Proceedings of the

Ottawa Linux Symposium

July 22–24, 1999

Ottawa, Ontario

Conference Organizers

Program Chair

Andrew J. Hutton, *Achilles Internet Inc.*

Program Committee

Andrew J. Hutton, *Achilles Internet Inc.*
Paul F. Hutton, *Achilles Internet Inc.*
Craig Ross, *Rebel.com*

Proceedings Formatting Team

Brenda Butler, *LaTeX student*
Greg Franks, *Carleton University*
Andrew J. Hutton, *Achilles Internet Inc.*
David Huggins-Daines, *P&L Communications*

Ottawa Linux Symposium
July 22–24, 1999
Ottawa, Ontario
Table of Contents

Papers

Slides

Late Submissions

Ninja Adventures in the Stratosphere or Two Attempts at Server Optimization

Zach A. Brown
zab@redhat.com

Abstract

With the explosive growth of the Internet, the demand for serving information over the network has increased tremendously. Servers are being pushed to their limits trying to meet increased demand. Very careful attention must be paid to the implementation of servers to make the best use of the hardware and operating systems at our disposal.

1 Introduction

This paper discusses two different approaches to writing high-performance network servers. The first approach is demonstrated by hftpd, a fully featured FTP server following a threaded model, and phhttpd, a trimmed down static HTTP cache engine based on non-blocking primitives. Both were done to test interesting approaches to scaling problems at hand.

2 Making FTP Go Fast

hftpd was born due to the lack of a fully featured and scalable FTP server implementation for Linux. The existing servers at the time [freshmeat] aimed to be universally portable and as such restricted themselves to the most basic Unix APIs which tended to be more than a little heavyweight.

2.1 Designing the nozzle

hftpd was intended to place a low burden on a machine so that any hardware could handle as much as it could. This allows the modest machine to add FTP to its array of services without melting the machine, and also allows tremendously huge sites to use fewer machines to serve their archives.

A threaded model was chosen very early on. This was intended to avoid incurring the expense of creating and destroying a process for each connection. Using threads was also hoped to cut down on the amount of additional state needed to maintain each connection.

Previous attempts at threaded FTP servers had met with frustration while dealing with the file system. Most threading APIs don't allow for each thread to have its own working directory, root of the file system, and file system authentication in the form of USER IDs for the thread. This requires the FTP daemon author to write file system access wrappers that handle the managing of the clients' file system accesses in user space. This is the approach Peter Eriksson took with his FTP server [pftpd]. Using Linux's clone() interface directly allows this overhead and added complexity to be avoided. clone() lets a process create threads that can optionally share attributes, like file system context, with their parent. With this flexible threading environment we were able to use the usual Unix file APIs and let the kernel deal with the book keeping associated with file system operations.

When hftpd is started, threads are spawned for each virtual FTP server specified in the server configuration. When these threads accept a new connection, they clone()s off a new thread to handle the new connection. These threads then handle all of the operations for the new client. They assume the identity of the authenticated user and traverse the file system natively. They block sending the data to the client and exit when the connection closes.

Threading cut down per client memory usage substantially. hftpd on an Intel box uses 24k of memory, including 8k for the kernel stack and 4k for the client's thread's stack. This can be compared to the roughly 160k per client that wuftpd, the canonical fork() based server, uses. These numbers were measured after just connecting a client, so it leaves out memory book keeping while the client is doing work, but gives a good impression of the difference.

2.2 Classic Optimizations

Basic IO optimizations were made as the server was coded. Some servers copy data from files into local buffers before transmission. The data path travels from the kernel, into user space as its read, and then back to the kernel as it is written. hftpd mmap()s the file into its address space and passes this address to write() directly. This avoids the overhead of moving the data in and out of userspace.

In addition, hftpd makes use of the sendfile() system call when it is available. It sends the contents of a file out a network socket without exposing the actual data to the calling program. This removes the expensive mmap() system call that was previously required to present an image of the data to write(). mmap() would be especially expensive in our threaded environment. Multiple threads must not be in mmap() at the same time because they share their VM context. Heavy use of mmap() in a threaded environment creates a nice point of locking contention. mmap() must also has to maintain page tables and must flush TLBs in an SMP system to ensure that all the CPUs don't have stale address mapping caches. None of these operations are required with sendfile() because the contents of the file are happily ignored.

ASCII mode transfers are a thorn in the FTP server's side. Line breaks in the local machine's character set must be converted into standard pairs of carriage returns and line feeds. In Unix, this amounts to adding a carriage return when a line feed is found in the data stream. Most servers read through the transmitting data a byte at a time and insert the carriage return as required. hftpd avoids this CPU overhead by mmap()ing the file and write()ing in large chunks around the line feeds that are found by efficient string searching functions.

Linux's TCP_CORK socket option was also used throughout the server to maximize network utilization. When TCP_CORK is enabled on a socket outgoing data is queued until there is enough to fill a maximally sized packet. When TCP_CORK is later disabled on a socket, pending data is immediately written to the network. This allows the application to write() to the socket in whatever manner it prefers and still make efficient use of the network. For example, the ASCII transmit code doesn't have to worry about sending the added carriage returns in their own packets, TCP_CORK guarantees that they will be sent out with other pending data.

2.3 Cache what you can

Optimizations specific to the FTP protocol and our server model were also made. The process of generating responses to file listings is very expensive indeed. Parsing the OS native file attributes into a human-friendly format requires a fair bit of non-contiguous disk access and CPU use to munge the data. Original FTP servers implemented this by executing the system's `ls` program so that they could remove themselves from the picture and remain nice and portable. Unfortunately this method does not perform well. The `ls` process must be fork()ed and destroyed for each operation and the data passes through the server from the task on its way to the client. Most recent FTP servers implement the file listing generation internally which removes a good deal of this overhead.

To solve this problem, hftpd caches the generated output for full directories so it may sent efficiently to the next client that asks for it. hftpd stores cached `ls` output in files. The cache filenames are keyed by a hash of the inode and device of the directory whose listing the file contains, as well as an integer representation of the `ls` options used to create the listing. The cache files are deemed stale if they're older than the directory they contain listings for, and will be refreshed in that case. It is up to the administrator to decide whether detailed listings should be cached as the data they present to the client can become stale without the directory timestamps changing. This optimization reduced the common full directory listing generation into a small number of system calls before a nice sendfile() that efficiently transfers the listing to the client.

2.4 Optimize the rest

Many servers have the ability to return recursive tar files of directories when asked to do so. They typically implement this by calling an external `tar` program, which has the same portability and code reuse advantages and performance disadvantages as using an external `ls` binary. We implement `tar` internal to the server and it uses sendfile() to send out the actual file contents in the tar file.

Implementing `ls` and `tar` internally puts a maintenance burden on the hftpd code as it can't rely on external parties to maintain the file listing and tar generation code. The performance benefits were judged to more than make up for the cost of keeping the relatively stagnant tar format and `ls` presentation code up to date.

The ability to run programs via an FTP session has long been an expected feature of Unix FTP servers, regardless of what a goofy idea it is. Sites typically use this to present enhanced file searching and information facilities to clients. The typical server simply fork()s its process and exec()s the program to be run. This would have been a bad idea in our case because the fork()ing process is threaded and shares the resources of all the currently connected clients. Shared resources, most notably file descriptors, would have been copied for the new process. hftpd would have had to waste its time tearing down these potentially sensitive descriptors before running the new program, wasting the effort the kernel expended in duplicating them in the first place. Instead a dedicated thread is created which fork()s and exec()s programs on behalf of other threads. This fork()ing thread is spawned early during the startup of the server before client state is built up. It doesn't share the VM or file contexts with the other threads. Its only connection back to the threaded monster server is through a Unix domain socket. This also creates a nice security barrier between running programs and the main hftpd process.

When a client wants to execute a binary, and is deemed allowed to do so, it sends a message down this socket to the fork()ing thread that contains control information about the new program. The fork()ing thread then does its duty by fork()ing, setting up the environment as the caller asked, and executes the binary. A quick network optimization was made here as well. Rather than have the client give its data back to the server who then passes it to the client, hftpd simply attaches the client's connected socket to the standard output of the running program. This lets the program write directly to the connected client without the server getting in the way. TCP_CORK is also set on the socket so that programs that may try and spew tiny pieces of data don't end up creating poor network utilization. While all of this is going on the calling thread is happily waiting for the fork()er thread to tell it when the program is complete.

2.5 Threading is not a silver bullet..

.. more of a rusty nail. Threading created lots of problems. Lots. The simple fact that threaded programming is difficult has made it so that the typical libraries are not very much fun to use in a threaded environment. glibc, for example, is only thread safe when linked against its pthreads libraries. It uses the locking primitives in the pthreads code to prevent reentry into certain parts of the library. Separating the locking exclusion code from the rest of the pthreads implementation in glibc did not sound lovely. So instead hftpd rose to the class of real applications by implementing its own memory allocator, simply to avoid reentry into glibc's allocator.

hftpd's allocator adds memory through mmap()ed regions associated with a particular thread, rather than locking around calls to brk(). Further allocations are made from these large blocks, making small consecutive *alloc()s and free()s inexpensive. Freeing all the memory associated with a thread then becomes a sequence of munmap()s. hftpd ties its allocator interface into hooks exported by glibc. This has the added advantage that allocations made by glibc itself will be associated with a thread and will be freed when that thread exits. Before this nice mechanism, allocations deep in glibc could not be accounted for and could cause memory leaks.

The allocator brought up an interesting problem. It needs a way to blindly decide what thread it is running under. Initially the thread's PID was used as a thread identifier. While this worked, and getpid() is about as light as a system call can get, it was still a significant number of system calls when getpid() was being called at every call to the allocator routines. The nasty hack that was spawned involves allocating the thread's stacks on powers of two and masking the stack pointer to find a cached version of the PID on the stack. To further obfuscate this, we 'know' where the master process' stack is mapped and know not to try and find the cached PID there but fall back to stock getpid(). While this runs very well, it did nothing for the long term portability of the hftpd.

2.6 It was a good first try

While this resulting server performs well, it still has limitations. By handling a single connection per thread, hftpd ties the limit on the number of connections it can handle to the number of threads the system can support. Non-blocking FTP protocol and transfer routines could be written that would allow multiple connections to be handled by single threads. This would also remove the connection between thread context and file system operations that caused the move to native thread instrumentation. hftpd could then use portable thread APIs and could approach something vaguely resembling portable.

3 phhttpd, static web acceleration

phhttpd has strict design constraints that make it a good test case for a non-blocking event model, it was never intended to be a fully functional web server. It serves a given set of static web pages from memory very quickly.

When phhttpd starts up it is given a list of files to serve. These files are mmap()ed into phhttpd's address space, mlock()ed so that they won't be swapped out, and read into memory. This puts a strict limit on the amount of data phhttpd can serve. The server precalculates headers for each file and places them directly in front of the mmap()ed space so that they can be served with a single write() call. writev() could have been used to save some memory at the cost of extra code, but having large contiguous regions sounded like more fun. phhttpd then stores these header and data regions based on a hash of the URL that references the cached document. Serving a page in this specialized environment is lightweight in the extreme. The new connection is accepted, the request is read in, the URL is hashed, and the response is returned to the client.

This all is driven by an event model inspired by Stephen Tweedie's SIGIO hacking. It merges SIGIO delivery on sockets with real time signal queues as specified by posix. Usually when using SIGIO a signal will be delivered when events occur to a socket for which SIGIO delivery has been registered. This is of limited use because the event that happened to the socket, and in fact which socket is involved, are not known at signal delivery time. Real time signal queues allow structures to be delivered with signals and also allow the program to queue these information structures instead of interrupting the process. Stephen's patches add SIGIO information into these siginfo structures that can be queued.

phhttpd uses queues of these siginfo structures to drive its non-blocking networking state machine. The state of sockets in the server are recorded in a table based on the descriptor number. The combination of the descriptor's state and the events that arrive in the SIGIO queues determine what operations are taken on the socket. This allows a single thread of execution to service many connections at once, only blocking when there is no more work pending in the queue.

New sockets are accept()ed into the engine based on the arrival of events on the listening sockets. SIGIO is enabled on the new socket and an initial read() is done to check for pending data. This reading is necessary to avoid a race between data arrival and the enabling of SIGIO. If data arrived on a new connection before SIGIO was enabled, no event would be triggered for the new data and the socket would appear dead to the engine. This could be avoided if Linux's stack was altered so that new sockets inherited socket options from their parent. It turns out this read is a good idea anyway. The client almost always transmits the entire request piggy-backed to the final acknowledgement in the connection handshake so connection processing latency is reduced by performing this initial read.

This system provides convenient load balancing when there are many short-lived connections. Multiple threads can be run, each with their own signal queue. To keep things simple descriptors are only handled by a single thread, with the exception of the listening descriptors. The delivery of SIGIO siginfo structures is tied to a thread by associating the thread's PID with the descriptor through fcntl(). After accept()ing a new connection, the thread currently owning the listening socket can associate it with the next available thread. This gives simple round robin load balancing amongst the threads and gets nearly ideal connection distribution. This can lessen the latencies in connection processing in an SMP system, and can minimize delays if one were to use this event model while accessing relatively slower file systems.

