BIOS INT15 function E820. After
the kernel starts, ACPI, PCI and other memory-mapped facilities such as I/O APICs and
LAPICs add to this table. Address ranges may be allocated by address type; *adr* splits
address ranges as necessary. Allocated maps are indicated by a non- *Mfree* value of
When maps are freed, they are merged with adjacent maps of the same address type.
Sub-page allocations are allowed to deal with devices that map less than one page of
physical memory.
Adrmapinit enters a new physical mapping which is required to be disjoint with all other mappings. Adralloc allocates a map entry by subdividing an existing entry and changing its use and memory type. Only free entries may be allocated, thus preventing mapping of non-existent memory ranges, double maps, or inconsistent memory types. Adrfree returns the physical range to free status. Cachability of an unmapped range of memory may be changed on allocation since it is unused. Adrmentype returns the page table memory type flags, use and base address for a given virtual address. The flags are stored in a form suitable for setting PAT flags for a 4KB page. The mmu code converts to large-page flags when mapping large pages.

Vmap also requires that we be able to generate a consistent virtual address for a given physical address. Since we have a full 64–bits of address space, we can simply map each physical address to a virtual address that is offset by a suitable constant, KSEG2. Now on initial vmap, adralloc can allocate the given range with a formulaic virtual address on the local processor. (A new function vmappat can specify PAT caching flags; it is used for mapping frame buffers.) Faults on other processors can be resolved by calling adrmentype to find a matching bit of allocated memory and return its memory flags on a page fault.

This scheme works well. Adr is only 513 lines of code (less than 20% larger than its predecessor). There is still only one table of physical addresses. It is never necessary to store the virtual address since we chose the virtual address by formula. Lookups on page faults are much faster than the page fault itself. The benefit of forcing all entries to be entered before mapping, however, is up for debate. On the one hand it does prevent errors, and has caught inconsistencies between MP and ACPI tables in describing APICs. But on the other hand, it is tedious and error prone. It remains because small errors in memory mapping can be hard to track down.

**VESAT BIOS Calls.**

To have a terminal, at least one hardware interface needs to be selected. “VESAT” was selected since most hardware will implement the VBE (VESAT BIOS Extensions) interface[2]. VBE uses 16–bit real mode calls to set up the frame buffer and provide other services like screen blanking. The 386 kernel uses 176 lines of assembly to return to real mode, make the VBE call, and return to 32–bit protected mode. A register–based interface allows calls in from user space. A special file, /dev/realmode provides access to bits of low memory required to run VBE calls.

To make direct BIOS calls from a 64–bit kernel would require all that code, in addition to code to make the 64–to–32 bit and the 32–to–64 bit transitions. Emulation seemed simpler. Fortunately, realemu(8) already provides emulation for VBE BIOS calls via the same register–based interface[3]. So no transition to real mode was required. A expanded version of /dev/realmode, /dev/resmem provides access to low, and ACPI–reserved memory.

Since the kernel itself needs to blank the screen, it uses the context of the process setting up video to maintain a channel to the BIOS emulator. These calls are initiated from the clock interrupt, so it is not possible to make this call directly. The solution taken was to use a kernel process reading a queue to call out to the VBE emulator. The cost is that the emulator needs to remain running.

Unfortunately, the real mode calling interface does not work in 64–bit mode. It assumes the same structure padding as the 32–bit Intel compiler, and little–ending encoding. A proper solution to this problem has not yet been implemented, since it would require a
rwork of the 32-bit kernel, the emulator, and vga(8) as well. The problem has simply been worked around with #pragma pack. It is worth noting that few calls are made through this interface, so a textual interface would be sufficient. Alternatively, 16-bit machine code could be passed in directly for emulation.

Porting Devdraw

There were two main challenges in porting the draw device and supporting libraries to 64-bits: the the dependence on the pool(2) allocator, which is not used by the 64-bit kernel; and two relatively obscure but important bugs.

