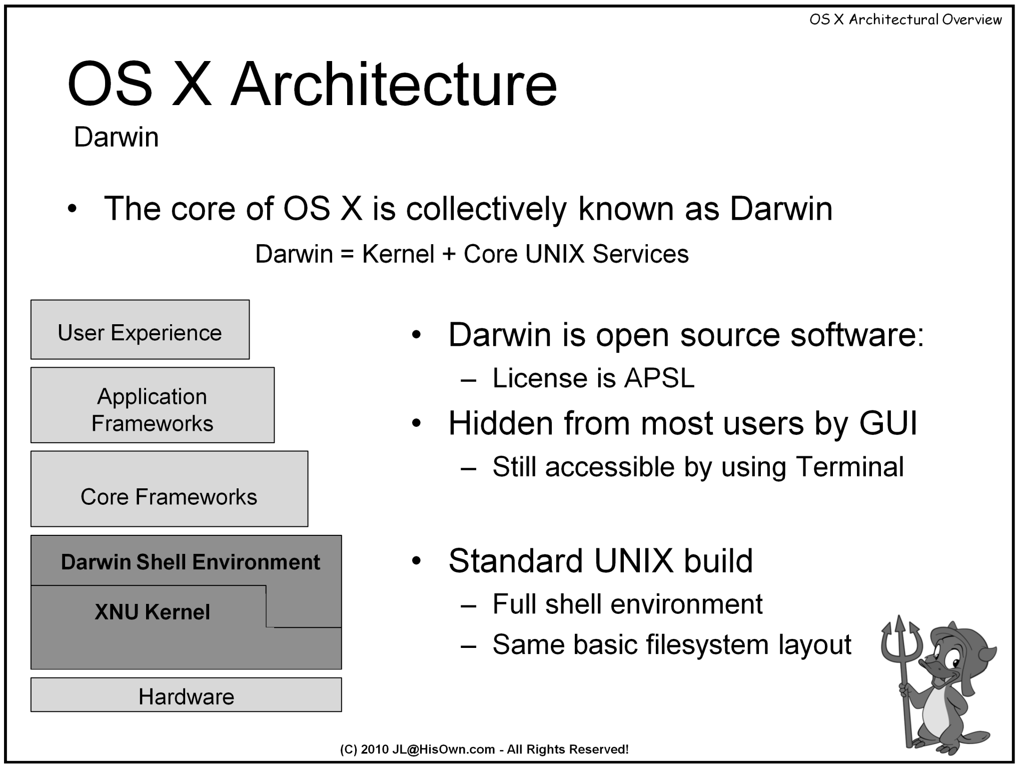
Table of Contents

Jonathan Levin specializes in training and consulting services. This, and many other training materials, are created and constantly updated to reflect the ever changing environment of the IT industry.

To report errata, provide feedback, or for more details, feel free to email JL@HisOwn.com

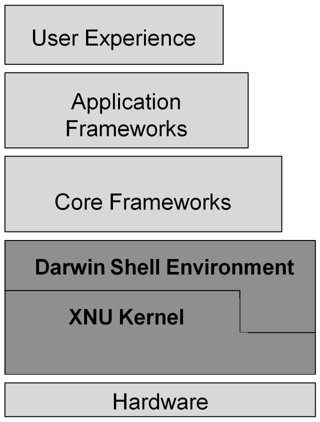


OS X Architectural Overview

OS X Architecture
Darwin

- The core of OS X is collectively known as Darwin

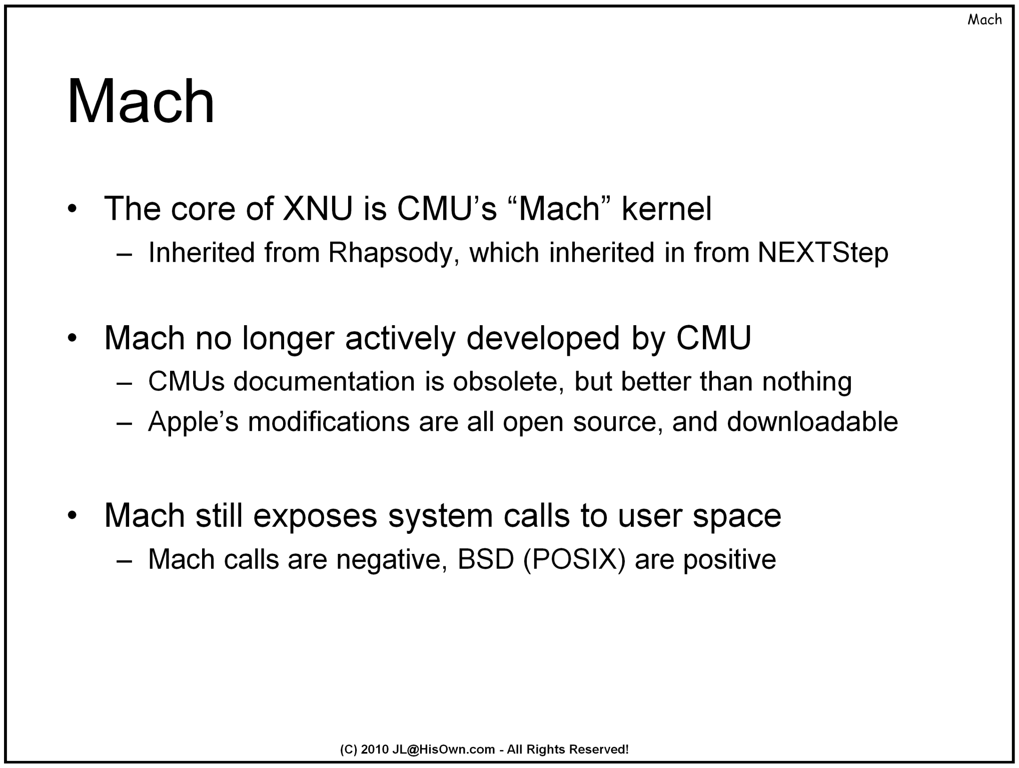
Darwin = Kernel + Core UNIX Services



- Darwin is open source software:
  - License is APSL
- Hidden from most users by GUI
  - Still accessible by using Terminal
- Standard UNIX build
  - Full shell environment
  - Same basic filesystem layout

