Mac OS X binary compatibility on NetBSD: challenges and implementation

Emmanuel Dreyfus

September 2004

Abstract

Binary compatibility is the ability to run binaries from foreign Operating Systems (OS) with a minimal performance penalty. It is limited to binaries built for the same processor family.

In this paper, we describe how binary compatibility works in NetBSD and then we concentrate on the challenges we need to overcome in order to execute Mac OS X binaries. Finally we present the current status of the project, together with the roadmap for the future.

It is assumed that the reader is familiar with Unix system programming.

1 Binary compatibility

NetBSD has a long record in binary compatibility with other operating systems. Provided a program was built for the same processor, NetBSD is able to execute it despite the fact that the program was not built for NetBSD but for Linux, FreeBSD, Solaris, IRIX, or many others. In this part, we will have a quick look at how binary compatibility works.

1.1 kernel and user mode

UNIX systems have two distinct mode of operation: user mode, and kernel (or system) mode. In user mode, the operating system executes code provided by users. This code

is run with restricted privileges. It has limited access to the computer's memory, and usually no access at all to the hardware.

When running in kernel mode, the OS is only executing trusted code, which was loaded at boot time. This code is known as the OS kernel. The kernel has full access to the memory and hardware. It is here to provide services to user programs: giving access to the hardware, scheduling processes, and enforcing resource allocation and protection.

The transition from user mode to kernel mode occurs on events called traps. A trap is a hardware or software exception that suspends user process execution, and gives control to kernel code. The kernel will handle the exception, after which it may return to user mode and resume the execution of the user process, or it may destroy it. Example of traps are division by zero, memory faults (accessing any virtual addresses where no physical memory is mapped), timer interrupts (that are used to switch between user processes), or system calls. System calls are software traps called by user processes to request access to resources controlled by the kernel. They can be seen as functions called by the user process executing with kernel privilege.

System calls are used to perform a lot of different tasks, ranging from reading from a file to creating a new process, or setting network communication options. The behavior of each system call is documented in section 2 of the manual. The operation described above are therefore documented in `read(2)`, `fork(2)`, and `setsockopt(2)`.

Each system call has a number, typically ranging from 0 to a few hundred. On many processors, system calls are invoked by loading CPU registers with the system call numbers and parameters and by calling a CPU instruction that cause trap. Here is an example of PowerPC assembly that call the `fork()` system call on NetBSD:

```
li      r0,2        # 2 is the system call number for fork()
                    # r0 is the register holding the system call number
sc                  # sc is the CPU instruction that causes the trap.
```

There is a clean separation between user mode and kernel mode. User processes run on top of the kernel with very little knowledge of what is inside a system call. They just expect a behavior documented by kernel developers in a set of man pages. Most programs do not care about kernel internals and will just work if you change the kernel, as long as the system call behavior is left unchanged. This is how binary compatibility works: by emulating the kernel behavior. The user program runs at full speed and is fooled into thinking it runs on the kernel it was built for, whereas it is really running on the NetBSD kernel.

1.2 System call tables

As we explained earlier, when a trap is encountered, control is transfered to the kernel. The kernel calls a function known as the trap handler to take care of the exception.

When the trap is a system call, a particular trap handler – the system call handler – is invoked. The system call handler looks in a table for the function implementing the system call – this is the system call table. The system call number is used as an index in the system call table.

Here is an excerpt of the file that defines the system call table in NetBSD. This file, which is named `syscalls.master`, is not written in C language. A shell script uses `syscalls.master` as input to produce several files in C language.

```
1    STD    { void sys_exit(int rval); }
2    STD    { int sys_fork(void); }
3    STD    { ssize_t sys_read(int fd, void *buf, size_t nbyte); }
4    STD    { ssize_t sys_write(int fd, const void *buf, size_t nbyte); }
5    STD    { int sys_open(const char *path, int flags, ... mode_t mode); }
```

The first idea behind binary compatibility is to have multiple system call tables: the native system call table for native processes, the Linux system call table for Linux processes, and so on.

As emulated OSes usually provide equivalent functionality as NetBSD does, the system call tables used to emulate them tend to mostly contain wrappers that call native kernel functions after doing some argument translation. When the native code succeeds the reverse argument translation takes place, or if the system call fails, the appropriate error code is returned. In some rare situations, an emulated system call has no native counterpart and it must be completely re-implemented.