Once a complete HTTP request is read through the event machine, the request is hashed. The hashed URL is compared with the URLs for files that are in phhttpd's cache. Should the URL not be found, phhttpd simply returns an error. If a document is found that matches the URL, it is returned to the client with a large write(). In the optimal case the write() will fit in the TCP buffers for the socket so phhttpd can close() this connection and let the kernel finish writing the data. Large files are written in pieces driven by SIGIO events that arrive when space is available in the TCP buffers.

3.1 Event Queue Coherency

Here there be stale event dragons. close()ing a socket does not remove pending events for that descriptor from the queue. A socket must not be close()ed, releasing it's descriptor for reuse, until there are no more events for that descriptor pending in the queue. phhttpd maintains clever data structures which determine when it is safe to finally close() sockets. A queue records the number of sockets pending for close() at each event loop iteration, and a FIFO contains the actual descriptors to be close()d. The queue must be as long as the signal queue is deep, and the FIFO must be as deep as the maximum number of descriptors that may be open. At each event loop iteration the next element in the queue is referenced. The number of sockets indicated by this element in the ring buffer are popped off the FIFO and close()ed. Sockets are added to these structures as their connection is terminated when events are pending in the queue. SIGIO is disabled on the socket and the socket is appended to the FIFO. The queue is updated to indicate that this inserted socket should be removed after the pending signal queue is flushed by further loop iterations. This setup works because the event queue can only shrink by a single element at each loop iteration, based on sigwaitinfo(), but can grow by any amount at any time. The overhead of maintaining these structures is cumbersome, but scales better than walking signal queues every time descriptors are closed.

3.2 Go, phhttpd, go

The resulting engine is efficient and scales well. When serving a few hundred clients over multiple 100 megabit interfaces, the CPU usage of phhttpd itself is in the 0-1% range on high end X86 machines. The kernel is left with the rest of CPU time to deal with protocols and interfaces. In addition, the code paths through the signal handling code show up substantially lower in the profiles than the select() and poll() that would service classic IO engines.

3.3 Further Work

It is hoped that this model can be merged into applications that could make use of its scaling properties. squid [squid] could be hacked to have the signal queues replace its select() loop. The apache MPM code currently being developed [apache] could have a SIGIO queue back-end that drives its asynchronous response model.

4 Acknowledgements

Credit is due Alan Cox and Stephen Tweedie for nice design inspiration. Phil Scwhan implemented parts of hftpd that I was avoiding, for which I owe him at least a beer. Many thanks also to the small army of folks that pummeled me with the grammar stick.

5 Availability

`http://www.zabbo.net/hftpd`

`http://www.redhat.com/~zab/phhttpd`

References

[pftpd] Peter Eriksson's nice pthreads FTP server, `ftp://ftp.lysator.liu.se/pub/unix/pftpd/`

[squid] Squid Internet Object Cache, `http://squid.nlanr.net/`

[apache] Apache development resources, `http://dev.apache.org/`

[freshmeat] A sampling of available FTP servers, `http://freshmeat.net/appindex/daemons/ftp.html`

Porting Linux to IA-64

Stéphane Eranian David Mosberger

Hewlett-Packard Laboratories
1501, Page Mill Road Palo Alto CA 94303 USA

{eranian,davidm}@hpl.hp.com

Abstract

The IA-64 architecture, co-developed by HP and Intel, is going to reach market some time next year with Merced as its first implementation. Major industry players have endorsed this new architecture and technical details are gradually becoming publicly available. However, the complete architecture will not be fully disclosed until machines become available. To provide for early availability of Linux on IA-64, in February 1998 HPLabs began a project to bring Linux to this new architecture with the eventual goal of releasing it to the open source community. This paper gives an overview of the IA-64 architecture and describes our effort so far.

1 Introduction

At HPLabs, we have been working on porting Linux to the IA-64 architecture since February 1998, an activity which is now part of a broader industry effort. The initial goal of our project was to produce a self hosting system that would be available when the first IA-64-based products would appear. Given the progress made so far, we are now looking into producing a fully optimized, complete Linux distribution with most standard packages available. We intend to release all the code to the open source community for eventual integration into the official code base when machines become generally available sometime next year.

Bringing Linux to a new architecture is more than just porting the kernel. To become really usable a system must include a development environment i.e., a complete tool chain, a kernel, the C and math libraries and thousands of tools and commands.

This paper gives an overview of IA-64 architecture with code samples to illustrate some key features. We describe how we brought the various pieces together by first producing a complete tool chain and creating a comfortable simulation environment, then working on the kernel and the C library until we reached the point where building real applications became possible.

2 IA-64 overview

The first implementation of the HP/Intel co-designed IA-64 architecture i.e., the Merced CPU, will reach market sometime next year and will be quickly followed by the faster McKinley[1] in 2001, Madison and Deerfield in 2002. This new architecture builds upon lessons learned from RISC, CISC and VLIW processors. It introduces a new computing paradigm called EPIC (Explicitly Parallel Instruction-set Computing). The basic idea is to expose instruction level parallelism (ILP) to the compiler and use faster and relatively simpler hardware. The compiler operates at a much higher level and has potentially a broader understanding of what the program is trying to accomplish leading to more optimization opportunities.

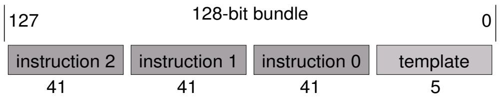


Figure 1: IA-64 Instruction Format

As for VLIW processors, IA-64 instructions are grouped into bundles as shown in Figure 1. Each bundle contains three instruction slots of 41 bits each and a 5-bit template field that encodes which execution unit types are needed by the instructions (M-unit for memory access, I-unit for integer operations, F-unit for floating point or B-unit for branching) and where a stop bit is needed. This synchronization point (barrier) may be necessary between instructions to avoid conflicting operations, like a register read after write dependency, for instance. Parallelism can span more than one bundle and the stop bit can be introduced in the middle of a bundle.

A total of 128 integers registers of 64 bits each and 128 floating point registers of 82 bits each are available. Integer registers between 32 and 127 are called "stacked registers" and are used with the stack engine during function calls. The architecture uses a simple load/store model like RISC and introduces some new features along with the more traditional capabilities expected from a modern CPU, such as multimedia instructions.

The following C code:

```
r2 = r1 == 0 ? r4+r5: r3+r6+1;
```

gets translated into:

```
     cmp.eq p1,p2=0,r1;;
(p1) add r2=r4,r5
(p2) add r2=r3,r6,1
```

Figure 2: example of predication

The concept of predication is implemented using 64 predicate registers of 1 bit each (`true` or `false`). Most instructions can be predicated; if the predicate evaluates to `false`, the instruction is simply not executed. This mechanism can avoid costly branches as is demonstrated in figure 2 with a classic `if-then-else` statement. Predicate `p1` will be set to `true` if `r1` equals zero, p2 will take the opposite value (`p2=!p1`) and only one of the two `add` instructions will be executed, without requiring any branches.

Another feature of IA-64 is control and data speculations, which provide ways to safely move loads off the critical execution path without having to worry about exceptions that could occur, like NULL pointer dereference. A compiler can take advantage of this mechanism to hide memory access latency. Speculation is available for both integer and floating point loads.

Control speculation is the execution of an operation before the branch which guards it. Data speculation is the execution of a load instruction before a potentially conflicting store (aliased address) and is also called advanced load.

The safety of the operation is ensured by the fact that failed speculative loads don't generate faults but instead mark their target register as invalid using a NaT (Not a Thing) bit i.e., the 65th bit of each register. In the case of floating point registers, a special value called NatVal is used instead of an extra bit. Several check instructions can be used to determine whether the load succeeded or not. In case of failure, a normal load can simply be executed or a jump to a recovery code is possible when more operations are required. Advanced loads rely on an internal table called ALAT (Advanced Load Address Table) which is used to check whether or not the target register of the advanced load contains stale information with regards to stores which might have happened after it. Figure 3 shows how a load can, speculatively, be moved before a branch. If the load (`ld8.s`) fails then the NaT bit will be set on `r1` and the check (`chk.s`) will jump to the recovery code.

```
(p1) br.cond label
     ld8 r1=[r5];;
     add r2=r1,r3
```

Can be transformed into:

```
     ld8.s r1=[r5]
     // do something else
(p1) br.cond label
     chk.s r1, recovery_label
     add r2=r1,r3
```

Figure 3: Example of control speculation

To avoid unnecessary registers spills and fills operations on function calls, IA-64 provides a dynamic register renaming scheme which is depicted in figure 4: `rXX` shows the logical numbers whereas the bars represent the physical register file. Registers `r47-r51` are holding parameters to pass to function `B`. The branch instruction (`br.call`) causes the stack frame to "virtually" move forward by renaming `r47` to `r32`. The `alloc` instruction simply resizes the current frame to accommodate local variables. So each time you enter a new function a "fresh" set of registers is available and the compiler does not have to worry about spilling/restoring callee/caller save registers. Registers outside of the current stack frame are considered "dirty" and if no more physical registers are available to satisfy the `alloc` instruction, the register stack engine (RSE) will spill the "dirties" onto a designated backing store location in memory. When returning from a function call, the saved registers are automatically restored from memory.

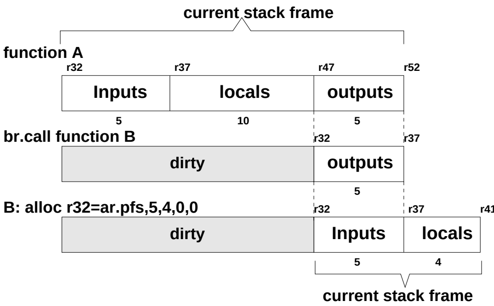


Figure 4: RSE behavior on function call

Finally, IA-64 provides a powerful register rotation mechanism to do software pipelining and unroll loops without incurring code expansion. Integer, floating point and predicate registers can be rotated during a loop creating the illusion of a pipeline. We'll give an example of this feature in section 4.

More details about the disclosed capabilities of the architecture can be found on HP's IA-64 web site[1] and in the Application Instruction Set Architecture Guide[2]. Intel's web site also provides online architecture overview tutorials[3].

3 First steps

This is possibly the first time that Linux is being ported to a new platform before any hardware is available. This means not only that we have to rely on simulation for most of the project but also that very few development tools exist.

Linux heavily relies on using a GNU C compiler for both the kernel as well as for user level code, like the C library. Such a tool did not exist for IA-64, therefore we decided to work on it first. The obvious candidate was `egcs`, a very active branch off the gcc development tree. Creating an optimizing compiler for EPIC is not a trivial task and would have required changes to the `egcs` front-end. Such effort was clearly out of the scope of this project. Instead, we focused our attention on producing a functional back-end which generates correct code but doesn't try to use EPIC features like speculation or register rotation, for instance. Cygnus maintains the GNU C compiler and has officially announced last April that they will produce an optimized version of the GNUPro toolkit for IA-64[2]. So as compilers improve, we expect to simply recompile our code in order to get better performance.

A compiler by itself is not enough; the GNU assembler, GNU linker, binary object manipulation library (BFD) and tools like `nm`, `objdump`, `ar` are also required. So we ported the GNU `binutils` package. The tool chain uses `LP64` as the programming model (Longs and Pointers are 64 bits) and the binary format is the official `ELF64` as defined for IA-64. By June 1998, the tool chain was able to pass the `gcc` test suite and the `"Hello World!"` program was generated correctly.

For the execution environment we use a simulator, developed by HP, which emulates the full instruction set of the CPU but not the whole platform i.e., no PCI nor BIOS emulation. It supports two modes of execution: user or system. The former allows us to run user-level applications and execution traps into the simulator for system call emulation. In system mode, kernel bring up is possible as the full VM and interrupts are simulated. Access to I/O devices is achieved by having special device drivers in the kernel which trap into the simulator and get service from the host OS. To simplify user-mode emulation, we have ported this simulator from HPUX to Linux/x86 which allowed a one to one mapping for system calls and parameter marshaling resorted, most of the time, only in 64 to 32 bits conversions.

Once the tool chain and the simulator were in place, the whole development environment was running on Linux/x86 and work on the kernel could really begin.

[1] See `http://www.hp.com/go/ia64`

[2] See `http://www.cygnus.com/news/ia-64.html`

4 The kernel

We started working on the kernel in late October 1998. Our goals for the kernel were as follows:

- deliver a straight port minimizing the changes to the machine independent part of the code
- follow very closely the development of the official kernel as this would make the final integration phase much smoother.

We have kept our modifications very localized creating new files in two machine dependent directories, namely `arch/ia64` and `include/asm-ia64`, which made it quite easy to follow the latest official kernel developments. In October 1998, we started with `2.1.126` and we are currently running `2.3.X`.