Darwin is the collective name for the core of OS X – comprised of the Kernel (XNU), and the shell environment. The core implements a full UNIX build, both in terms of APIs as well as the various binaries (df, ls, mv, rm…), libraries (libc, libpthread…) and UNIX filesystem (/bin, /lib, /sbin, /etc..). Most users don't even see those aspects, as even the filesystem is hidden (it can't be normally seen in OS X's Finder).

The Darwin environment is full open source – Apple makes valiant efforts to keep its sources updated with every release of OS X, at http://www.opensource.apple.com/. The sources are released under the "Apple Public Source License", which is a BSD-like license – not surprising, considering Darwin ripped major portions of the FreeBSD kernel.



Mach

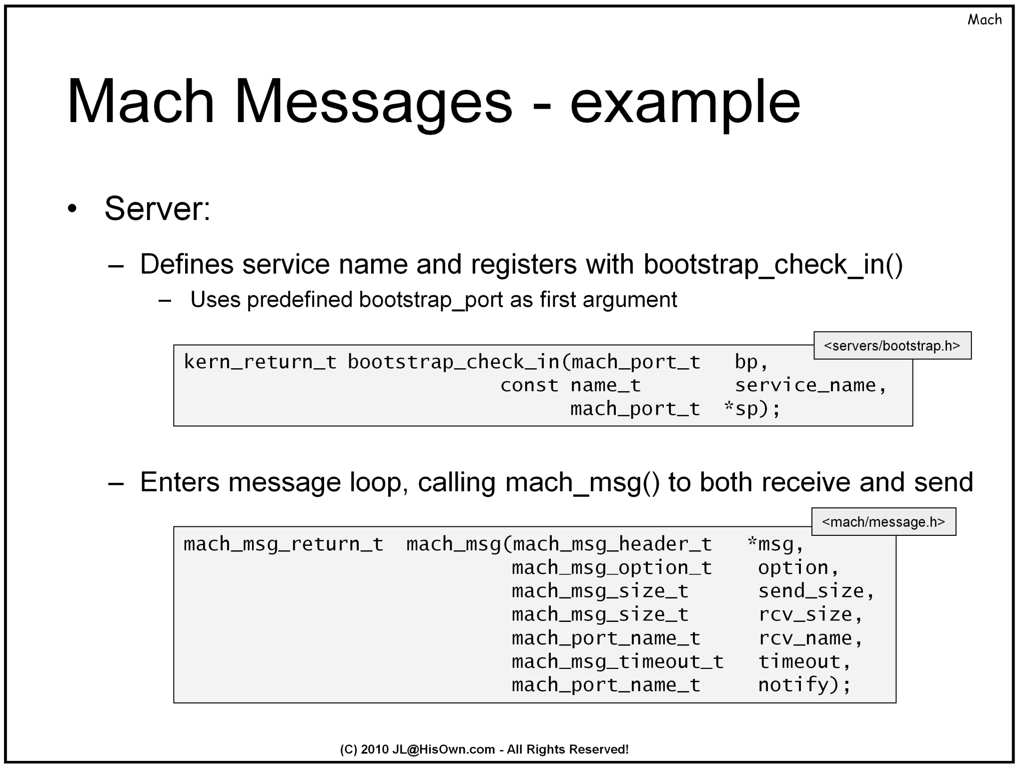
Mach

- The core of XNU is CMU's "Mach" kernel
  - Inherited from Rhapsody, which inherited in from NEXTStep
- Mach no longer actively developed by CMU
  - CMUs documentation is obsolete, but better than nothing
  - Apple's modifications are all open source, and downloadable
- Mach still exposes system calls to user space
  - Mach calls are negative, BSD (POSIX) are positive

OS X has a rather long and convoluted evolution from Rhapsody, which in turn evolved from NEXTStep, brought together to Apple with Steve Jobs' return. As a result, its kernel, XNU, is a hybrid of Mach (the core of NEXTStep) and BSD/POSIX (which is more common in other UNIX environments).

Mach itself is a microkernel architecture, originally developed by Carnegie Mellon University (CMU). CMU hasn't really publicly developed anything in Mach recently, and Apple took over, and installed their own changes – first and foremost the integration with the BSD top layer, and the I/O Kit environment (both of which will be discussed, in due time). However, Apple really didn't bother much with cumbersome documentation – and, as such is the case, most of the documentation on the Mach subsystem is the very obsolete Mach 3.0 documentation – last updated around 1992! At least the source of XNU remains freely open and downloadable (http://www.opensource.apple.com), so one can learn a lot from reading it (which is how this very book was created).

Since OS X must supply both personalities – Mach and BSD/POSIX – concurrently, the BSD system calls occupy a positive range (as on all other systems). The Mach calls are all negative numbers.