Of course, the kernel must have a clear idea of what system call table must be used for a given process. It does that by keeping track of the emulated OS for each process. At process launch time, in the `execve()` system call, the OS emulation is discovered and the appropriate system call table is selected. This system call table will then be used for any system call the process will issue.

There are a few other problems that must be taking into account, including the transitions from kernel to the user process: when the process is first launched, or when it catches a signal, the kernel must setup the stack and registers in the same way the emulated kernel would have do. This is implemented in NetBSD by having emulations hooks for that operations, so that a given binary compatibility layer can perform its particular setup.

Another major problem is dynamic linking. If an emulated program is dynamically linked, it will open shared libraries, which must be built for the same OS. The program usually expects these foreign libraries to be in the same place and with the same names as NetBSD native libraries.

The solution to that problem is to have a shadow root directory for foreign processes, where files are first looked up before looking at the real root. For instance, when a Linux binary attempts to open `/usr/lib/libncurses.so`, the NetBSD kernel will first

attempt to open `/emul/linux/usr/lib/libncurses.so` and if that fails, it will try `/usr/lib/libncurses.so`.

The shadow root directory is also extremely useful to store configuration files for foreign binaries, as the file names often clash with their native counterparts.

2 Mac OS X binary compatibility challenges

Binary compatibility is an old feature in NetBSD, and the kernel now contains enough compatibility code to easily emulate another Unix flavor. But when we came to Mac OS X binary compatibility, we hit several new challenges that had never been encountered while working on binary compatibility layers in NetBSD.

Here is a list of the most challenging problems:

- Mac OS X uses an executable format called Mach-O. NetBSD knew nothing about Mach-O. It was only able to run ELF, ECOFF, and a.out binaries.
- Mac OS X is an hybrid system, based on a Mach kernel and a BSD kernel. It features a dual system call table: positive system call numbers refer to the BSD system call table, and negative system call numbers refer to the Mach system call table. NetBSD never had to handle such an odd setup.

  While Mac OS X's kernel BSD interface is very close to NetBSD kernel interface, and therefore can be very easily emulated, the Mach interface has just nothing to do with a Unix system. This is a complete set of system calls with no NetBSD equivalents that must be completely re-implemented.
- NetBSD provides user programs with hardware device access through traditional Unix device files, in the `/dev` directory. Mac OS X provides such a feature for a limited subset of devices: mostly storage units and terminals. Other devices, such as video, keyboard and mouse, are made available to user programs through the IOKit, an extended device driver interface specific to Mac OS X.
- And last but not least, NetBSD uses the X Window System for the graphic user interface, whereas Mac OS X uses Quartz, a PDF based display system. The two systems are similar but incompatible: they are both based on a client/server scheme, which a display server handling the video output and graphic user interface applications behaving as client. The protocol used between the clients and the server is different.

3 Running Mach-O binaries

Binary compatibility layers are usually developed in an incremental way. The developer attempts to run a simple binary built for the target OS, and of course that fails. The failure is caused by a feature of the target OS kernel that the NetBSD kernel does not emulate properly. The developer fixes that by implementing the missing feature emulation, retry running the binary, finds another failure, and things repeat until the emulation is good enough to transparently run the foreign OS binary.

When trying to implement a feature emulation, a few sources of informations are useful: the man pages and other system documentation, and the system include files. For some OSes, such as Linux or FreeBSD, a possible source of information is the kernel sources, but contrarily to a popular belief, having access to the target OS source code does not help very much. What we are interested in is the target system behavior, not its implementation. It tends to be much more productive to write unit test programs, to run them on the target OS, and to see what they do, rather than reading the target OS kernel sources to understand what it should do.

Usually, the work is done on system calls emulations. But when trying to run a Mac OS X binary, the first show stopper was not occurring on a system call. The NetBSD kernel was unable to actually load and launch the foreign binary. The reason for that failure was that Mac OS X uses an executable format called Mach-O, which was not known to NetBSD.

NetBSD knew about the legacy a.out and the newer ELF executable formats. Migration from a.out to ELF occurred a long time ago, but NetBSD retained the ability to run a.out binaries, for the sake of backward compatibility. The support for running binaries in two different executable formats on the same kernel helped a lot when introducing Mach-O executable format support. Let us have a closer look on how it was done.