The draw libraries assume the pool memory allocator. This allocator provides for optional memory compaction, to prevent a relatively small image memory from becoming too fragmented to allow for the allocation of large images. Removing this dependence required isolating the memory allocation in the memdraw library to a single file, which still depends on the pool library. Then a replacement for the kernel was written using the kernel standard allocator. It simply ignores compaction requests. Originally, a compacting allocator was planned, as has been the tradition since the blit[4] but it turns out that this optimization has not been necessary. No memory allocation failures have yet been observed, and memory use seems reasonable.

Unlike the 386, x86-64 cpus have enough registers to sensibly consider using the extern register construct again. This storage class is special. It does not have any relationship to extern nor to register. It means that one register should be allocated per processor for storing the given value. (That is, there will be one independent value per cpu.) The kernel stores a pointer to the local virtual machine Mach *m and the current process Proc *up using this storage class. The compiler allocates these from “the top,” or R15, down, while regular register allocation starts with AX and works up. As the compiler manual notes[5], one must make arrangements that all code including libraries be compiled with the same external registers. For previous compilers using this technique, 28 or more usable registers were available and failure to observe this rule was harmless, since the kernel contains no code that has so many live registers. However, the drawing libraries can use all registers up through R15, which clobbered the external registers. The implemented solution was to prevent the compiler from allocating regular registers higher than R13, since this is more practical than recompiling all libraries for the kernel.

The second bug was with the following code in byteaddr.

```c
uchar *a;

a = i->data->bdata+i->zero+sizeof(ulong)*p.y*i->width;
```

When p.y was small and negative, and a was a normal kernel address (greater than -256MB), a took on values just under 4GB. On careful examination, it was seen that this is a consequence of the Plan 9 compilers being unsigned preserving. Suppose p.y is -1 and the i->width is 3. Then the third term will be 0xffffffff. Since sizeof(ulong) is itself a ulong, the term is of type ulong. If we add this value to a 32-bit pointer, sign doesn’t matter, since unsigned and signed addition are the same. But when adding this value to a 64-bit pointer, we zero-extend and end up with a large positive value. The solution to this bug is trivial, and obvious once found. Simply cast the term to an int to enable sign-extension. It is likely that there are other, similar bugs.
Additional Devices
To equal the functionality of the 32-bit kernels a few additional devices were required: support for more network cards, and usb. To a large extent, this was an exercise in pipe fitting, and replacing ulong with a more descriptive type, usually u32int. However USB devices require 32-bit buffers, and network drivers (especially for 10gbe) can consume more than the maximum 256MB of standard kernel heap. Since the kernel heap has physical addresses less than 4GB (typical physical addresses start at 1MB), the fact that USB has a 32-bit interface is not currently a problem.

The issues with networking are a little more interesting. Due to limitations of the instruction set, it is difficult to run code out of a virtual address that is not either a sign-extended 32-bit value, or simply below 4GB. Since we traditionally place the kernel at the top of memory, and traditionally place the heap above the stack, this limits kernel memory to 256MB. However, there is no requirement about data, so it would be possible to place the real kernel heap below the kernel text. The current kernel takes a hybrid approach. The kernel heap remains where it is, but network Block*s are allocated directly from physical memory. This means they are mapped below the kernel text at KSEG2. While this is somewhat less than ideal—the rest of kernel memory is quite constrained—network buffers may grow to fill most of memory. But it does show the way to having an arbitrary amount of kernel heap. Due to 32-bit devices like USB, however, exceeding 4GB of kernel heap may require careful memory tracking. And 4GB may be a practical limit for the heap.

Conclusion
Currently the 64-bit NIX kernel runs on a variety of AMD, Intel and emulated hardware as a terminal using VESA graphics through VBE calls. This kernel is fast, is able to use all installable memory, and up to 255 cores, and provides broadly equal hardware support to the 32-bit kernels. It is a complete and full-featured replacement for the 32-bit kernels, for hardware supporting 64-bits.

However, there are several areas worth attention in the near future. The scheduler tends to suffer with more than 8 cores. And due to the use of 2MB pages for user segments, user processes use too much memory. Using a virtual page size of 64KB, and abutting segments with increasing page sizes are under investigation.

Abbreviated References