Mach

Mach Messages - example

- Server:
  - Defines service name and registers with bootstrap_check_in()
    - Uses predefined bootstrap_port as first argument

<servers/bootstrap.h>

```
kern_return_t bootstrap_check_in(mach_port_t   bp,
                           const name_t        service_name,
                                 mach_port_t  *sp);
```

  - Enters message loop, calling mach_msg() to both receive and send

<mach/message.h>

```
mach_msg_return_t  mach_msg(mach_msg_header_t   *msg,
                            mach_msg_option_t    option,
                            mach_msg_size_t      send_size,
                            mach_msg_size_t      rcv_size,
                            mach_port_name_t     rcv_name,
                            mach_msg_timeout_t   timeout,
                            mach_port_name_t     notify);
```

To consider a simple example of a Mach IPC server, let’s construct one, step by step:

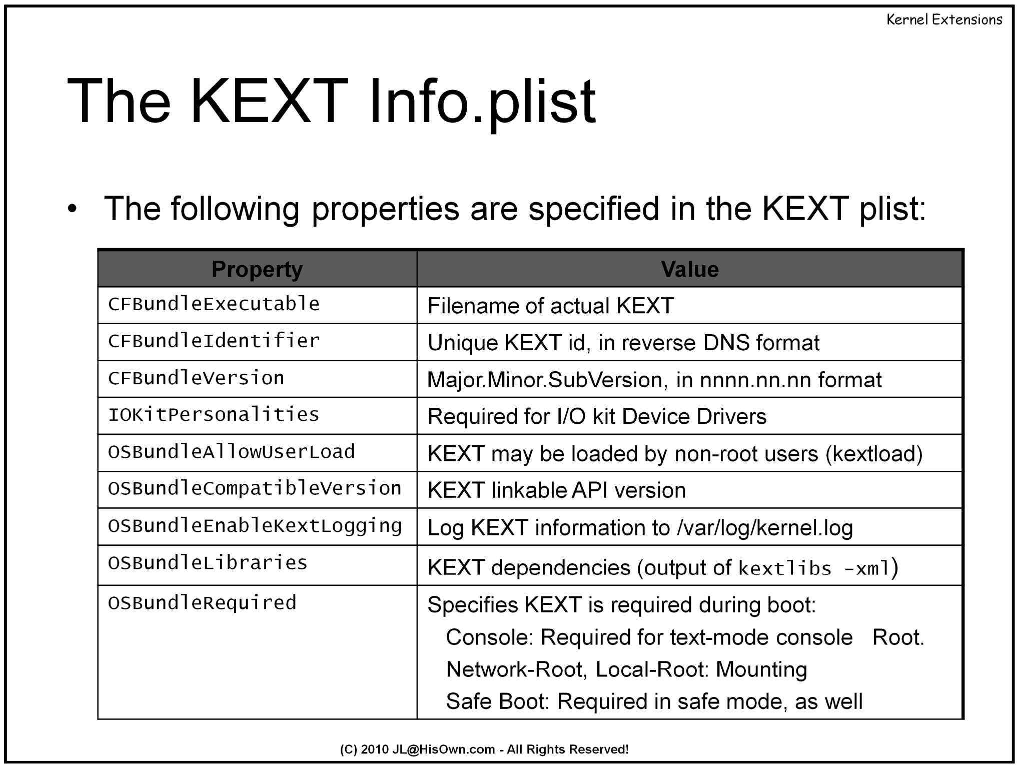
The server can either request a specific port, or choose to register with a bootstrap server. This is the preferred method, because the same bootstrap server can then be used to query the port in question (in a sense, this is similar to the Sun RPC port mapper).

The bootstrap server has a preexisting port, **bootstrap_port**, which is defined as an extern in **<servers/bootstrap.h>**. All the server needs to do is #define a service name, in reverse DNS format (e.g. “com.hisown.sampleService”), and call **bootstrap_check_in**(). It will return success, or an error code (e.g. BOOTSTRAP_NOT_PRIVILEGED or BOOTSTRAP_SERVICE_ACTIVE).

```
#define SERVICE_NAME    “com.hisown.sampleService”
kern_return_t rc = bootstrap_check_in(bootstrap_port,
                                       SERVICE_NAME,
                                      &server_port);

if (rc != BOOTSTRAP_SUCCESS) { … };
```

Once the server has an allocated port, we’re pretty much done. The service publishing is handled for us by the bootstrap server, which means we can enter our message loop, and start processing messages – receiving requests, and sending replies.



The KEXT Info.plist

- The following properties are specified in the KEXT plist:

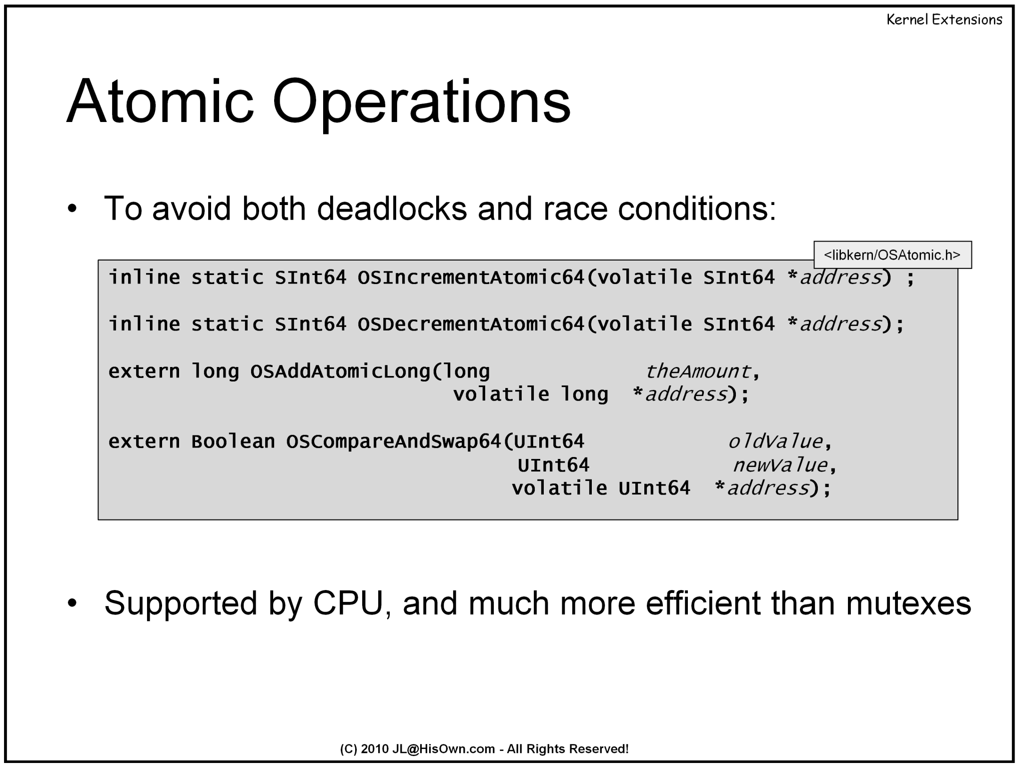
Property	Value
`CFBundleExecutable`	Filename of actual KEXT
`CFBundleIdentifier`	Unique KEXT id, in reverse DNS format
`CFBundleVersion`	Major.Minor.SubVersion, in nnnn.nn.nn format
`IOKitPersonalities`	Required for I/O kit Device Drivers
`OSBundleAllowUserLoad`	KEXT may be loaded by non-root users (kextload)
`OSBundleCompatibleVersion`	KEXT linkable API version
`OSBundleEnableKextLogging`	Log KEXT information to /var/log/kernel.log
`OSBundleLibraries`	KEXT dependencies (output of `kextlibs -xml`)
`OSBundleRequired`	Specifies KEXT is required during boot: Console: Required for text-mode console Root. Network-Root, Local-Root: Mounting Safe Boot: Required in safe mode, as well