The kernel is running in native 64-bit mode and uses little-endian byte ordering for obvious compatibility reasons with x86. The page size is currently 8KB and the virtual address space of a user process is 43 bits (8TB).

In order to get access to I/O, we developed a series of interrupt driven device drivers which trap into the simulator to get service from host OS. We built a SCSI driver (`simscsi`), a console driver (`simserial`) and later an Ethernet driver (`simeth`). The SCSI driver is very simple and calls the simulator for read/write requests using `[offset,size]` tuples. The disk is emulated using a file on the host as a disk image. Using the `loop` device, we can easily transfer files back and forth between the host and target. This driver also allows us to exercise the complete SCSI code. The serial console driver traps into the simulator for get/put character and an `xterm` is used as the front-end. The Ethernet driver manages raw Ethernet frames which are obtained, via the simulator, from the real interface on the host using raw sockets. The interface is put in promiscuous mode and using a block packet filter, we were able to allocate a specific IP address to the simulated kernel.

We took the incremental approach of bringing up subsystems one by one. We began in late October with an almost completely commented out `start_kernel()` function. At that point we had only the kernel banner working. Since, we have been enabling components like VM and interrupts which allowed us to get through the famous `BogoMips` loop. Shortly after we added context switches and by Christmas we were able to create and run kernel only threads.

Then, we added support for system calls and we landed in user mode in January and were able to execute the `"Hello world!"` program produced a few months ago. At that time we did not have a complete C library, therefore it was impossible to recompile standard applications and run them on the kernel. Instead, we had a μlibc i.e., an extremely small subset of a classic which we used to rewrite simple test programs like a tiny shell (`tsh`), `ls`, `cat`, `mount`, `halt`, etc. Those tools helped us recreate a familiar and comfortable test environment.

In early March, the network stack was up and running. The system had its own IP address and you could ping in and out as well as login from any remote machine. Here again, we rewrote simple versions of `ping`, `rlogin`, `inetd` and `ifconfig`. By Easter, we had signal support and a few weeks later `ptrace` was in place (including system call tracing, single stepping and peek/poke) and it became possible to use the `strace` program for debugging purposes.

```
     ...
     init p6 to true
     add r17=8,r16
1:
     ld8.s r32=[r16],16
     ld8.s r34=[r17],16
     czx1.r r14=r33
     czx1.r r15=r35
     ;;
     cmp.eq     p6,p0=8,r14
     cmp.eq.and p6,p0=8,r15
(p6) br.wtop.dptk.few 1b
     ...
```

Figure 5: core loop of strlen_user()

As an example of how to combine control speculation with register rotation inside the kernel, we show, in Figure 5, an actual code sequence extracted from `strlen_user.S`[3]. Normally, this function uses an optimistic exception scheme to avoid systematic bounds checking. When a fault is detected, execution goes through an exception table and branches back slightly later in the code with some registers holding special error codes. This exam-

[3]Usually found in `arch/ia64/lib`

ple demonstrates how one can use control speculation to achieve exactly the same goal[4]. We use two pipelines of depth 2, `[r32-r33]` and `[r34-r35]`. We load 8 characters at a time (`ld8.s`) speculatively which is handy to safely look way forward in the string. Each iteration loads 16 bytes[5] taking advantage of the memory bandwidth (2 memory operations allowed per bundle) and looks for the zero byte in the previous 16 bytes. Registers `r16` and `r17` are initialized 8 bytes apart and used as base pointers on the string. They are automatically incremented (by `16`) by the load. The `czx` instruction returns the position of the zero byte or `8` if not found.

Data is inserted in `r32` and `r34` and gets "rotated" each time we go around the loop, it is eventually consumed when it reaches `r34` and `r35` at the next iteration. In reality registers are simply renamed (no data copied) by group of eight (`[r32-r39]`). The illusion of smaller pipelines is created by always entering data at fixed "stages", like we do for `r34`. In case we go too far ahead and hit a page that's not mapped, when the register gets used in stage 1, its NaT bit will be set and the parallel compare instructions will result in `p6` set to 0 (`p0` acts as a sink in this case) forcing the execution out of the loop.

5 The User land

Once you have a kernel, the work is far from being done as most of the code lives at the user level. First, the C and math libraries need to be ported, then thousands of commands, tools and extra libraries need to be recompiled and sometimes fixed.

While we were doing kernel work at HPLabs, CERN[6] decided to join our effort and started working on those libraries. The first goal was to deliver a generic port and then, look at doing EPIC optimizations for performance critical routines. Linux is using the GNU libc version 2.1 on the major platforms and was, thus, the obvious choice.

After just three weeks of intense work, they were able to run the `"Hello World!"`. With the first code drop from CERN we were able to compile real world applications. We quickly managed to recompile a complete login sequence with `init`, `mingetty`, `login` directly using RPMs[7] from standard distributions. Soon, we had the other basic packages like `util-linux`, `sh-utils`, `fileutils` and even `netkit-base`.

Because we were still missing some libraries, noticeably `curses` some packages were still incomplete but we managed to get shells like `pdksh` and `bash`. We also got our first full screen editor with `vim`, a vi-clone.

Porting existing packages can be very tedious as code quality varies a lot. Most of the problems we've encountered so far revolve around non 64-bit clean code. As an example, the basic `ping` command is still not clean even though one can argue that it runs on Linux/Alpha but it's simply because it relies on the unaligned access trap handler. This is clearly not a long term solution and programs must be gradually fixed.

It is our goal to get a complete distribution, so the basic libraries and utilities are just the first step and work is needed to port other, possibly larger, packages. Clearly a GUI is needed and X11 i.e., XFree86, its associated applications, toolkits and desktop environment like GNOME/KDE will need to be ported. A decent debugger, namely `gdb`, must be also be available for any serious development to become possible. These days a system wouldn't be complete without a web browser and thus Mozilla must also be worked on. Languages like Java, Perl, Tcl/Tk, GNU Fortran, Python must be ported as well.

6 Next steps

As of today, we have a complete IA-64 development environment hosted on Linux/x86 and based on `egcs-1.1.2`, `gas-990404` and GNU libc v2.1. Our tool chain produces functional code and has proven to be quite robust to get us that far. We have a working kernel with major subsystems enabled. Many real world applications are running on our kernel and also directly on Linux/x86 using the user-mode simulation.

[4] The whole function is not shown for space reasons

[5] `r33`,`r35` are properly initialized before the loop

[6] Centre Européen de la Recherche Nucléaire – Geneva, Switzerland, see `http://www.cern.ch`

[7] Redhat Package Manager, see `http://www.rpm.org`

We are actively collaborating with other industry partners like Cygnus, Intel, SGI and VA Linux Systems (Trillian project) and expect to see a major development speed up from this broader effort.

In the near term, we're planning on working on the kernel to fill out the missing pieces like SMP support, the platform specific code, the boot loader and also the IA-32 emulation. Our CERN partners will continue to work on the libraries and noticeably on the dynamic linker and various optimizations. As Cygnus moves forward with their compiler work, we expect to see major improvement to our code. We also intend to tackle the large application space.

While this project still needs to be protected by strict non-disclosure agreements (NDAs), we are keeping Linux developers abreast of our progress and intend to share as much as we can as more information about the architecture is disclosed to the public. Even though it may be hard to join this project, you can still help significantly by making sure than any code you write or read is 64-bit clean. This not only means looking at all abusive casts but also at things like hardcoded data structure sizes and others bad coding habits. We discovered that many of the packages used with Linux don't have good validation tests, so another way of helping would be to develop good test suites. While no tests can be perfect, it would help catch some errors very early on.

7 Conclusion

After giving a rapid overview of the major features of IA-64 we have described what it really takes to port a complete Linux system to this new architecture. At this point in time we have brought forward a complete GNU-based tool chain, a simulator, most of the kernel and the beginning of a real Linux distribution.

While the non-disclosure restrictions make it hard to work completely in the open, we are trying to stay as close as possible to the spirit of open source projects by working with outside partners whenever and wherever appropriate. By doing so, we've made significant progress and think we are on time to deliver a complete distribution to the open source community sometime next year when machines become available. We also hope that our effort will jumpstart a Linux community around this new exciting architecture.

References

[1] Linley Gwennap. Intel outlines high-end roadmap. *Microprocessor Report*, pages 16–19, October 1998.

[2] Hewlett-Packard Company/Intel Corporation. *IA-64 Application Instruction Set Architecture Guide*. http://www.hp.com/go/ia64.

[3] Intel Corporation. *Merced Processor & IA-64 Architecture*. http://developer.intel.com/design/IA64.

The InterMezzo Filesystem

Dr. Peter J. Braam
Carnegie Mellon University & Stelias Computing
braam@cs.cmu.edu

Michael Callahan
The Roda Group
mjc@rodagroup.com

Phil Schwan
Stelias Computing
pschwan@thepuffingroup.com

Abstract

Is it possible to implement a distributed filesystem, retaining some of the advanced protocol features of systems like Coda, while getting an implementation that is an order of magnitude simpler? We claim that the answer is ”yes”, and InterMezzo is a prototype demonstrating this claim. The key design decisions were to exploit local filesystems as both server storage and client cache and make the kernel filesystem driver a wrapper around a local filesystem. We use a rich, high level language and work with asynchronous completion instead of threads. Finally, we rely on existing infrastructure, such as TCP. This is in contrast with systems like Coda and AFS, which implement their entire infrastructure from the ground up, in C. This paper will describe the design and implementation decisions used in our prototype.

1 Introduction

Distributed file systems allow sharing of files between multiple computers. Such systems make it feasible to access a single copy of a file from multiple computers, and help in system administration, for example with backups. More recently, distributed file systems have been expected to support mobile computing and high availability. Two key features meeting these requirements are disconnected operation with reintegration and conflict detection, and server replication. Disconnected operation solves the desktop/laptop synchronization problem, while server replication offers high availability. The Coda file system pioneered these features.

Designing and implementing distributed filesystems poses a major intellectual challenge. Many excellent ideas have come out of research projects, and some systems offer many advanced features.

The basic question we wanted to answer is the whether or not an interesting distributed filesystem can be built with an order of magnitude less code than a system like Coda [coda] has, but retaining some of the advanced features. We claim the answer is ”yes”. In this paper we will describe some of the design and implementation decisions for the InterMezzo filesystem.

To keep our project modest, we are first targeting the task of online directory replication on a system area network, allowing for network and system failures. This exposes many of the subsystems needed, but leaves some others aside. However, our paper describes the broader design issues, parts of which will be implemented and likely modified as InterMezzo develops.

2 Acknowledgements

The authors thank the other members of the Coda project for fruitful discussions. This research was supported by the Air Force Materiel Command (AFMC) under DARPA contract number F19628-96-C-0061. IBM and Novell provided additional support. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of AFMC, DARPA, IBM, Novell, Carnegie Mellon University, Stelias Computing Inc., The Roda Group LLC, or the U.S. Government.

3 Designing Distributed Filesystems

There are several components to distributed filesystems. First, network filesystems have clients, which can fetch data from servers and send modifications to servers. This requires network request processing including bulk transfer. The fact that the client gains filesystem access to the files requires a kernel level filesystem driver. Without adding bells and whistles here we already have a list of complicated components.

Delicate protocols and versioning information are needed to provide good semantics of sharing. For example, a server may have to notify clients that cached versions of the files or directories are out of date. This requires the client to both make requests to servers, on behalf of client processes accessing and/or modifying filesystem objects, as well as to answer requests from the server to maintain cache validity.

Performance is another important consideration, which usually becomes pervasive in a design. One must minimize the number of network interactions, and also make these interactions as efficient as possible. Security, management and backup add another dimension to a filesystem project.

Given this picture, it should not come as a surprise that not all of the advanced filesystem projects have delivered systems that gained widespread use. Fortunately, many features and ideas have been explored and thoroughly evaluated during the last 15 years. A central question that has not been answered appropriately is how to implement filesystems reliably, with a reasonable amount of code.

Here something can be learned from other distributed systems. Ericsson was facing the daunting task of implementing control systems for telephone exchanges. The Erlang language [erlang] was developed as a result of extensive studies on how to construct reliable distributed systems. The Teapot [teapot] project at the University of Wisconsin explored domain specific languages enabling construction and verification of distributed shared memory software. This toolkit was subsequently used by the Berkeley XFS project [xfs]. The ACE toolkit [ace], developed at George Washington University, provides extensive C++ frameworks for building distributed systems.

Interestingly, all of these projects have reached conclusions bearing considerable similarity: a state machine with asynchronous event notifications is used, which gives a better overview of the distributed state than the classical "multi-threaded" server models.

Another key difficulty with filesystems lies in the kernel code. Linux development has shown that to get a correct implementation of even a relatively simple filesystem such as NFS, numerous subtle issues have to be addressed, resulting in a long arduous development path. UCLA's Ficus [ficus] project made an important contribution in this field, namely to use stackable filesystems [stackable], also available for Linux [zadok]. These are filesystem layers that leverage and extend the functions of other filesystems. We use this approach in InterMezzo and avoid the difficulties associated with a full filesystem implementation.