The `execve()` system call is responsible for running a new binary. It uses a `struct execsw` table called the exec switch to perform its duties. Each entry in the exec switch defines the operations used to load the binary for a given OS emulation and executable format. For example that switch contains entries for NetBSD ELF binaries, NetBSD a.out binaries and Linux ELF binaries, among many others. The first part of the job was to add an entry for Mac OS X Mach-O binaries.

This entry defines a few operations for loading Mach-O binaries, which had to be implemented:

First, the probe function. Each entry in the exec switch defines a probe function whose task is to tell if the entry is able to run a given binary or not. The probe function looks at the executable header to decide if it is a binary it can handle. In `execve()`,

the kernel first walks the exec switch, using the probe functions to discover which entry should be used to execute the binary.

Once the kernel knows which entry in the exec switch should be used, it uses another function – the `makecmds()` function – to load the executable. This function is responsible for setting up the process virtual memory space, loading the text and data sections from the executable, and setting up stack space.

Things are more complicated when implementing the `makecmds()` command for Mach-O binaries than for ELF binaries. Mach-O binaries can be fat, that is, they can contain text segments for different architectures. Of course that needed to be taken into account so that the right text section would be loaded.

Another difference is in object loading. When loading an ELF executable, the duty of the kernel is just to load the executable, and possibly a dynamic linker. With Mach-O binaries, the kernel also has to load any dynamic library needed by the executable. The kernel duty stops there, as it is not required to load the libraries used by the libraries used by the program: the dynamic linker will do that from userland.

Finally the kernel needs to setup the stack and populate it with arguments and other information the userland startup code expects. There are some Mac OS X oddities here: On many systems, the stack of a newly created program starts by the argument count and a pointer to the argument vector (the famous `argc` and `argv` arguments to the `main()` function of a program written in C). Mac OS X processes' stack starts by a pointer to a copy of the Mach-O executable header, the argument count and the pointer to the argument vector.

In NetBSD kernel source, all the code for running Mach-O binaries was written by Christos Zoulas. It can be found in `src/sys/kern/exec_macho.c`. The exec switch is defined in `src/sys/sys/exec.h` and `src/sys/kern/exec_conf.c`. The code used to setup the stack of Mac OS X processes is available in `src/sys/compat/darwin/darwin_exec.c`.

4 Mach system calls

As we explained earlier, the Mac OS X kernel is an hybrid system, featuring two sets of system calls. Apple used the following scheme: positive system call numbers are used for the BSD interface, whereas negative system call numbers are used for the Mach interface. Mac OS X user processes mix system calls to both parts of the kernel: BSD and Mach. The BSD part features a well known Unix kernel interface, while the Mach part's interface has just nothing common with Unix.

Mach is a microkernel, implementing virtually anything as processes called servers.

some servers run in user mode, other run in kernel mode. The kernel only provides two services: process scheduling and Inter-Process Communication (IPC). IPC is extensively used in a Mach-based system, because a process that need a system resource will send a request to a server process instead of sending it to the kernel as it does on a Unix system.

Most of the Mach kernel interface is therefore devoted to Mach IPCs. The Mach microkernel implements a message passing system, which uses objects called messages, ports, and rights.

A Mach message is a packet of data with a 24 bytes header and a payload that can carry any information.

A Mach port has nothing to do with TCP or UDP ports. It is a message queue maintained in the kernel. A single process reads from it whereas multiple processes may write to it. Each message is sent with a destination port and a source port, so that the server can answer the request to the right process.

A Mach right determines a process access right on a port, such as a send right or a receive right. The right is a kernel resource that the process acquires, uses and releases, just like files in Unix. For the process, a right is handled through a 32 bits integer, which is usually called a port name. You can think of port names as Unix file descriptors. Rights can be obtained through some Mach system calls, or they can be carried by messages. For instance, when a process receives a message from another process, the message normally carries a send right to the source port so that the receiver can reply to the message.

The Mach system call table can be found in `src/sys/compat/mach/syscalls.master` in NetBSD kernel sources. The most frequently used system call is `msg_trap()`, which is used to send and receive Mach messages. This system call was quite complicated to implement since it has to handle both sending and receiving, asynchronous reception, timeouts, and other features. Moreover, it has to juggle with port and right lists. `msg_trap()` is implemented in `src/sys/compat/mach/mach_message.c`.