The KEXT's Info.plist is a vital component. Without it, the Kernel loader cannot properly determine dependencies and load ordering. This file, usually generated by XCode and later edited for minor modifications by the developer, defines the KEXT's various properties – the most of important of which are shown in the table above. With the exception of **IOKitPersonalities**, which we will discuss when we get to I/O Kit, the fields are detailed and explained above.

Like all plists, the file is an XML dict file. It's comprised of key/string combinations. For example, the consider the following values, from the IOATAFamily driver (found in /System/Library/Extensions):

Property	Value
`CFBundleExecutable`	IOATAFamily
`CFBundleIdentifier`	com.apple.iokit.IOATAFamily
`CFBundleVersion`	2.5.1
`OSBundleCompatibleVersion`	1.0b1
`OSBundleLibraries`	com.apple.kpi.bsd 8.0.0 com.apple.kpi.iokit 8.0.0 com.apple.kpi.libkern 8.0.0 com.apple.kpi.mach 8.0.0
`OSBundleRequired`	Local-Root

.



Atomic Operations

- To avoid both deadlocks and race conditions:

<libkern/OSAtomic.h>

```
inline static SInt64 OSIncrementAtomic64(volatile SInt64 *address) ;

inline static SInt64 OSDecrementAtomic64(volatile SInt64 *address);

extern long OSAddAtomicLong(long            theAmount,
                            volatile long  *address);

extern Boolean OSCompareAndSwap64(UInt64           oldValue,
                                  UInt64           newValue,
                                 volatile UInt64  *address);
```

- Supported by CPU, and much more efficient than mutexes

Atomicity is very important at the Kernel level. The Kernel itself is multithreaded, and multiple threads may attempt to access the same memory area. This could lead to serious data corruption: Consider the case of two threads, trying to increment the same memory area. If thread preemption causes one thread to be switched in mid operation, when it comes back, its stored value (now in a register) is not the same as that in the memory location.

Atomic operations guarantee that will not be the case. OS X defines a set of atomic operations, in <libkern/OSAtomic.h>. These functions are implemented differently on different architectures. The actual implementation for x86 is at /kernel/xnu//libkern/x86_64/OSAtomic.s

```
_OSCompareAndSwap64:
_OSCompareAndSwapPtr: #;oldValue, newValue, ptr
        movq             %rdi, %rax
        lock
        cmpxchgq        %rsi, 0(%rdx)   #; CAS (eax is an implicit operand)
        sete            %al             #; did CAS succeed? (TZ=1)
        movzbq          %al, %rax       #; clear out the high bytes
        ret

_OSAddAtomic64:
_OSAddAtomicLong:
        lock
        xaddq   %rdi, 0(%rsi)           #; Atomic exchange and add
        movq    %rdi, %rax;
        ret
```

While these functions are slower than their normal, non atomic equivalents, (and thus shouldn't always be used, in non threaded situations) they are guaranteed exclusivity, especially in SMP environments, and save the significant overhead of mutexes.

Exercise

To show system call invocation, consider the following simple program:

```
void main(int argc, char **argv)
{
        fork();
        exit(1);
}
```

Compile it and the use “otool” to disassemble. You will need to use “-tV” for the disassembly, and “-I” to show symbols. Like so:

```
Darwin # gcc a.c –o a
Darwin # otool -tV –I a
Indirect symbols for (__TEXT,__symbol_stub1) 2 entries
address            index name
0x0000000100000f30     9 _exit
0x0000000100000f36    10 _fork
  ...
_main:
0000000100000f0c        pushq   %rbp
0000000100000f0d        movq    %rsp,%rbp
0000000100000f10        subq    $0x10,%rsp
0000000100000f14        movl    %edi,0xfc(%rbp)
0000000100000f17        movq    %rsi,0xf0(%rbp)
0000000100000f1b        movl    $0x00000000,%eax
0000000100000f20        callq   0x00000f36
0000000100000f25        movl    $0x00000001,%edi
0000000100000f2a        callq   0x00000f30
```

What do we see here? That “f36” over there is the reference to fork(), and the “f30” – to exit (note that the “1” argument to exit is passed (line 100000f25 – in the EDI register). If you use “nm” on the binary, you’ll see that the _fork and _exit are both undefined:

```
Darwin # nm a
..
0000000100001058 D _environ
                 U _exit
                 U _fork
0000000100000f0c T _main
```