InterMezzo is at present a prototype, and the first target for the system is to do reliable directory replication between two nodes, on a secure system area network. Its protocols are suitable for extensions, and a fully featured distributed filesystem can be constructed from this starting point.

4 Overview of InterMezzo

Faced with a multitude of design choices, we drew up the following list of requirements:

1. The server file storage must reside in a native filesystem.
2. InterMezzo's client kernel level filesystem should exploit existing filesystems, and have a persistent cache.
3. Filesystem objects should have meta-data suitable for disconnected operation.
4. Scalability and recovery of the distributed state should leverage scalability and recovery of the local filesystems.
5. The system should perform kernel level write back caching.
6. The system should use TCP and be designed to exploit existing advanced protocols such as rsync for synchronization and ssl/ssh for security.
7. Management of the client cache and server filesystems should differ in policy, but use the same mechanisms.

Many issues remain open. For example, we have not yet decided if we should identify InterMezzo file objects (i.e. files & directories) through file identifiers, path names or server/device/inode triples. Though we currently use path names, our system is designed to enable us to switch from one model to another with a relatively small effort.

Security and management are important. AFS and Coda set good examples for management of server space by working with file sets, a.k.a. volumes. File sets are groups of files that form a sub-tree of the InterMezzo name space, and which are typically much bigger than a single directory and much smaller than an entire disk partition. Coda and AFS, as well as DCE/DFS and Microsoft dfs, have a single name space. This means that all client systems see one large file tree containing all the file sets exported by InterMezzo servers in a cluster.

Often it is desirable to work with a small collection of file sets separately, so we left room to manipulate the InterMezzo exported file sets more explicitly than in those systems.

5 File sets, mount points, servers and clients

Each file tree made available by a cluster of InterMezzo servers is built up of file sets or volumes, which are similar to Coda and AFS volumes. Each file set has a root directory and can contain InterMezzo mount points of other file sets. An InterMezzo mount point is a concept similar to but distinct from a Unix mount point. A client can have any file set as the root of an InterMezzo filesystem.

A file set has a file set storage group associated with it, describing the server holding the file set. The file set location database describes the mount points of file sets, as well as their storage group. An InterMezzo cluster is a group of servers sharing a single file set location database (FSLDB).

A client will be able to choose any file set as the root for a mounted instance of an InterMezzo filesystem. If the root file set is not the default, then all ancestors of the mount point of that file set are invisible; this accommodates exporting sub-trees of the entire cluster file tree. File sets contain files and directories, and generally user level software is not aware of the presence of file sets.

The file set location database is implemented as a sparse tree of directories, handled as a special object in the filesystem. An update record is associated with each modification of the file set tree, and version numbers are associated with each update. This allows the file set database to be updated, without downloading the latest version (which is also kept).

InterMezzo makes a distinction between clients and servers, but mostly as a matter of policy, not mechanism. While the same mechanisms are used, the server maintains the on-disk structures as authoritative records of file state, and also coordinates state coherence across the InterMezzo clients. Each server will also be an InterMezzo client, but the filesystem layout on the server is slightly different from that on other clients. A file set mount point on a server is not necessarily a directory in the InterMezzo filesystem as it is on a client. If the server stores the file set, a symbolic link is placed at the mount point, which targets a directory on the server holding the data. This target directory is the file set holding location (FSHL).

6 Journaling updates and filtering access

An InterMezzo cache is simply a local media filesystem, which is mounted with an extra filter layer wrapped around this local filesystem. The filter layer is called ”Presto” for Linux and ”Vivace” for Windows platforms. The purpose of the filter is twofold:

1. Filter access, to validate currency of the data held in the local filesystem
2. Journal updates made to the filesystem

All requests emanating from Presto are handled by Lento, the user level cache manager on the system. Note that Lento acts both as the file server and as the client cache manager. Figure 1 graphically describes the operation of the InterMezzo filter driver.

The code in Presto is quite straightforward. When InterMezzo is mounted, Presto is informed of the filesystem type that it is wrapping. It remembers all VFS methods associated with dentries, inodes and files of the wrapped filesystem in the ”bottom_ops” structures. The ”bottom_ops” are typically those of the ext2 filesystem. Presto must filter all access and journal all updates, but must make an exception for the process managing the cache, which is named ”Lento” and will be discussed later. Lento's modifications of the cache should be immune from filtering or journaling.

As an example let us describe the open call for directory inodes. When an object is accessed and it is not present in the cache, the object is fetched. Files are presently fetched in their entirety. Directory fetching is implemented by creating the name, and creating sparse files and empty subdirectories for each of their entries. While this introduces significant latency in the access to directories, compensation comes from the fact that the subsequent acquisition of attributes of the entries is entirely local.

In pseudo code, the kernel open method has the following structure:

```
int dir_open(inode *inode, file *file)
{
        if (!HAVE_DATA(inode) && !ISLENTO) {
                upcall(inode, "get_data");
        }

        return bottom->dir_open(inode, file);
}
```

InterMezzo uses upcalls from the kernel to request service from the user level cache manager. In the snippet above, this is done with the upcall function.

Updates to objects are made as follows. First a Permit is acquired to execute concurrency control. On clients, the updates are made in the cache and journaled in a client modification log for reintegration to servers. The server holding the file set repeats the modifications made to the volume and will forward the modification log to clients registered for replication. We call the process of forwarding modification logs to the clients ”update notification”. The processes of reintegration and update notification are almost identical.

The modification log details modifications made to directories and file attributes, as well as all close

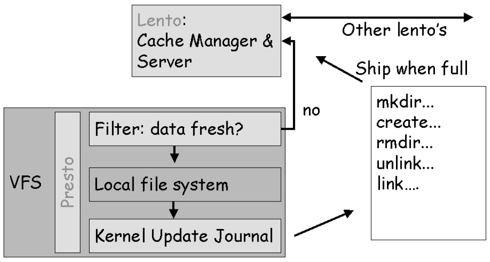


Figure 1: Network Request Handling

operations on files. Such files, which have been closed after a modification, are then "backfetched" by the server. If a server detects that it does not have a directory created by the client, it recursively back-fetches the contents of that directory from the client.

The mkdir method in Presto is roughly implemented as follows:

```
int mkdir(inode *inode, char *name)
{
        if (!HAVE_PERMIT(inode) && !ISLENTO) {
                lento_getpermit(inode);
        }
        rc = bottom->mkdir(inode, name);
        if (! rc && ! ISLENTO) {
                journal("mkdir", inode, name);
        }
        return rc;
}
```

A client can place a replication request for a file set with the server holding the file set - the server will then propagate all updates it receives to this client.

If a file set is modified on the server, an almost identical process takes place. The updates are journaled but now only propagated to clients registered for replication. Other clients will invalidate their caches when they learn of the updates, and re-fetch the data. In order to accomplish this, the holding location of a file set is itself mounted as an InterMezzo filesystem to enable journaling of updates.

We see a high degree of symmetry in the operation of clients and servers, and Lento running on clients and servers must be capable of:

1. File Service: fetching (on clients) and back-fetching (on servers) of files and directories

2. Reintegration service: receiving modification logs (on servers) and update notifications (on clients) for reintegration

We see that both servers and clients should be capable of making modifications and serving data.

A key distinction is that a server implements security while a client can trust an authenticated server and implement changes without checking permissions on a per-object basis. Secondly, a client may make changes to its cache solely for freeing up space. Such modifications are, of course, not propagated to servers.

In a trusted environment, an optimization can take place, which simply changes the holder of the file set to the system modifying the file set. This system then is responsible for updating all replicators of the volume.

7 Protocols

The Coda, AFS and DCE/DFS protocols [dfs-protocols] have many attractive features, and our intention was to preserve many aspects of these.

Each cached object (i.e. directory or file) has attributes HAS_DATA, and HAS_ATTR to indicate if its data/attributes are valid in the cache. Before accessing an object, the client will verify that it is current, and re-fetch it if necessary. The protocol has a GetDir and GetFile call for fetching and a Validate call for assessing currency. The presence of a HAS_DATA or HAS_ATTR flag is called a callback and enables the client to reuse the object without contacting a server.

The server will notify clients holding callbacks on objects before storing modified objects. InterMezzo's design could also allow breaking these callbacks before the modification is initiated. In either case, a server-initiated request BreakCallback is issued. The BreakCallback request is handled as a multi RPC, as in Coda. This means that the BreakCallback request is sent out to all clients and only then replies are gathered – this avoids multiple timeouts. When data is to be modified, a client will get a permit to do so. There is a GetPermit request to do this. A successful permit acquisition is indicated with a HAS_PERMIT flag in the status field.

The act of breaking a Permit can in itself be a highly complicated event. When the holder of a Permit is asked to relinquish it, it must first reintegrate any pending modification logs, as well as handle any error conditions that might result.

Every object will be identified by an identifier and a version stamp. Our protocols will guarantee that two objects with the same identifier and version-stamp are identical. In addition to object version stamps, volumes have stamps too, and allow for rapid revalidation of entire volumes, in the common case where no objects in the volume have changed.

InterMezzo can validate cached objects with Validate requests. As in Coda, these can be issued at the volume and filesystem object level. Because we anticipate that InterMezzo's first use will be as a replication server, initial releases will depend largely on the mtime to assess currency of data.

The propagation of updates is done through the Reintegrate request. This request uses the modification log in a way that is similar to Coda's reintegration. The client modification log (CML) is shipped to the server. The server proceeds to incorporate the changes to the directory tree and then, for any close operations in the CML, fetches the files from the client.

Coda avoids many race conditions by including the version with the RPC for the file affected by the update. For example, the creation of a new directory would include the version of the parent directory present on the client. This allows the server to make sure that the server version being modified is that of the client. If the versions are unequal, the reintegration of this modification causes a conflict, which needs special handling. InterMezzo will do this as well.

So at present InterMezzo's protocol is very simple.

8 Lento - cache manager and file server

Lento is responsible for handling file service requests from the network or the kernel. Traditional servers and cache managers are implemented using a thread pool model. The requests come in and are placed on a queue. Worker threads pick up and process the requests, and block while doing I/O, during which other worker threads can continue

We chose to start with a single threaded event driven implementation based on asynchronous I/O. In Lento, requests come in and a session is instantiated to handle them. A session is not a thread, but a data structure that contains state and event handlers. I/O is done asynchronously and a kernel signals the completion of I/O, through the dispatching of events. The kernel activates all event handlers and is also responsible for garbage collecting sessions when no events can reach them. Sessions have process style relationships.

For prototyping we used the POE (Perl Object Environment) toolkit [poe]. POE implements a session framework as described above and has Wheels that consist of Drivers to do the I/O and Filters to pre-process the data received.

We added two wheels to POE. The PacketWheel is there to send and receive network requests, which the filter unpacks and then gives to the connection. The connection determines the destination session for the request, which can be an existing session or the request dispatcher, in case a new or specially chosen session must handle the request. Our UpcallWheel unpacks requests originating in the kernel, which reach Lento through a character device /dev/presto.

These wheels have been combined with an Acceptor, which is a SocketFactory Wheel that creates accepted TCP connections.

The processing of net and kernel requests is graphically indicated in figures 8 and 8.

As an example we will give the pseudo code for the upcall session servicing a FetchFile request from the kernel. The notation below describes a session as a hash of event handlers.

```
Fetchfile = new POE::session
        (init => {
                if (!have_attr)
                        req_attr(attr_arrived);
                else
                        post(fetch_data);
        },
        attr_arrived => {
                if (status == success)
                        post(fetch_data)
                else
                        destruct_session(error);
        },
        new_filefetch => {
                queue_event(this) ;
        },
        complete => {
                reply_to_upcall;
                handle_queue;
                destruct_session;
        }, ......
        );
```

Each of the event handlers may engage in a blocking I/O operation, for example to fetch the attributes. In doing so it indicates what event will complete the asynchronous operation. Ultimately, when the data is all on the client, the request completes to the kernel by replying to the upcall. A queue associated with the session allows further requests to fetch the data for the same object to be handled by the session that is already in the process of fetching.