In order to make debugging easier, the `ktrace` command on NetBSD was modified to record Mach system calls, a feature which is not available in Mac OS X's `ktrace`. NetBSD `ktrace` is even able to dump Mach messages. Here is an excerpt of the kernel trace for the Mac OS X's `ls` command running on NetBSD. It gives a good insight on the way Mach IPC works.

```
   541 ls       CALL  host_self_trap
   541 ls       RET   host_self_trap 35454977/0x21d0001
(...)
   541 ls       CALL  reply_port
   541 ls       RET   reply_port 35454979/0x21d0003
   541 ls       CALL  msg_trap(0xbffedd70,3,0x18,0x30,0x21d0003,0,0)
   541 ls       MMSG  host_page_size [202]
        000     00001513   00000000   021d0001   021d0003  ................
```

```
        010     00000000   000000ca                          ........
   541 ls       MMSG  host_page_size reply [302]
        000     00001200   00000028   00000000   021d0003  .......(........
        010     00000000   0000012e   00000000   00000000  ................
        020     00000000   00001000   00000000   00000008  ................
   541 ls       RET   msg_trap 0
```

`host_self_trap()` gives a send right to the host port, which is used to request host-specific configuration. `reply_port()` allocates a receive right to a new port. Then `msg_trap()` is used to send a message to the host port and get a reply.

The Mach message header is defined in `src/sys/compat/mach/mach_message.h`. It contains 6 words of 32 bits:

```
typedef struct {
        mach_msg_bits_t msgh_bits;         /* flags */
        mach_msg_size_t msgh_size;         /* message size */
        mach_port_t     msgh_remote_port;  /* destination port */
        mach_port_t     msgh_local_port;   /* source port */
        mach_msg_size_t msgh_reserved;     /* unused */
        mach_msg_id_t   msgh_id;           /* Message Id */
} mach_msg_header_t;
```

The message Id is used to characterize the message meaning and the payload type. Here, a message Id of 202 sent to the host port requests the machine memory page size. The message payload is void. The server listening behind the host port responds by a message with a 24 bytes payload. That message ends with a 32 bits word containing the requested value (here 0x1000, or 4096), followed by a 64 bits message trailer.

It is interesting to note that using the Unix kernel interface, the same operation can be done by doing a single call to `sysctl()`, requesting the `hw.pagesize` variable.

5 Mach kernel servers

The Mac OS X kernel implements many Mach servers inside the kernel. They can be reached through three ports: the host port, the task port and the thread port. The host port is used to request configuration about the machine the caller is running on, whereas the task and thread ports, are used to request system resources on behalf of the calling task (a task is a Unix process in Mach terminology) or thread.

Because binary compatibility takes place at the kernel boundary, NetBSD had to implement all the Mach kernel servers. Instead of implementing different kernel threads servicing the requests, the NetBSD kernel services each request in the process context of the caller. The message Id is used to lookup the function that will get the request message and produce the reply.

The NetBSD implementation of `msg_trap()` uses a table defined in `src/sys/compat/mach/mach_services.master` to select the appropriate function. Like the `syscalls.`

master file, this file is not written in C, and a shell script is used to produce various C files from it. Here is an excerpt from that file:

```
200     STD     host_info
201     UNIMPL  host_kernel_version
202     STD     host_page_size
203     UNIMPL  memory_object_memory_entry
204     UNIMPL  host_processor_info
205     STD     host_get_io_master
206     STD     host_get_clock_service
207     UNIMPL  kmod_get_info
208     UNIMPL  host_zone_info
209     UNIMPL  host_virtual_physical_table_info
210     UNIMPL  host_ipc_hash_info
```

Here we find the information that a Mach message with message Id 202 must be handled by the host_page_size() function. This function is defined in src/sys/compat/mach/mach_host.c. Here is its complete implementation:

```
int
mach_host_page_size(args)
        struct mach_trap_args *args;
{
        mach_host_page_size_request_t *req = args->smsg;
        mach_host_page_size_reply_t *rep = args->rmsg;
        size_t *msglen = args->rsize;

        *msglen = sizeof(*rep);
        mach_set_header(rep, req, *msglen);

        rep->rep_page_size = PAGE_SIZE;

        mach_set_trailer(rep, *msglen);

        return 0;
}
```

mach_host_page_size_request_t and mach_host_page_size_reply_t are the request and reply Mach messages, defined in src/sys/compat/mach/mach_port.h:

```
typedef struct {
        mach_msg_header_t req_msgh;
} mach_host_page_size_request_t;

typedef struct {
        mach_msg_header_t rep_msgh;
        mach_ndr_record_t rep_ndr;
        mach_kern_return_t rep_retval;
        mach_vm_size_t rep_page_size;
        mach_msg_trailer_t rep_trailer;
} mach_host_page_size_reply_t;
```

host_page_size() call mach_set_header() and mach_set_trailer() to fill the Mach header and trailer for the reply packet, and it sets the requested value: the page size.