Where, then, will we find them? Using “otool” again, this time with “-L”, shows us the dependency:

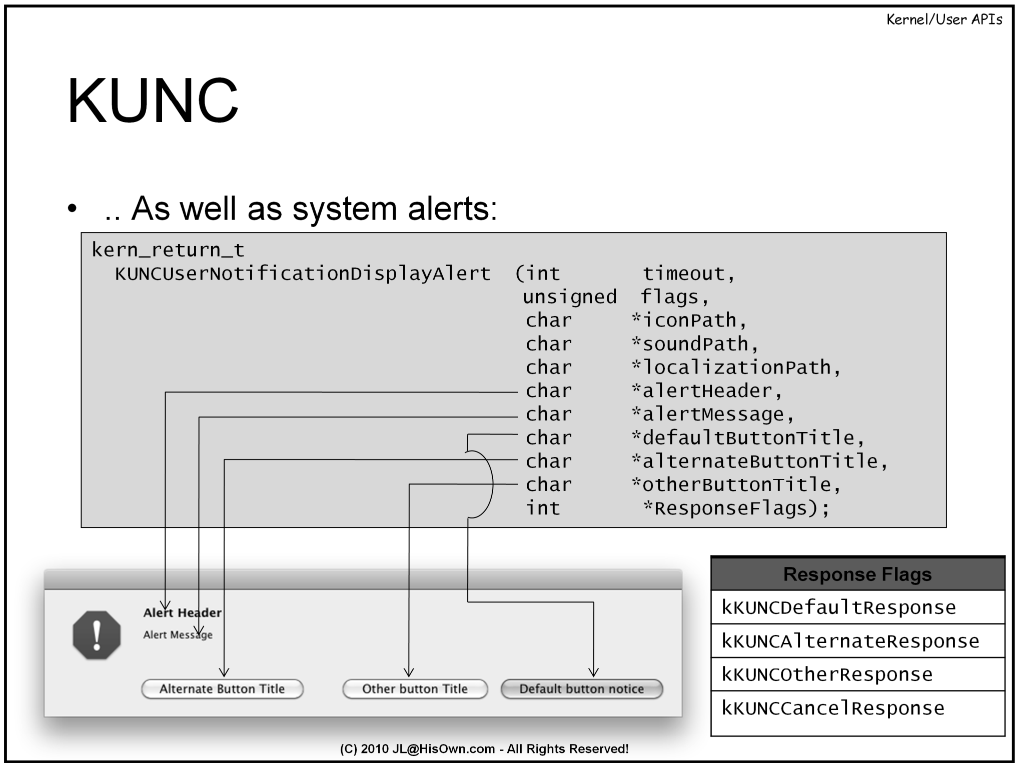
```
Darwin # otool -L a
a:
   /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 125.2.0)
```

Exercise (cont.)

So, onward to libSystem, it is:

```
/usr/lib/libSystem.B.dylib:
(__TEXT,__text) section
_mach_reply_port:
000010c0        movl    $0xffffffe6,%eax    # note negative system call number
000010c5        calll   __sysenter_trap
000010ca        ret
000010cb        nop
_thread_self_trap:
000010cc        movl    $0xffffffe5,%eax    # note negative system call number
000010d1        calll   __sysenter_trap
000010d6        ret
000010d7        nop
..
___mmap:
0000400c        movl    $0x000000c5,%eax    # note positive system call number (197)
00004011        calll   __sysenter_trap
..
__sysenter_trap:
000013d8        popl    %edx
000013d9        movl    %esp,%ecx
000013db        sysenter           # this is where the magic happens
000013dd        nopl    (%eax)
```

So, you see EAX holds the system call number, which is passed to the Kernel, and *all* system calls go through the same choke point – sysenter. Mach system calls are negative, BSD/POSIX calls – positive.



KUNC

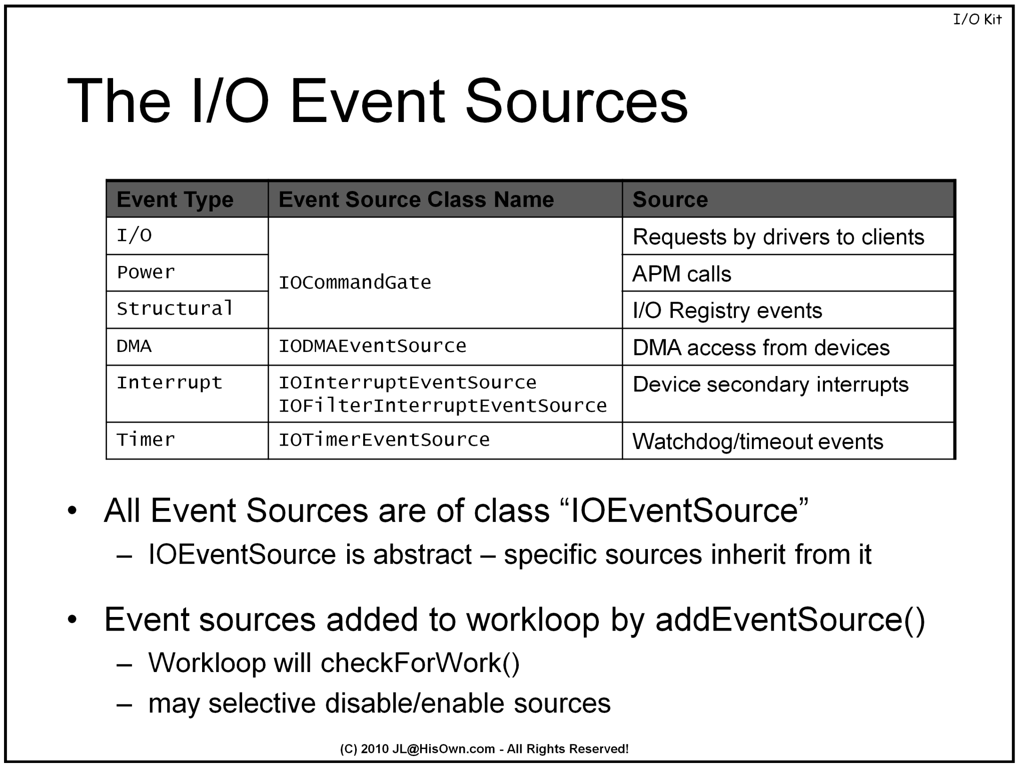
- .. As well as system alerts:

```
kern_return_t
  KUNCUserNotificationDisplayAlert  (int       timeout,
                                    unsigned  flags,
                                     char     *iconPath,
                                     char     *soundPath,
                                     char     *localizationPath,
                                     char     *alertHeader,
                                     char     *alertMessage,
                                     char     *defaultButtonTitle,
                                     char     *alternateButtonTitle,
                                     char     *otherButtonTitle,
                                     int       *ResponseFlags);
```