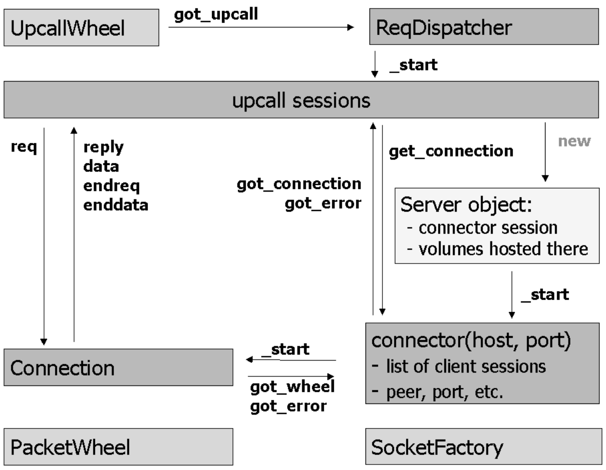


Figure 2: Upcall Handling

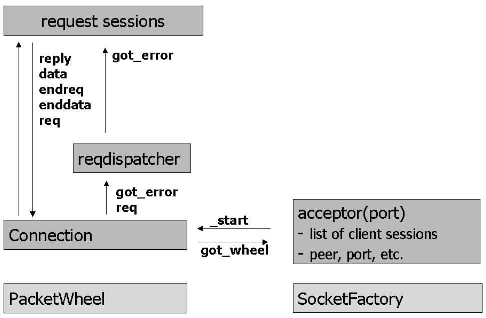


Figure 3: Network Request Handling

Lento also needs a variety of daemon sessions, which are not garbage collected but perform simple periodic or persistent tasks. For example, we have the ReqDispatcher, which sets up new request handling sessions. For the management of client initiated connections, we use Connector daemons, which create a TCP socket connect to a server, and manage the sessions using the connection.

Network packets contain Asynchronous Com-

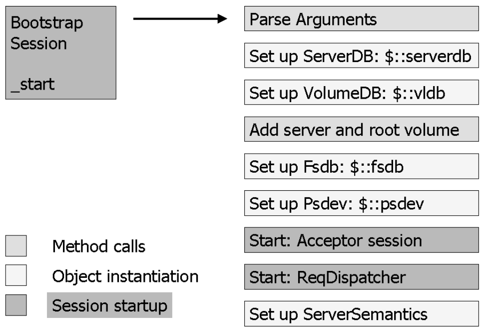


Figure 4: Lento Startup

pletion Tokens (ACTs [ace]) which encapsulate the source and destination sessions for request handling. There are 6 types of packet: requests, messages, replies, data, end-of-data and end-of-request packets.

Lento also uses various tables: tables exist for servers, volumes, and filesystem objects, each with their own attributes. When Lento starts a number of data structures are set up, as depicted in figure 4.

Some of the set-up processes are quite involved. For example, setting up the filesystem database (Fsdb) may involve recovery. (Currently this consists simply of reintegrating any pending records and, if the client is a replicator, downloading update logs from the server.) The server semantics supplement the Fsdb database with callback and permit tracking, as these are released and acquired by clients.

9 Recovery and cache validation in more depth

On Linux, InterMezzo can use the ext2 filesystem to hold server and cached data. We envisage having a fast and sophisticated filesystem database (Fsdb) for extra meta-data required by InterMezzo. This would be managed by Lento, and would hold the version information for cached objects, as well as inode-to-filename or inode-to-file-identifier translation information. If a client crashes, some of the data will be held in buffers, other data will be on the disk, and it is necessary to think through precisely what is needed for clean recovery.

The simplest recovery process is to simply wipe the cache, which for large caches is clearly very undesirable. The basic attribute for establishing currency of a cached object is the version stamp. If the version stamps of the objects are equal the objects should be equal, and version changes originating from a single client should be ordered.

Consider the situation of a crash while fetching data into the cache. The situation to avoid is to believe that the cache is up to date, while in fact the data was not fully fetched. To achieve this, the version stamp should not be written to persistent storage until it is certain that the fetched data has made it to the disk. On Linux this can be achieved by simply waiting 30 seconds before flushing the Fsdb database buffers. Upon reconnection to a server, all objects having an out of date version stamp should be flushed from the cache. If recovery is followed by disconnected operation, one could allow the possibly incompletely fetched objects to be used, for maximum availability. Such objects can be identified as those having a creation time less than 30 seconds since the last Fsdb flush. If such objects are modified and reintegrated, they'll be regarded as "suspect" and be handled as though they were conflicts.

If a client is modifying existing data when crashing, then the reverse risk exists: the client, when recovering, may see an old version stamp in the database, and possibly also an old modification time, the so called mtime, in the inode. If the object has just been fetched, the version stamp may even be older than that on the server. Yet the data on the client disk may be newer than that on the server. There are several approaches here, the simplest being "laissez faire" and accepting the fact that a few seconds of data modifications might be lost. This is not very desirable, since everything should be done not to lose the latest version of data.

Alternatively, the inode under modification could be updated with a new mtime and synced to disk. Provided modifications do not happen within the granularity of mtime units this allows the recovery mechanism to identify this file as suspect. The collection of suspect files is defined as those having an mtime later than the last sync time of the Fsdb. Such suspect files could be made subject to rsync style reintegration.

Yet another solution is to write the Fsdb database record to disk, before starting the modification, indicating that the inode has now changed. The latter involves a context switch to Lento, but since disk traffic is involved anyway, this is probably not a significant overhead [bottlenecks].

This detailed and slow initiation of modification procedure would not affect newly created objects.

InterMezzo aims to be usable for daemon controlled situations, for example to provide replication for WWW data. In such cases handling conflicting updates should not involve user interaction and we will introduce policies to handle conflicts automatically, for example by giving preference to the server copy and notifying an administrator of a conflict.

10 InterMezzo on Windows 9x, Windows 2000, and other Unix systems

We believe that the InterMezzo system can fairly easily be ported to Windows 9x and Windows NT/2000, building on the experience we gained when doing this for Coda [coda-win].

Both systems have the capability to do sophisticated filtering of filesystem requests and combine them with upcalls. Simple examples of such filter drivers are found in the FileMon utility [filemon]. For Windows 9x an additional difficulty arises from the non-reentrancy of the operating system. Lento will have to be a 32-bit DOS application, just like Coda's cache manager Venus. Fortunately, the Coda project provides the tools to make a port possible, and somewhat amazingly the Coda DOS cache manager works very well.

Porting InterMezzo to other Unix systems should be possible, but might require knowledge only obtainable through kernel source.

11 Conclusion

With 2,500 lines of C kernel code and 5,000 lines of Perl code (don't worry, no one liners here) we have managed to implement the basic functionality of the InterMezzo filesystem.

This is very encouraging and we hope to continue towards a fully functional robust filesystem.

12 Availability

The cache manager and file server code, as well as Linux kernel code, are free software, released under the terms and conditions of the GNU General Public License [gpl]. InterMezzo is available via anonymous FTP from `ftp://ftp.inter-mezzo.org/pub/intermezzo`.

Furthermore, the InterMezzo home page contains papers, instructions, documentation, and updates. It can be found at `http://www.inter-mezzo.org`.

References

[ace] `http://siesta.cs.wustl.edu/~schmidt/ACE.html`

[bottlenecks] Peter J. Braam & Philip Nelson, *Removing Bottlenecks in Distributed FIlesystems: Coda and InterMezzo*, Linux Expo 1999

[coda] `http://www.coda.cs.cmu.edu`

[coda-win] `http://www.usenix.org/events/usenix99/braam.html` [dfs-protocols] Peter J. Braam, *Distributed Filesystems for Clusters: a Protocol Perspective*, Usenix Technical Conference, Extreme Linux Track, 1999

See also: `http://www.extremelinux.org/activities/usenix99/docs`

[erlang] `http://www.erlang.org`

[ficus] `http://ficus-www.cs.ucla.edu/travler`

[filemon] `http://www.sysinternals.com`

[gpl] `http://www.gnu.org`

[poe] `http://www.netrus.net/users/troc/perl.html`

[stackable] John Shelby Heidemann, *Stackable Design of Filesystems*, Technical Report CSD-950032, University of California, Los Angeles, September, 1995

See also: `http://www.cs.columbia.edu/~ezk/research`

[zadok] Erez Zadok and Ion Badulescu, *A Stackabe File System Interface For Linux*, In Proceedings of Linux Expo '99, May, 1999

See also: `http://www.cs.columbia.edu/~ezk/research`

[teapot] Satis Chandra, Brad Richards, and James R. Larus, *Teapot: Language Support for Writing Memory Coherance Protocols*, In Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI), May, 1996

[xfs] `http://now.cs.berkeley.edu`

Linux on PA RISC
The Final Frontier

Alex deVries
adevries@thepuffingroup.com

Abstract

Linux has been ported to a wide variety of CPUs and architectures, and the last CPU with a significant install base without Linux support is Hewlett Packard's PA RISC series.

The PA RISC is a RISC processor with a virtual address cache, nested interruptions, shadow registers, and segments.

There's a significant amount of this hardware in existence, since the product line extends back to 1986. There are 135 different machines, with a combination of about 7 different system busses, 3 of which are proprietary.

One thing that's simplifying the development is that HP machines have Processor Dependant Code (PDC) on all of the machines, which offers a common interface to the hardware regardless of which machine. This means that console access is trivial, which is far simpler than other ports such as the Apple Macintosh.

There are some existing open source projects to draw resources from: the partially complete mkLinux port and OpenBSD. The native unix for PA RISC is HPUX.

There are two different CPUs that will be supported. The PA RISC 1.1 is a 32 bit CPU, and is found on machines manufactured in the 1990s, primarily in uniprocessor machines. These are widely available, and therefore will be the first CPUs supported.

Later developments will include support for the PA RISC 2.0 CPUs, which are 64 bit processors. Multiprocessor support for both PA RISC 1.1 and 2.0 CPUs is expected.

There are compatibility benefits in supporting the same executable loading format as the host vendor. However, HP used the Spectrum Object Model (SOM) for their 32 bit versions of HPUX, which is not used in any other operating system. In 64 bit HPUX, there is support for 64 bit ELF. Part of the project involves creating the first 32 bit version of ELF on PA RISC, for which there is an unimplemented but released specification.

Development supported by HP was started in March of 1999.

By May of 1999, developers have cleaned up the existing 32 bit PA RISC development tools, binutils and egcs. Portions of the kernel have been completed, the bootloader, memory setup and interruptions are operational. Device drivers for SCSI, ethernet and PCI are not yet complete. Following their completion, glibc and other userland applications will be ported.

1 Introduction

Like many Linux development projects, this one started out being done only for the hack value. Over time, the port was taken considerably more seriously.

The idea for a monolithic kernel on PA RISC came up in October of 1998 at Atlanta Linux Showcase. A group of developers were having lunch in a food court, and the topic of the different architectures came up. So the list flowed... i386, M68k, PowerPC, MIPS, Sparc, DEC Alpha and a few others. It was soon clear that it would be easier to talk about the architectures that weren't yet supported. It was rumoured that the VAX port had started, and the ones that remained were SH3, M88k and PA RISC.

And so the port started. The understanding was that there were plenty of PA RISC machines sitting idle, that they'd all be similiar, that we'd find some documentation treasure if we just digged in the right places, that we could reference the mkLinux source at worst. And hey, we had gcc and binutils for HPUX already, right?

There were a few more challenges in getting started than expected, but recent developments have been going extremely quickly.

1.1 Working with Hewlett Packard to Choose a PA RISC Model to Target

In February of 1999, Hewlett Packard sponsored the port by talking to the company organizing the port, The Puffin Group. Their hope was to have Linux running on modern hardware that was actually shipping.

The developers were looking for a machine which had the following characteristics:

- low cost equipment is ideal for distributing hardware between developers and from hardware manufacturers to developers

- an architecturally simple machine will kick start the initial development

- having a machine that has peripheral control components for which there are already drivers helps considerably

- a machine that already existed in the market, but which would be easy to obtain second hand

- a machine for which the manufacturer felt comfortable in releasing hardware documentation

Together it was agreed upon which machines should be the focus of the port, and what documentation had to be made publicly available. Many of the hardware documents had to be re-edited for public consumption.

HP shipped twelve developers hardware and manuals, as well as providing server space and technical advice.

The HP PA RISC line goes back to 1986, so there are quite a few architectural changes between those machines.

To start, Hewlett Packard has released documentation for just one machine, the A class 180C. This has a single PA RISC 1.1 32 bit CPU with a custom PCI controller chip (Dino), custom I/O controller (Lasi), something called a GSC bus, and no graphics. This is an ideal machine because it presents the simplest possible case of all the modern PA RISC machines that HP is shipping.

The A180 is somewhat related to other machines. The 700 series, also called Gecko, features a fully connected Lasi controller. Development for the A180 will lead to many of the features of the Lasi being supported, so 700 support will be included quickly. There are many 712s available cheaply for PA RISC Linux developers.

2 Porting the Kernel

2.1 PA RISC CPUs

The PA RISC is a RISC CPU that can run in either byte order, although for Linux it will be run in big endian mode for compatibility with HPUX.

Version 1.1 processors have 32 bit word sizes, version 2.0 processors are 64 bits. Both have a fixed instruction size of 32 bits, and different models within the 1.1 and 2.0 versions have a different number of pipelines.

There are 32 general purpose registers, 32 control registers, 32 floating point registers, and numerous state and control registers.

I/O is mapped into a range of memory that is not cached.

There is support in hardware for multiple processors. The architecture provides support for a single coherent TLB and cache among CPUs.