There are many other Mach kernel services, most of them being unused by Mac OS X binaries and therefore left unimplemented in NetBSD. The most used services deal with port, task, thread, and memory management. For example a task can spawn a

new thread or request a memory mapping by sending a Mach message to its own task port.

And of course there are services implemented as user-level daemons. We do not have to do anything special for them: the Mac OS X binary can be executed on NetBSD on the top of the Mac OS X binary compatibility layer, and they will provide the adequate service to other Mac OS X processes running on NetBSD.

6 Mach IPC bootstrap

The Mach IPC is at the core of Mac OS X, it is used intensively everywhere. As a result, Mac OS X processes have a recurrent problem: how to obtain a send right on the port of a given server?

For kernel-level servers, Mac OS X processes use the host, task and thread ports, obtained by the `host_self_trap()`, `mach_task_self()`, and `mach_thread_self()` system calls.

For user-level servers, a bootstrap mechanism is needed. The `mach_init` daemon is responsible for providing this service.

`mach_init`, always running with PID 2, is the first user-level process spawn on Mac OS X. The reader used to Unix will probably wince: Usually, `init` is the first process spawn, and it has PID 1. Mac OS X even happens to have an `init` process with PID 1.

In fact, `mach_init` is really the first user process spawn, with PID 1. It then forks, and the father, with PID 1, uses `execve()` to run `init`, while the child, with PID 2, continue executing `mach_init`. This odd move is there to ensure that `init` still gets the PID that a lot of Unix programs expect, while `mach_init` is launched first.

Once it has forked the traditional Unix's `init`, `mach_init` behaves as a directory service. It registers to the kernel as being the bootstrap process, thus making one of its ports available to all processes through another special port any process can access: the bootstrap port.

Each time a server process starts, it uses the bootstrap port to send a register message to `mach_init`, giving the ports on which it is servicing and the service name. And when a client process needs a send right to a server port, it uses the bootstrap port to send a message containing the service name. `mach_init` will reply by a message carrying a send right to the server port.

Implementing support for this was easy. We just had to implement the Mach service used by `mach_init` to register its port: `task_set_special_port`. The NetBSD kernel maintains a global variable called `mach_bootstrap_port`, and once `mach_init` registers,

any process requesting a send right to the bootstrap port will get a right to the registered port. That way things work as expected.

But there was one small problem: `mach_init` checks its PID, and will only behave as the bootstrap process if it is started with PID 1. On NetBSD, PID 1 is always used by `init`, so it is not possible to book it for `mach_init`. It is not possible either to spawn `mach_init` at system startup instead of `init`.

The solution was finally to fool `mach_init` into thinking it has the PID 1 whereas it is not the case. `mach_init` uses the BSD system call `getpid()` to obtain its PID. We just had to recognize that `mach_init` was started and have `getpid()` answering 1 instead of the real PID.

But the kernel has no way of recognizing `mach_init`. We therefore use the help of the system administrator, which tells the kernel that it runs `mach_init` using a `sysctl` variable.

This is done with the following shell command:

```
sysctl -w emul.darwin.init.pid=$$ && exec /emul/darwin/sbin/mach_init
```

Using `sysctl`, the system administrator informs the kernel that a Mac OS X process running with this PID (remember that `$$` is the shell's PID) should be fooled into thinking that it's PID is 1. Then we use `exec` to run `mach_init` without forking a new process, thus retaining the same PID.

That way, `mach_init` thinks it is the first process spawned, with PID 1, and it behaves as the Mach bootstrap process.

Of course, Mac OS X being an open source OS, it would have been possible to patch `mach_init` sources so that it does not check its PID and always behave as the Mach bootstrap process, but the goal of binary compatibility is to run unmodified binaries from the foreign OS, therefore the `sysctl` hack choice.