Response Flags
kKUNCDefaultResponse
kKUNCAlternateResponse
kKUNCOtherResponse
kKUNCCancelResponse

Even better, KUNC can be used to display alerts. Unlike notices, alerts have up to three buttons, enabling the user to select one, and essentially return a choice to the kernel code. The **KUNCUserNotificationDisplayAlert()** adds three more arguments: AlternateButtonTitle, OtherButtonTitle, and ResponseFlags. The two titles are of the corresponding buttons, and "ResponseFlags" is an out parameter, which will contain the choice of the button pressed:

Response Flags	
kKUNCDefaultResponse	User didn't press any button (timeout)
kKUNCAlternateResponse	User pressed the alternate button
kKUNCOtherResponse	User pressed the "other" (3rd) button
kKUNCCancelResponse	User pressed cancel



I/O Kit

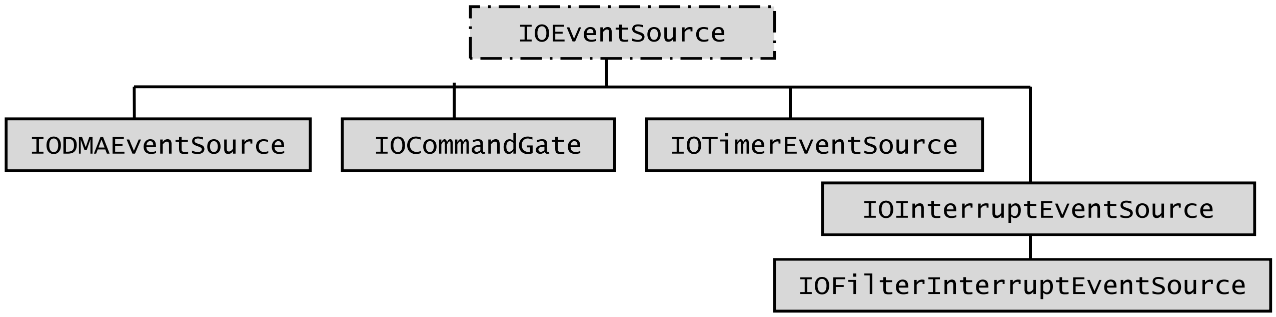
The I/O Event Sources

Event Type	Event Source Class Name	Source
I/O	IOCommandGate	Requests by drivers to clients
Power		APM calls
Structural		I/O Registry events
DMA	IODMAEventSource	DMA access from devices
Interrupt	IOInterruptEventSource IOFilterInterruptEventSource	Device secondary interrupts
Timer	IOTimerEventSource	Watchdog/timeout events

- All Event Sources are of class "IOEventSource"
  - IOEventSource is abstract – specific sources inherit from it
- Event sources added to workloop by addEventSource()
  - Workloop will checkForWork()
  - may selective disable/enable sources

The I/O Workloop handles one or more "event sources". There are currently six types of events, handled by four types of sources, as shown above.

All the various event sources are derived classes of the **IOEventSource** abstract class:



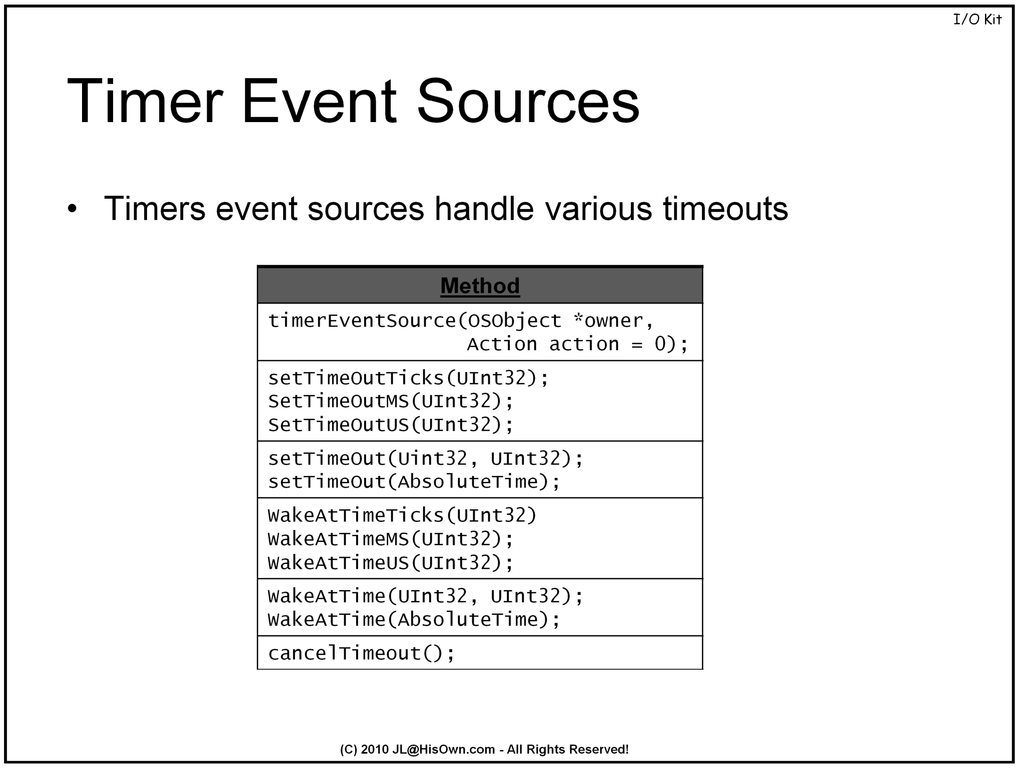
IOEventSource is a very simple class, which exports, among others, the following:

Method	Used for
disable()/enable()/isEnabled()	Disabling/Enabling source
setAction(IOEventSource::Action)	Action to execute on event

Specific subclasses may export more functions, but these are the most important ones. setAction() is used to install a callback for the event. IOEventSource::Action is a typedef of a function pointer:

```
typedef  void ( *Action)( OSObject *owner, ...);
```

Where "Owner" is usually the driver object itself (i.e. "this").



Timer Event Sources

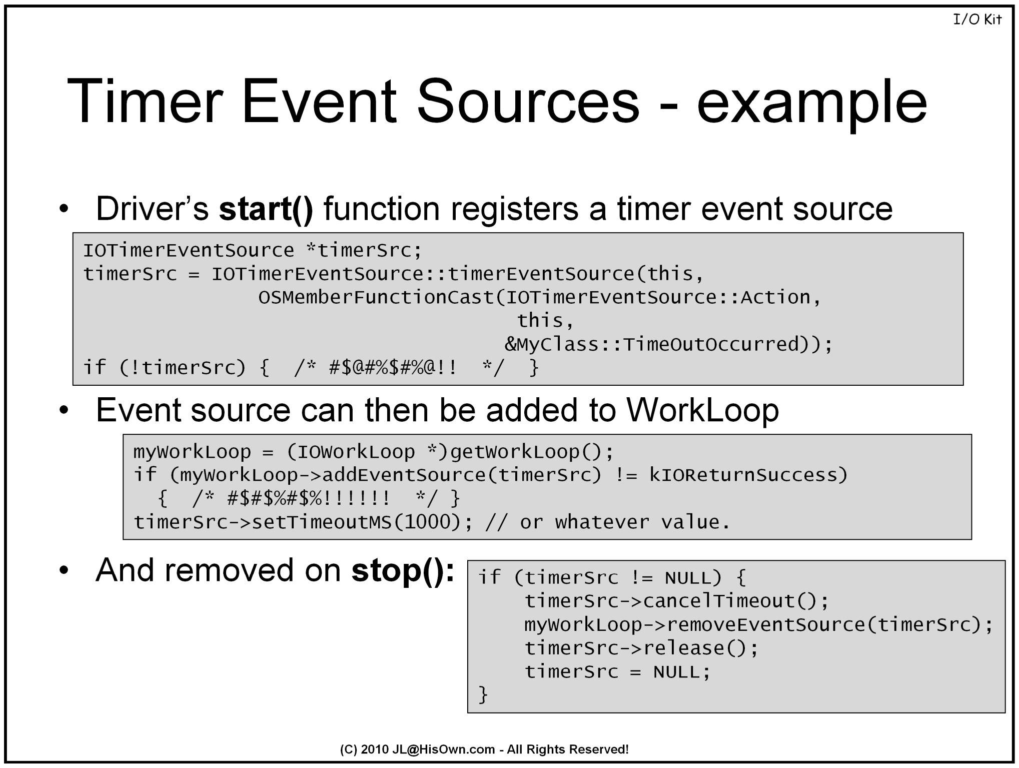
- Timers event sources handle various timeouts

Method
`timerEventSource(OSObject *owner, Action action = 0);`
`setTimeOutTicks(UInt32);` `SetTimeOutMS(UInt32);` `SetTimeOutUS(UInt32);`
`setTimeOut(Uint32, UInt32);` `setTimeOut(AbsoluteTime);`
`WakeAtTimeTicks(UInt32)` `WakeAtTimeMS(UInt32);` `WakeAtTimeUS(UInt32);`
`WakeAtTime(UInt32, UInt32);` `WakeAtTime(AbsoluteTime);`
`cancelTimeout();`

Timers are useful event sources for implementing all sorts of timeouts in your driver. This comes in handy when some device requests are asynchronous in nature, and your device has to respond within a set time period. Using timers, you can set a device watchdog, and handle cases where the response fails to arrive in a timely manner.