2.2 Build Tools

Building a cross compiling toolset is usually the first place to start in a new architecture's kernel port. The start of this was inheriting the tools used by mkLinux, and patching it slightly to include aliases for 'parisc-linux' instead of 'hppa-linux'. It was decided to simply adopt their executable object format, a self-concocted ELF.

A bit further into the development (particularly in linking), there were large problems encountered in building test kernels. This was long before the kernel was actually booting, even linking a dummy kernel failed. Upon further investigation it was discovered that there were serious problems with both the ELF specification used and the linking tools.

Through some discussions with Jeff Law and others who had worked extensively with the tools in question, it was determined that the existing PA RISC ELF implementation was quite broken. In the end we were targetting HPUX binary compatibility, so choosing the proper executable format was critical.

HPUX has used two different object formats; the first is the Spectrum Object Model (SOM) (named after the original product name of the CPU line), the next is ELF64, which is used on 64 bit implementations of HPUX 11 and later.

Hewlett Packard had actually released specifications for their 32 bit version of ELF, which is very similiar to ELF64.

While this was being figured out, HP announced that they'd sponsored Cygnus to do the ELF64 port for gcc and binutils. This is a very large development task, so the open source developers decided to leave ELF32 alone until there was an ELF64 implementation available.

So, the only functional available object format was SOM. However, the complex relocations involved in a SOM linker were never implemented in binutils' ld. That left the only available linker as HPUX's ld, and all building had to be done on HPUX. This has slowed development down considerably, and is unlikely to change until proper ELF32 implementations can be derived from Cygnus' ELF64 work.

Having ELF32 functional is critical for userland application building, because of the reliance on ELF in glibc. Fixing glibc to use SOM is a dead end; the future is in ELF64 anyway for 64 bit systems.

2.3 Booting

There's been significant source code examples from which to draw the boot code. Jason Eckhardt built the first thing that could actually boot and print 'Linux' on the screen. It was a seperate application that could be booted either by writing to a bootable tape or disk, bootp or through HP's custom network booting protocol, rbootd.

Over time this code was merged into the kernel to produce an Image that contained the kernel itself, and actually be able to boot and load an appended kernel image.

The boot header used is actually quite simple; there are pointers at certain locations that explain to the bootrom where to start loading the image. Within that, there's code embedded into the boot image to actually load the kernel, which is at the end of the boot image.

The intention is that eventually the boot code will be copied into a different executable that can load kernels that are built into an ext2 or UFS filesystem. This bootloader will be called HILO.

2.4 The Joy That Is PDC

Perhaps the easiest thing about Linux for PA RISC is the ability to interface with devices before the operating system loads. All PA RISC 1.1 machines are shipped with a common interface to this library of routines that is built into the hardware. PDC works by initializing all the peripheral components', and setting up a simple interface to each of them. The routine handle everything from a basic terminal interface to interval timers to reading data off of a disk.

So the bootloader itself is actually a collection of PDC calls, and is able to put output on the console immediately.

Helge Deller fixed the kernel to the point of being able to load the first kernel, and in June of 1999, the first kernel booted, and promptly died. There was international celebration.

The Linux kernel ports for many operating systems have the problem at the start of development of not being able to provide feedback to report how far the booting process got. PDC ensures that we're all able to display debug messages from the very start.

2.5 Interruptions

Interruptions are events that interrupt the normal flow of instruction execution. These are caused by faults, traps, hardware interrupts and checks.

Upon receiving an interruption, the CPU copies certain general purpose registers into a set of shadow registers. The are copied back at the end of an interruption service routine.

All hardware interrupts result in the same interruption being called, so the hardware calling the interrupt leaves a certain number in one of the control registers on the CPU (IO EIR), the interruption service routine determines which device the interruption comes from and passes it on to the lower half of the Linux kernel's interrupt handling.

2.6 Memory Management

The PA RISC CPUs differ slightly in how they manage virtual memory.

Access permissions and mapping between virtual and physical memory is provided in hardware on the CPU by Translation Lookaside Buffers, TLBs. User processes and the kernel run within one or more spaces. The memory addresses in processes are offsets from the start of the space.

By default, pages are 4kB large, and the TLB has room for 120 slots. Being able to access more than 480kB requires management of the TLB, and in this case the use of block TLBs.

Some models of PA RISC CPUs do TLB miss handling using a hash algorithm implemented in hardware. One current TLB miss handler idea is to simply disable that feature and do the miss handling in software. This would allow common memory management code for the different PA RISC processors.

The actual details of the different PA RISC processors haven't been released by HP as of this writing, so details of the implementation have yet to be decided.

2.7 Dino - Talking to Dinosaurs

Dino is the PCI controller on the A180, and several other PA RISC machines with both a GSC and a PCI bus. It acts as either a bridge to a PCI bus that's physically on the motherboard, or in card mode on a GSC expansion card with a local PCI bus. This is the arrangement that's found on GSC expansion cards with multiple ethernet controllers.

Dino has an RS232 port on it that is unused on the A class, but is intended for keyboard support on machines with no other I/O controller on board.

Dino's base physical address and interrupts can be mapped into any range that's available. There is code to negotiate the allocation of these resources.

Most of the Dino controller has been written, but it remains untested because of the lack of interrupt support in the kernel.

2.8 Lasi

Lasi is the I/O controller that's used on older machines with an internal GSC bus, such as the 700 series and a plethora of newer machines.

The chip is really an aggregation of existing hardware all put onto one chip, with a bit of GSC glue. The fact that these components all exist in other architectures will help considerably with Linux support.

It is possible there will be some issues with cache coherency for devices that DMA data directly into memory.

The devices onboard are:

- SCSI, an NCR53c710 - LAN, an 82596 - parallel port, a WD16C522 - serial port, an NS 16550A with minor changes - keyboard and mouse port - real time clock - flash EPROM - power control (custom) - PC floppy driver controller, a WD376C65C - telephone and audio controller (custom)

The Lasi glue that holds them together is essentially just a map of the components' registers to the GSC bus.

For machines with several ethernet or SCSI controllers, there's a need to have multiple LASI controller chips. Lasi's architecture allows room for four controllers total.

Not all machines that contain Lasi have all of the devices connected. On the A180, for instance, the ethernet, audio, floppy, mouse and keyboard controllers are simply not connected. Different machines will use different amounts of Lasi.

The reduction of so many components into one IC has resulted in very small layouts for machines. The 712 motherboard layout is essentially a PA RISC CPU, Lasi, some glue logic and connectors.

3 The Current Status

As of mid-July, 1999, the kernel booted, handled interruptions, and actually reported sane BogoMIPS values. Major portions of Lasi and Dino device drivers were written, but not yet functional. The kernel was reported to actually boot on more than a dozen different machines. Developers had started working on memory management.

4 Future Challenges

The PA RISC 2.0 series procesors will offer several new features. The first is the that this is a 64 bit CPU; the existing 32 bit code base needs to work with this word size.

The intention is to initially run the PA RISC 2.0 CPU in a 32 bit mode.

Work will have to be done on writing device drivers for peripheral controllers that exist in other machines. This will depend on HP's release of other documentation.

5 Acknowledgements

The PA RISC Linux project would have been difficult to make possible without the contributions of Philipp Rumpf, Helge Deller and other developers' contributions. Hewlett Packard has greatly accelerated the project by offering assitance, usually on the developers' terms.

6 References

`http://www.thepuffingroup.com/parisc`

"Will Code For Hardware" – Supporting the Unsupported and Getting Cool Toys

Nathan I. Laredo
Stradis, Inc.
3170 Sugarplum Road, NE
Atlanta, Georgia, 30345-1553 USA
laredo@gnu.org, http://mpeg.openprojects.net

Abstract

This paper discusses the development of the Stradis MPEG2 hardware decoder driver for Linux. It addresses development process, driver architechture, and problems encountered even with the full technical support of Stradis and full documentation on all of the chips used on the board. It will also cover some techniques used to protect intellectual property of the manufacturer as well as well as covering the optimization of the driver for real use after the initial just-get- it-working hack. The Video4Linux API was used for the driver, and an overview of the extensions to the API will also be presented.

1 Introduction

I was first contacted by Doug Ledford who was swamped with far more projects than the average human is capable of supporting in early 1999 about some new toys he had been given in exchange for writing a Linux driver. It was a good deal for Stradis. The Chief Technical Officer at Stradis, Chris Hodges, mailed Doug a package with two MPEG-2 decoder boards, the Philips SAA7146 PCI Bridge Software API Description (which turned out to be only useful to understand the Windows driver), the Philips SAA7146 Data Sheet (a book actually), a CD-R with some MPEG test sequences, the Windows drivers and software, and the actual source code to the Windows playback application. I thought to myself, "This should be a piece of cake," so I told Doug that I would be happy to take over this project he would never have time for. I received a phone call from Stradis just as I was crawling into bed. They decided they wanted Linux support because they had a customer that was asking for it, and I learned that they have no great love for Microsoft either. The driver was to be written in exchange for two $1000 value 50Mbps MPEG2 decoder boards.

I received a package from Red Hat with the hardware just before I started packing for my move to Atlanta after an unusual four years in the Air Force. I thumbed through the documentation, installed the hardware, and tried the Windows driver. It failed. Since Stradis was in Atlanta, I decided to just wait until I got there to get it working.

I moved into my new apartment March 1st and spent two weeks getting 30 boxes of junk sorted out and new furniture purchased. I met with the CEO and CTO of Stradis two weeks later. After sticking an oscilloscope on the card, we discovered that all of my problems were caused by my PCI bus running at 37MHz, an unfortunate consequence of my I-have-a-faster-machine-than-you-have ego trip. The Windows driver has since been hacked to work with the faster speed. I honestly had zero code written at this point, but I was able to pick up a manual for the MPEG decoder chip on the board.

2 The Blank Page

Every writer hates to stare at a blank page, especially when writing a paper for a conference with the deadline the same day. I had zero code, and the project seemed a more daunting task than I thought I could handle. Using bttv.c as a reference, I hacked out a skeleton of a driver that did nothing but printk for everything. I followed the Windows driver initialization code and poked that into the skeleton after I borrowed the pci device detection code from the bttv driver.

I continued hacking in code, replacing every printk I had with code to actually do something. I quickly progressed from no code to something that closely resembled a driver that could very well work, but I had no way to test it fully without finishing more of it. From a feeling that I would never finish, I reached a point where I thought that I was almost finished with the driver, but I soon discovered that nothing that I had written actually did what it was supposed to do.

3 Bugs Are Everywhere, Even in Data Books

After about five days of debugger's block, my brain was about to explode. This was defused by an offer from Chris Hodges to stick the old oscilloscope on the card to see what it was actually doing. The first thing that I learned is that Windows apparently has a concept of breakpoints and watches everywhere, and I only had printk and learned to return -1 on module init to break out, with usual precautions against leaking kernel memory.

In theory, everything in my driver should have been working perfectly. In reality, I produced nothing. The card was nicely designed with a LED that would turn off as soon as the FPGA was properly programmed. Of course in the end I discovered that it was not the FPGA loader code that was wrong, it was the code that wrote to the 16-bit interface of the SAA7146.

Philips, in their infinite wisdom, designed a chip that would raise an IRQ when a transfer was "finished" before the transfer was complete. Not only that, but I was guilty of treating a 32-bit pointer as if it were an 8-bit pointer, so between my crashing the kernel all day and the Philips chip not working as the manual describes, it was a wonder that I made any progress at all.

Since there was a direct path from the IBM MPEG decoder chip through the FPGA to the Philips SAA7121 video encoder chip, and the FPGA was not required to access the I2C bus, I decided the next order of business was to get the SAA7121 programmed properly for output. The code was already written but like the FPGA code, it did not work either. The SAA7146 has a smart I2C interface and a really annoyingly wrong manual. After much oscilloscope poking on the card, we discovered that it also suffered from the same "I'm done now but not really" problem as the other parts. It also suffered from a severe case of poor examples in the manual.

4 It Works! It Really Works!

The way the Stradis card is designed, the FPGA must be programmed properly before anything else can be tested, so once we got the FPGA programmed, I fleshed out the code to program the microcode into the IBM MPEG decoder chip. I had already written it, but never actually called the code since the FPGA was not working before. I was still on my second visit to Stradis at this point and felt motivated to pull an all-night hack session after seeming to get the IBM chip programmed on the first try. I am not quite sure why the hours between 1AM and 4AM are so productive, but I am quite thankful that they exist.

The IBM chip seemed to get programmed on the first try. I was lucky, but during my all night hack session I got nothing more interesting than colorbars produced directly by the SAA7121 video encoder. The IBM chip would produce output that resembled feeding random data to it. At this point, everything was still being done by polling. No IRQ routines were yet written. This was day two at Stradis. Around lunchtime, I realized that I was walking all over my DMA buffer. I was filling it with new data while the previous block transfer was in progress.

After lunch, I wrote the code to poll for block transfer complete before loading the new data into the buffer, and as if by magic, I saw beautiful full-frame-rate NTSC output from the elementary MPEG2 streams. No audio yet, but beautiful video. Audio would prove to be a thorn in my side, and in the end a trivial bug, but that is a story for later.