7 Handling binaries built for Mac OS X.3

Unix processes tend to use a a few library functions such as `memcpy()` or `bzero` very often. In a dynamically linked executable, calling a library function means walking various tables, which is time consuming. Starting with Mac OS X.3, Apple introduced a nifty optimization: the kernel maps a few pages of code containing various utility functions at the end of each processes' address space. Theses pages are called the comm pages.

The functions can be reached at an absolute memory address that is carved into the stone. Calling such a function is blazingly fast because it just involve branching to a

well known address, there is no more performance loss caused by dynamic linking.

Another advantage of this approach is that the kernel can map optimized versions of the functions that make use of some particular optional hardware feature, such as Altivec on the PowerPC G4. The user process does not have to deal with checking the hardware ability and/or the kernel version, nor does it have to include multiples versions of some function to match various optional optimization.

For the binary compatibility layer developer, this optimization caused surprising failures. As most of Mac OS X.2 command line programs worked on NetBSD, any binary built for Mac OS X.3 quickly died with a segmentation fault. After some investigation, it became obvious that something weird was taking place: the segmentation fault was caused by the program jumping at a fixed absolute address where nothing was mapped. Testing on Mac OS X with `gdb` did show that a page of memory containing code was mapped at that fixed address. Because it was mapped before the process did any system call, it was obvious that the kernel had to do it.

Fortunately, when running on Mac OS X, `gdb` displayed symbols when dumping the memory in the comm pages, so it was not that difficult to understand the purpose of the code they contained. The last part of the job was to actually implement the functions in the comm pages. It had to be done in assembly, as some functions had to fit in a small slot of memory, a constraint that a C compiler is not able to understand.

The assembly code for the comm pages can be found in `src/sys/arch/powerpc/powerpc/darwin_commpage_machdep.S`. Peter Grehan, Srinivasa Kanduru, and Wolfgang Solfrank helped a lot writing it.

8 Running Aqua application and emulating the IOKit

By implementing the Mach IPC and a collection of kernel services, we have been able to run most command-line binaries from a Mac OS X system. Programs using the X Window graphic interface are also likely to work, though this has not been tested. But what we are really aiming for is running programs using the native Mac OS X graphical user interface, known as Aqua.

Aqua is based on a display system called Quartz, which is similar but incompatible with the X Window system. Quartz uses a display server and Aqua applications are clients that talk to the display server. In Mac OS X.2, the display server is called `WindowServer`. In Mac OS X.3, it was replaced by `QuartzDisplay`.

The main problem with running Aqua applications is to have a Quartz display server so that they can actually display something. We have several ways of obtaining a Quartz Display server running on NetBSD.

- Write a Quartz Display server from scratch. That solution is clearly not the way to go, since it means re-implementing the code for managing various video boards.
- Write a Quartz to X11 bridge, in order to reuse X11 video hardware support. The problem with that solution is that we need to discover how Quartz clients talk with the Quartz server in order to implement the bridge. Moreover, it is not certain that this bridge is posible to implement, because Quartz seems to have many more features compared to X Window.
- Use Mac OS X's Quartz display server and run it on the top of our binary compatibility layer. We chose that path, with the idea that it will be easier to reverse engineer the Quartz client/server interface and work on a Quartz to X11 bridge once we will have the Quartz display server running on the top of NetBSD.

We therefore tried to run `WindowServer`, and later `QuartzDisplay`, on NetBSD. The big problem we encountered was to provide an emulation for the IOKit, which is the device interface used by the Quartz display server to access the video board, the keyboard and the mouse.

Traditional Unix device interface is rather simple, not to say poor. Devices are available through special files in the `/dev` directory, which can be opened for reading or writing. Any operation that cannot be implemented through a read or write is implemented through the `ioctl()` system call, a general purpose I/O function used for virtually anything.

Mac OS X uses the traditional Unix device interface for disks and terminals, but most of the hardware is not available through that interface. Video boards, keyboards and mice are only available through a big object oriented framework known as the IOKit. The IOKit provides mechanisms based on Mach IPC for discovering and accessing hardware. It defines device classes, and device drivers are objects within the classes.

The display server uses two device classes: `IOFramebuffer`, to access the video, and `IOHIDSystem`, for the input systems (keyboard and mouse). In order to run the display server, we needed to implement an `IOFramebuffer` driver and an `IOHIDSystem` driver, plus enough Mach services from the IOKit interface so that things just work.