Timers offer several levels of granularity: Milliseconds (MS), Microseconds (US) or timer ticks. The former two are absolute, but in the case of ticks, the timer resolution is dependent on the Kernel value of HZ – usually 1/100th of a second. <IOKit/IOTypes.h> defines scale factors as an enum:

```
enum {
    kNanosecondScale  = 1,
    kMicrosecondScale = 1000,
    kMillisecondScale = 1000 * 1000,
    kSecondScale      = 1000 * 1000 * 1000,
    kTickScale        = (kSecondScale / 100)
};
```



Timer Event Sources - example

- Driver's **start()** function registers a timer event source

```
IOTimerEventSource *timerSrc;
timerSrc = IOTimerEventSource::timerEventSource(this,
               OSMemberFunctionCast(IOTimerEventSource::Action,
                                      this,
                                     &MyClass::TimeOutOccurred));
if (!timerSrc) {  /* #$@#%$#%@!!  */  }
```

- Event source can then be added to WorkLoop

```
myWorkLoop = (IOWorkLoop *)getWorkLoop();
if (myWorkLoop->addEventSource(timerSrc) != kIOReturnSuccess)
  {  /* #$#$%#$%!!!!!!  */ }
timerSrc->setTimeoutMS(1000); // or whatever value.
```

- And removed on **stop():**

```
if (timerSrc != NULL) {
    timerSrc->cancelTimeout();
    myWorkLoop->removeEventSource(timerSrc);
    timerSrc->release();
    timerSrc = NULL;
}
```

Timers can be created, using the **IOTimerEventSource** class. Its constructor takes two parameters: the driver object itself, and the action – which is the handler function, which is an IOEventSource::Action object. Once the event source is created, it's straightforward to add it to the workloop, as shown above.

To implement the call back:

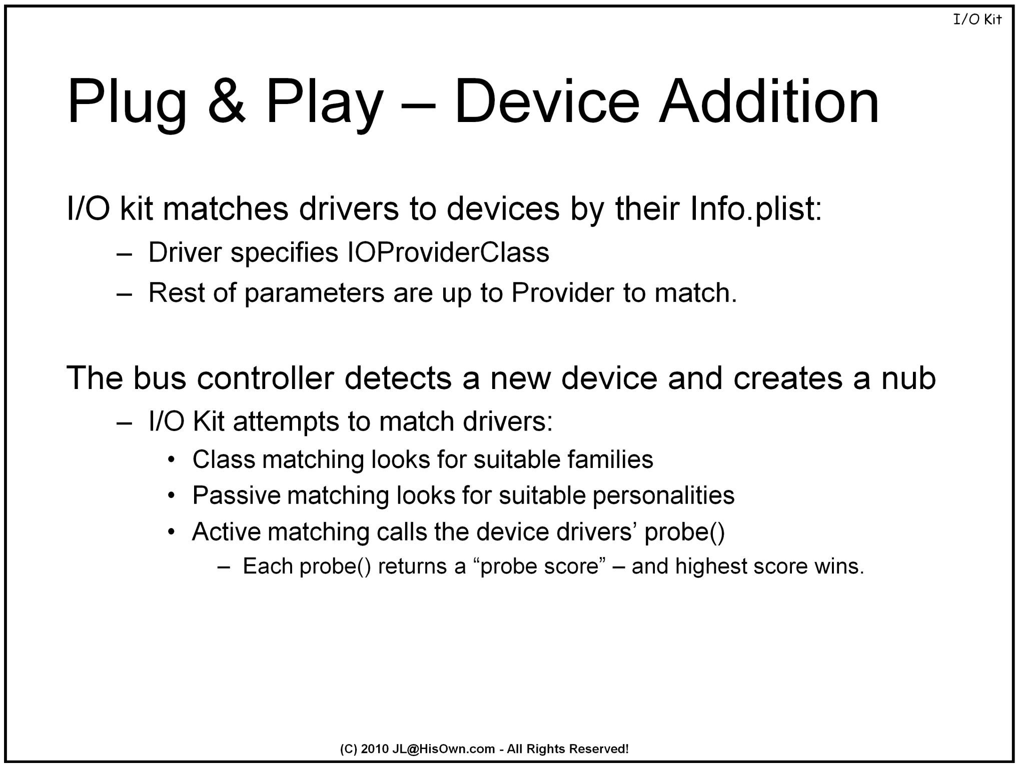
```
void    MyClass::TimeOutOccurred(OSObject *o, IOTimerEventSource *t)
{
    // o is the “this” pointer, and “t” is the TimerEventSource..

        IOLog("Timeout!\n"); // Normally, do something here..

}
```

It's also important to remove the TimerEventSource in the free() function (which is why it should be global in your driver):

```
if (timerSource != NULL) {
            timerSource->cancelTimeout();
            _workLoop->removeEventSource(timerSource);
            timerSource->release();
            timerSource = NULL;
            }
```



Plug & Play – Device Addition

I/O kit matches drivers to devices by their Info.plist:

- Driver specifies IOProviderClass
- Rest of parameters are up to Provider to match.

The bus controller detects a new device and creates a nub

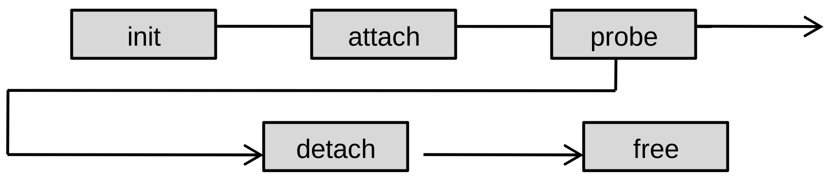
- I/O Kit attempts to match drivers:
  - Class matching looks for suitable families
  - Passive matching looks for suitable personalities
  - Active matching calls the device drivers' probe()
    - Each probe() returns a "probe score" – and highest score wins.

Upon device addition, I/O relies on the bus driver to detect some change in the bus, and to create nubs for the added device. To this nub, will be added one of several possible drivers for the device. The addition process is totally transparent to user mode, but in fact quite lengthy, in three stages:

Stage I: Class Matching – by examining all drivers (in /System/Library/Extensions) I/O kit isolates drivers whose PLists specify the bus as their IOProvider. Families which do not match in any way are ignored.

Stage II: Passive matching – examines the Info.Plists even further, trying to find provider match hints. These are different, according to each provider (for example, PCI uses IOPCIMatch). But if the provider detects a match, we have something to work with.

Stage III: Attempts active matching, by initializing each driver through its first stage of the lifecycle. Recall:



Each driver returns a probe score, and the one with the highest score wins.