5 Total Time Elapsed: 5 days. No Interruptions

The people at Stradis were impressed. I found myself with a job offer. I was also given an opportunity to go to Las Vegas to the National Association of Broadcasters 1999 convention to demonstrate the card running in Linux. Needless to say, I accepted the free trip. The show was mid-April, and between the design of the new revision of the decoder board and preparing to ship everything to the show and setup the booth, everyone at Stradis was quite busy.

At this point, the driver was working for MPEG video and still using 99% of the CPU because it used no interrupts. I hacked out some code to split a transport stream into audio and video streams since the IBM chip required them to be written to different ports. The driver needed to be moved to an interrupt-based architecture as well to support the audio properly. The IBM chip has a very small audio buffer and a large video buffer. Threshold interrupts could be setup to tell the driver what buffer was starving for data.

Interrupts were a bit more complicated than I first imagined. Thanks to Alan Cox, I learned the virtues of the wait queue. Without `wake_up_interruptible()` and `interruptible_sleep_on()`, I would have never figured out how to structure the driver. The driver also needed a true ring buffer, but that would have to wait until after NAB99 which was quickly approaching.

Just before the show I got the interrupt-based driver working perfectly on my dual PII400 machine. It was working on less than 1% cpu, and I solved that elusive audio bug. Apparently in writing the audio microcode I was trying to do 16-bit writes when the chip could only take an 8-bit write so half of the microcode was being lost to the great bit bucket in the sky.

Now I could take it to Las Vegas and show of a driver that uses next to zero cpu and does both video and sound, and in the meantime I hacked together the extra few lines of code it took to do a fixed-size VGA video overlay to framebuffer memory. Little did I realize how absolutely horrid the performance of the Compaq Presario that was used for the show was. On the NAB99 show floor I was hacking the driver back to do no interrupts and no audio only minutes before the show was to open.

6 After the First 99% of Development Comes the Next 99%

Overall the major mysterious bugs in the driver were attributable to pointer math being done behind my back. Treating a 32-bit pointer as an 8-bit pointer is not wise, especially if you must wait through fsck on a Compaq Presario with a Quantum Bigfoot drive. The ring buffer code was no exception. While in North Carolina in May for Linux Expo, I was sitting at Red Hat trying to figure out what was causing the random noise every time the buffer wrapped around. I banged my head against the desk, played five levels of Ms. Pac Man, drank about twelve cans of orange juice, only to find another elusive pointer error.

Dr. Michael Fulbright at Red Hat was nice enough to let me demo my driver in the RHAD Labs booth at Linux Expo. This demo was complete with audio and video and starwars MPEG trailer. By this point my machine had become a dual Pentium III 550, so I knew that the real question would not be answered until I got home from the conference and tried the driver on the piece of work Compaq Presario. Indeed I had a good solid driver that even stood up as the Compaq began to swap.

Now the question came up from Stradis, "This is nice that it all works, but you're still using the console to do everything. We think it should have some graphical user interface like the Windows version with VCR-like buttons." But, luckily I'd just listened to Miguel and Federico and decided that GNOME was the way to go. So far, this part has not materialized, but some day it will. I like to keep my hands in the kernel if possible, and the original agreement was to write the driver in exchange for two cards, and it mentioned nothing of a user interface.

The driver also needed statistical collection and frame and timecode tracking, as well as other functions not specifically required for the actual playback of the MPEG files, but instead for the applications that the board was actually intended for when installed in a professional environment.

7 Can I Buy You a Thought?

I knew it would happen from the start. Stradis came back to me and said, "You know, it would probably be better if we didn't provide our competition with everything that they needed to reverse engineer our board and produce a cheap knockoff by taking advantage of all the time and effort that we've invested in the board." Ok. This is fair enough. "There's a lot of effort put into the FPGA code for example, and withholding that would make it next to impossible to do anything with the driver without obtaining a proper FPGA licence." Enter the next two weeks I spent moving all the intitialization code around so that the FPGA and IBM microcode could all be loaded runtime instead of included in the kernel itself.

Yet another chance to crash my machine. In retrospect, it may have been wise for me to setup a dedicated NFS server and use NFS root for kernel development so that long delays from crashes would be non-existant, but I have limited means at home. The new microcode loader also allows a user to change to new FPGA code without ever recompiling the driver again. This could prove useful in a world of people who would not want to hack kernels. The board is soft enough to allow one pixel wide timing changes just by updating the FPGA code.

So what do we gain by allowing the firmware to be uploaded instead of compiling in the kernel? It means that some Asian knock-off company will have a harder time reverse engineering the board without the FPGA bitfile which not only lets the Stradis board be more flexible than any other board, but it also was a considerable investment in hardware development time. Giving anyone this firmware allows them to potentially copy the Stradis card and mass produce it in such a way that Stradis could not compete as a small startup company and would fail. This eases the fears of Stradis relating to the idea of an open source driver.

8 The Alan Cox Seal of Approval

After all is said and done, I still had to get the elusive Alan Cox Seal of Approval. I sent Alan a copy of the API changes, and naturally he sent me back a one line cryptic reply as if I were some coding machine. Sometime around Linux Expo, I lost all of Alan's comments on the proposed changes to the API. Yes, there were other comments outside of email longer than one line.

Once again, I sent an email with the proposed API (which just happened to exactly match the one I had already implemented), and he obligingly sent me back yet another suggestion. After a short chat, everyone seemed to be happy and Alan hinted that we could see the new stuff in the 2.2.11 kernel. I had earlier met Linus when he did a keynote at a conference in Atlanta and found out that there would be no problems getting the driver itself into the 2.2 tree.

9 The Extended Video4Linux API

9.1 VIDIOCSPLAYMODE:

This set the output video mode (PAL/NTSC/SECAM) As well as special features such as GENLOCK, Pause, Fast Forward, Single Frame Step, Slow Motion, Normal Play, and Non-Interlace Mode, Black Frame, and Colorbar output.

9.2 VIDIOCSWRITEMODE:

This tells the driver what type of data to expect on write(). The user-mode code could write compressed MPEG audio, MPEG video, an on-screen-display overlay bitmap (alpha blending done in hardware by the IBM MPEG decoder, written in native format), Closed Captioning or Teletext bytes to overlay using the SAA7121 video output, or MJPEG (which was for other devices, not the Stradis decoder)

9.3 VIDIOCGPLAYINFO:

This returns information about the current stream being played. Info returned includes frames output since decode began, the horizontal and vertical unscaled sizes, current SMPTE timecode the current picture type, current temporal reference, and the last user data found in the compressed stream.

9.4 VIDIOCSFIRMWARE:

This was added to load new FPGA bitfiles and IBM Decoder microcode once the device driver is loaded. A driver can use the ”loadwhat” field to address the given code to a specific area of a card.

10 Usage

10.1 The Simple Case

```
cat susi_080.m2v >/dev/video0
```

for sending a video elementary stream to the decoder card for playback without video framebuffer overlay.

10.2 The Most Complicated Case

```
parsempeg french.mpeg 224 192
```

for sending a transport stream with video on PID 224 and audio on PID 192 for playback without the video framebuffer overlay.

MPEG playback at its finest. Multiple opens were hacked in around an upper level Video4Linux restriction in order to allow a user to use separate processes for sending the MPEG file to the card for decoding and doing other controls with the card. This way a Video4Linux-compliant program such as tvset could be used for the VGA overlay while cat or a more complicated MPEG parser is used to actually send the MPEG sequence to the card for decoding.

Still remaining to implement is the MMAP capture interface of Video4Linux and a poll routine to allow the user to use select to synchronize events on the pending output decoded video frame.

Alan also suggested that I use spinlocks. My SMP system thanked me immediately when I made that change.

11 Conclusion

At the start, I began to doubt that I would ever be able to get the hardware to do anything and that

Stradis would ask for the cards back in frustration. The moral of the story is always follow pointers and never forget to check your references, and if you ever offer to write a device driver in exchange for free hardware, even the coolest hardware in the world, you are grossly underestimating your own worth. The mythical man month really was right after all.

12 Acknowledgments

Special thanks go to Alan Cox for his sage advice, and to the countless people who helped edit this paper.

13 Availability

The Stradis 4:2:2 Hardware MPEG-2 Decoder driver and the development diary is at

`http://mpeg.openprojects.net/`

Linux Your Game With Simple DirectMedia Layer

Sam Lantinga
Loki Entertainment Software
slouken@devolution.com

1 Table of Contents

1. Author Introduction
Sam Lantinga, Lead Programmer for Loki Entertainment Software. Loki Entertainment is a company devoted to porting top games to Linux. You can find out more at `http://www.lokigames.com/`

2. Simple DirectMedia Layer

1. What is it?
   - SDL is a free cross-platform multi-media development API
   - Used for games
   - Used for game SDKs
   - Used for emulators
   - Used for demos
   - Used for multimedia applications
2. What can it do?
   - Video
   - Events
   - Audio
   - CD-ROM audio
   - Threads
   - Timers
   - Endian independence
3. What platforms does it run on?
   - Linux, of course
   - Win32
   - BeOS
   - Unofficial ports, ports in progress

3. Using the Simple DirectMedia Layer API

1. Initializing the library
2. Video
   - Choosing and setting video modes (the easy way)
   - Drawing pixels on the screen
   - Loading and displaying images
3. Events
   - Waiting for events
   - Polling for events
   - Polling event state
4. Sound
   - Opening the audio device
   - Loading and playing sounds
5. CD-ROM audio
   - Opening a CD-ROM drive for use
   - Playing the CD-ROM
6. Threads
   - Create a simple thread
   - Synchronizing access to a resource
7. Timers
   - Get the current time, in milliseconds
   - Wait a specified number of milliseconds
8. Endian independence
   - Determine the endianness of the current system
   - Swap data on systems of differing endianness

4. Porting games to Linux, using DirectX sample "donuts" as example:

1. Compile the source code
   - Create a Makefile
   - Samples of original Win32 source
   - Samples of modified Linux source
2. Identify major subsystems
   - Video and Sound
3. Rewrite major components
   - Samples of original Win32 source
   - Samples of modified Linux source
4. Polish the code
   - Remove unnecessary functions
   - Remove menus and options
   - Remove extra C preprocessor defines
5. Add Linux specific features
   - Add icon and title caption
6. Optimize and debug
   - Compare DirectX flip and fillrect with X11 primitives

Appendix A:

Sample routines for playing nicely with X windows from within SDL

2 Author Introduction

Author Introduction:

Sam Lantinga, Lead Programmer for Loki Entertainment Software. Loki Entertainment is a company devoted to porting top games to Linux. You can find out more at `http://www.lokigames.com/`

Background:

- Author of the Simple DirectMedia Layer (SDL)
- Ported Maelstrom from the Macintosh to Linux
- Worked on port of the Macintosh emulator Executor to Win32
- Ported some of the DOOM tools from DOS (DEU, DHE, etc.)
- Other projects: `http://www.devolution.com/~slouken/`

Contact:

- `slouken@devolution.com`

3 Simple DirectMediaLayer

`http://www.devolution.com/~slouken/SDL/`

1. What is it?
   - SDL is a free cross-platform multi-media development API
   - Used for games:
     - Descent 1X, DOOM, Hexen, Maelstrom `http://www.devolution.com/~slouken/SDL/download.html`
     - Hopkins F.B.I. `http://hopkinsfbi.com/`
     - Civilization: Call To Power `http://www.lokigames.com/products/civ/civ.html`
     - Myth II: Soulblighter `http://www.lokigames.com/products/mythII/myth.html`
     - Railroad Tycoon II `http://www.lokigames.com/products/railroad/railroad.html`
     - Bumpracer `http://members.linuxstart.com/~karlb/bumpracer.html`
     - Jump! `http://members.xoom.com/pesterhazy/`

  - Billy The Kid
    `http://www.kokkeskolen.dk/billythekid.exe`
  - UClient
    `http://www.worldforge.org/website/client/uclient/`
  - LateKnights
    `http://lateknights.penguinpowered.com/`
  - sdlbomberman
    `http://www.xdr.com/dash/linux.html`
- Used for game SDKs:
  - PowerPak
    `http://www.angelfire.com/va/powerpakgsdk/`
  - LibU
    `http://www.rhrk.uni-kl.de/k̃laux/libu.html`
  - SparkGL
    `http://SparkGL.netpedia.net/`
- Used for emulators:
  - Executor, a Macintosh emulator
    `http://www.ardi.com/`
  - DGen/SDL, a Sega Genesis emulator
    `http://www.humboldt1.com/õgnir/dgen-sdl.html`
  - jzIntv, an Intellivision emulator
    `http://www.primenet.com/ĩm14u2c/intv/`
- Used for demos:
  - HElliZER, qrash
    `http://www.devolution.com/s̃louken/SDL/download.html`
  - loop
    `http://www.csrmusic.org/loop/`
- Used for multimedia applications:
  - MpegTV, an MPEG player
    `http://www.mpegtv.com/`
  - SMPEG, an MPEG player
    `http://www.lokigames.com/opensource/opensource.html`

2. What can it do?

- Video
  - Set a video mode at any depth (8-bpp or greater) with optional conversion, if the video mode is not supported by the hardware.
  - Write directly to a linear graphics framebuffer.
  - Create surfaces with colorkey or alpha blending attributes.
  - Surface blits are automatically converted to the target format using optimized blitters and are hardware accelerated, when possible. MMX optimized blits are available for the x86.
  - Hardware accelerated blit and fill operations are used if supported by the hardware.