The IOKit interface is quite complex and not exciting enough to be covered here. The two drivers were more tricky.

Drivers in the `IOFramebuffer` class implement access to a framebuffer. There are a few Mach services used to read and write configuration information about the framebuffer, such as pixel depth, color palette, gamma table, screen size, and so on.

`IOFramebuffer` drivers must also make two memory mappings available to a user

process that would request them:

- The framebuffer itself.
- A page of memory shared between kernel and userland where cursor related configuration is stored. A user process can use that area to read the cursor position or to modify the cursor visibility, for instance.

The biggest problem was to provide access to a framebuffer while the NetBSD kernel does not know about all the various video boards that may be present in a machine. Fortunately the Power Macintosh boot environment, known as Open Firmware, provides a framebuffer to NetBSD. It is slow and not configurable, but it is available.

The `IOFramebuffer` driver in the Mac OS X compatibility layer maps the framebuffer from the wscons console driver. On a Power Macintosh, we know that this framebuffer will be at least the Open Firmware framebuffer. If the NetBSD kernel supports accessing the framebuffer of the video board in a more efficient way (for instance, NetBSD kernel's `machfb` device is able to use the ATI Mach64 framebuffer), then the `IOFramebuffer` driver will automatically use it.

As we do not support hardware accelerated cursors yet, the shared memory page used for the cursor configuration is not emulated beyond a simple mapping of zero-filled memory.

All the code for the `IOFramebuffer` driver is located in `src/sys/compat/darwin/darwin_ioframebuffer.c`. The code that implements the IOKit interface is located in `src/sys/compat/mach/mach_iokit.c`

The `IOHIDSystem` driver was the most difficult part. Like `IOFramebuffer`, it works by mapping a shared page of memory between the kernel and the user program. The kernel will write keyboard and mouse events to a queue located in that page. The display server will read them from the queue. This mechanism saves a lot of system calls for reading user input.

In our implementation, when the display server maps the page of shared memory, the kernel spawns a new kernel thread that opens the wscons console driver. This kernel thread, called `iohidsystem`, reads wscons input events, converts them to `IOHIDSystem` events, and writes them to the queue. That way, the keyboard and mouse events are made available to the display server, without the need to hack some hooks in the NetBSD native input drivers for keyboard and mouse.

The code for the `IOHIDSystem` driver can be found in `src/sys/compat/darwin/darwin_iohidsystem.c`

Working with the Quartz display server was not easy, as the IOKit interface is really complex and the user program is not open source. Fortunately, XFree86 provides a X

server called XDarwin, which can use the IOKit. Working with XDarwin did help a lot, since it was possible to poke debug `printf()` in it to understand how things were going on.

We are now able to run a fully functional XDarwin on NetBSD/macppc. This means that the IOKit emulation, the `IOFramebuffer` and `IOHIDSystem` drivers are good enough to be actually used. Unfortunately, as of today, the Quartz display server won't work yet. Debugging the problems that prevent it from working is the next item on the project TODO list.

Conclusion

Mac OS X binary compatibility in NetBSD grew quite large. It now features more than 20.000 lines of C and assembly code. For now support has only been written for NetBSD PowerPC ports. NetBSD could also run Darwin/i386 binaries, provided the machine dependant part is ported (it accounts for 5% of the code).

As of today, NetBSD-current is able to run most command line tools from Mac OS X. Mac OS X Programs using the X11 graphical user interface such as Matlab or Open Office should work too, though nobody explored that area yet. It is difficult to give an idea of when NetBSD will be able to run Aqua applications, because we do not really know the issues we are going to encounter and solve. Moreover, some event could shorten the delay: if some Quartz display server become available for NetBSD (for instance as an open source Mac OS X remote desktop project), it would remove the major problem.

The biggest and hardest part of the work so far was to implement the Mach IPC and various Mach services related to task, threads, ports, memory, and many others resources. The IOKit was one of the most complex part of the picture.

On the performance front, a comparison of Mac OS X and emulated Mac OS X on NetBSD would be interesting. Binary compatibility does not cause major performance loss. If the native implementation of a feature is much more efficient than the target OS implementation, then the emulation can even be faster than the original for that feature. Such a comparison will probably be the subject of a future paper.

Acknowledgements

Thanks to Christos Zoulas for reviewing this paper.