> **Tip:**
> You can set your application's title-bar (if any) and icon using the `SDL_WM_SetCaption()` and `SDL_WM_SetIcon()` functions respectively.

- Events
  - Events provided for:
    * Application visibility changes
    * Keyboard input
    * Mouse input
    * User-requested quit
  - Each event can be enabled or disabled with `SDL_EventState()` .
  - Events are passed through a user-specified filter function before being posted to the internal event queue.
  - Thread-safe event queue.

> **Tip:**
> Use `SDL_PeepEvents()` to search for an event of a particular type in the event queue.

- Audio
  - Set audio playback of 8-bit and 16-bit audio, mono or stereo, with optional conversion if the format is not supported by the hardware.
  - Audio runs independently in a separate thread, filled via a user callback mechanism.
  - Designed for custom software audio mixers, but the example archive contains a complete audio/music output library.

> **Tip:**
> Use the `SDL_LockAudio()` and `SDL_UnlockAudio()` functions to synchronize access data shared by the audio callback and the rest of the program.

- CD-ROM audio
  - Complete CD audio control API

Tip:
If you pass a NULL CD-ROM handle to the CD-ROM API functions, they will act on the last CD-ROM that was opened.

- Threads
  - Simple thread creation API
  - Simple binary semaphores for synchronization

Tip:
Do not use C library functions like I/O and memory management from threads if you can help it - they lock resources used by other threads.

- Timers
  - Get the number of milliseconds elapsed
  - Wait a specified number of milliseconds
  - Set a single periodic timer with 10ms resolution

Tip:
You can easily replace the Win32 `GetTickCount()` with `SDL_GetTicks()`

- Endian independence
  - Detect the endianness of the current system
  - Routines for fast swapping of data values
  - Read and write data of a specified endianness

Tip:
When reading your data files, you may need to byteswap 16-bit graphics

3. What platforms does it run on?

- Linux, of course
  - Uses X11 for video display, taking advantage of XFree86 DGA extensions and new MTRR acceleration for fullscreen display.
  - Uses the OSS API for sound.
  - Threads are implemented using either the `clone()` system call and SysV IPC, or glibc-2.1 pthreads.

Tip:
You can get at the hidden portions of the SDL driver interface via the `SDL_GetWMInfo()` function. This allows you to do things like remove window decorations and programmatically iconify your window.

- Win32
  - Two versions, one safe for all systems based on Win32 APIs, and one with higher performance, based on DirectX APIs.
  - Safe version uses GDI for video display. High performance version uses DirectDraw for video display, taking advantage of hardware acceleration if available.
  - Safe version uses waveOut APIs for sound. High performance version uses DirectSound for audio playback.

Tip:
You must call the SDL event functions periodically from your main thread to pump the Windows message queue and keep your application responsive.

- BeOS
  - BWindow is used for video display.
  - BSoundPlayer API is used for sound.

Tip:
Linux and BeOS support the SDL_INIT_EVENTTHREAD flag which, when passed to `SDL_Init()`, asks the event loop to run asynchronously in another thread. This is useful for color cursors that respond even when the application is busy.

- Unofficial ports, ports in progress
  - Solaris, IRIX, FreeBSD
  - MacOS

4 Using the Simple DirectMedia Layer API

1. Initializing the library

   Use `SDL_Init()` to dynamically load and initialize the library. This function takes a set of flags corresponding to the portions you want to activate:
   SDL_INIT_AUDIO
   SDL_INIT_VIDEO
   SDL_INIT_CDROM
   SDL_INIT_TIMER

   Use `SDL_Quit()` to clean up the library when you are done with it.

Example:

```
#include <stdlib.h>
#include "SDL.h"

main(int argc, char *argv[])
{
    if ( SDL_Init(
        SDL_INIT_AUDIO | SDL_INIT_VIDEO)
        < 0 ) {
        fprintf(stderr,
            "Unable to init SDL: %s\n",
            SDL_GetError());
        exit(1);
    }
    atexit(SDL_Quit);

    ...
}
```

> **Tip:**
> SDL dynamically loads the SDL library from the standard system library locations. Use the `SDL_SetLibraryPath()` function to use an alternate location for the dynamic libraries distributed with your application.

2. Video

- Choosing and setting video modes (the easy way)

  You just choose your favorite bit-depth and resolution, and set it!

  Example:

```
{ SDL_Surface *screen;

    screen = SDL_SetVideoMode(640, 480,
        16, SDL_SWSURFACE);
    if ( screen == NULL ) {
        fprintf(stderr,
            "Unable to set 640x480 "
            "video: %s\n",
            SDL_GetError());
        exit(1);
    }
}
```

> Tip #1: You can find the fastest video depth supported by the hardware with the function `SDL_GetVideoInfo()`.

> Tip #2: You can get a list of supported video resolutions at a particular bit-depth using the function `SDL_ListModes()`.

- Drawing pixels on the screen

  Drawing to the screen is done by writing directly to the graphics framebuffer, and calling the screen update function.

  Example:

```
Drawing a pixel on a screen of
arbitrary format

void DrawPixel(
    SDL_Surface *screen,
    Uint8 R,
    Uint8 G,
    Uint8 B)
{
    Uint32 color =
        SDL_MapRGB(screen->format,
            R, G, B);

    if ( SDL_MUSTLOCK(screen) ) {
        if ( SDL_LockSurface(screen)
            < 0 ) {
            return;
        }
    }

    switch (screen->format->
        BytesPerPixel) {

        /* Assuming 8-bpp */
        case 1: {
            Uint8 *bufp;

            bufp =
                (Uint8 *)screen->pixels
                + y*screen->pitch + x;
            *bufp = color;
        }
        break;

        /* Probably 15-bpp or 16-bpp */
        case 2: {
            Uint16 *bufp;

            bufp =
                (Uint16 *)screen->pixels
                + y*screen->pitch/2 + x;
            *bufp = color;
        }
        break;

        /* Slow 24-bpp mode,
           usually not used */
        case 3: {

            Uint8 *bufp;
```

```
            bufp =
                (Uint8 *)screen->pixels
                + y*screen->pitch + x;
            *(bufp+screen->format->
                 Rshift/8) = R;
            *(bufp+screen->format->
                 Gshift/8) = G;
            *(bufp+screen->format->
                 Bshift/8) = B;
        }
        break;

        case 4: { /* Probably 32-bpp */
            Uint32 *bufp;

            bufp = (Uint32 *)screen->pixels
                + y*screen->pitch/4 + x;
            *bufp = color;
        }
        break;
    }

    if ( SDL_MUSTLOCK(screen) ) {
        SDL_UnlockSurface(screen);
    }
    SDL_UpdateRect(screen, x, y, 1, 1);
}
```

Tip:
If you know that you will be doing a lot of drawing, it is best to lock the screen (if necessary) once before drawing, draw while keeping a list of areas that need to be updated, and unlock the screen again before updating the display.

- Loading and displaying images

  SDL provides a single image loading routine for your convenience, `SDL_LoadBMP()`. A library for image loading can be found in the SDL demos archive.

  You can display images by using `SDL_BlitSurface()` to blit them into the graphics framebuffer. `SDL_BlitSurface()` automatically clips the blit rectangle, which should be passed to `SDL_UpdateRects()` to update the portion of the screen which has changed.

  Example:

```
void ShowBMP(
    char *file,
    SDL_Surface *screen,
    int x,
    int y)
{
    SDL_Surface *image;
    SDL_Rect dest;

    /* Load the BMP file into a
       surface */
    image = SDL_LoadBMP(file);
    if ( image == NULL ) {
        fprintf(stderr, "Couldn't load "
            "%s: %s\n",
            file, SDL_GetError());
        return;
    }

    /* Blit onto the screen surface.
       The surfaces should not be locked
       at this point.
     */
    dest.x = x;
    dest.y = y;
    dest.w = image->w;
    dest.h = image->h;
    SDL_BlitSurface(image,
        NULL, screen, &dest);


    /* Update the changed
       portion of the screen */
    SDL_UpdateRects(screen,
        1, &dest);
}
```

Tip #1: If you are loading an image to be displayed many times, you can improve blitting speed by convert it to the format of the screen. The function `SDL_DisplayFormat()` does this conversion for you.

Tip #2: Many sprite images have a transparent background. You can enable transparent blits (colorkey blit) with the `SDL_SetColorKey()` function.

3. Events

- Waiting for events

  Wait for events using the `SDL_WaitEvent()` function.

  Example:

```
{
    SDL_Event event;
```

```
    SDL_WaitEvent(&event);

    switch (event.type) {
        case SDL_KEYDOWN:
            printf("The %s key "
              "was pressed!\n",
              SDL_GetKeyName(
               event.key.keysym.sym));
            break;
        case SDL_QUIT:
            exit(0);
    }
}
```

> **Tip:**
> SDL has international keyboard support, translating key events and placing the UNICODE equivalents into event.key.keysym.unicode. Since this has some processing overhead involved, it must be enabled using `SDL_EnableUNICODE()`.

- Polling for events

  Poll for events using the `SDL_PollEvent()` function.

  Example:

```
{
    SDL_Event event;

    while ( SDL_PollEvent(&event) ) {
        switch (event.type) {
            case SDL_MOUSEMOTION:
                printf("Mouse moved "
                    "by %d,%d "
                    "to (%d,%d)\n",
                    event.motion.xrel,
                    event.motion.yrel,
                    event.motion.x,
                    event.motion.y);
                break;

            case SDL_MOUSEBUTTONDOWN:
                printf("Mouse button "
                    "%d pressed at "
                    "(%d,%d)\n"),
                    event.button.button,
                    event.button.x,
                    event.button.y);
                break;
            case SDL_QUIT:
                exit(0);
        }
    }
}
```

> **Tip:**
> You can peek at events in the event queue without removing them by passing the `SDL_PEEKEVENT` action to `SDL_PeepEvents()`.

- Polling event state

  In addition to handling events directly, each type of event has a function which allows you to check the application event state.

  If you use this exclusively, you should ignore all events with the `SDL_EventState()` function, and call `SDL_PumpEvents()` periodically to update the application event state.

  Example:

```
{
    SDL_EventState(SDL_MOUSEMOTION,
        SDL_IGNORE);
}


void CheckMouseHover(void)
{
    int mouse_x, mouse_y;

    SDL_PumpEvents();

    SDL_GetMouseState(&mouse_x,
        &mouse_y);
    if ( (mouse_x < 32) &&
         (mouse_y < 32) ) {
        printf("Mouse in upper "
            "left hand corner!\n");
    }
}
```

> **Tip:**
> You can hide or show the system mouse cursor using `SDL_ShowCursor()`.

4. Sound

- Opening the audio device

  You need to have a callback function written which mixed your audio data and puts it in the audio stream. After that, choose your desired audio format and rate, and open the audio device.

The audio won't actually start playing until you call SDL_PauseAudio(0), allowing you to perform other audio initialization as needed before your callback function is run.
After you are done using the sound output, you should close it with the SDL_CloseAudio() function.

Example:

```
#include "SDL.h"
#include "SDL_audio.h"
{
    extern void mixaudio(void *unused,
                         Uint8 *stream,
                         int len);
    SDL_AudioSpec fmt;

    /* Set 16-bit stereo audio at 22Khz */
    fmt.freq = 22050;
    fmt.format = AUDIO_S16;
    fmt.channels = 2;
    /* A good value for games */
    fmt.samples = 512;
    fmt.callback = mixaudio;
    fmt.userdata = NULL;


    /* Open the audio device and
       start playing sound! */
    if ( SDL_OpenAudio(&fmt, NULL) < 0 ) {
        fprintf(stderr, "Unable to open "
            "audio: %s\n", SDL_GetError());
        exit(1);
    }
    SDL_PauseAudio(0);

    ...

    SDL_CloseAudio();
}
```

> **Tip:**
> If your application can handle different audio formats, pass a second SDL_AudioSpec pointer to SDL_OpenAudio() to get the actual hardware audio format. If you leave the second pointer NULL, the audio data will be converted to the hardware audio format at runtime.

- Loading and playing sounds

  SDL provides a single sound loading routine for your convenience, SDL_LoadWAV().
  After you load your sounds, you should convert them to the audio format of the sound output using SDL_ConvertAudio(), and make them available to your mixing function.

  Example:

```
#define NUM_SOUNDS 2
struct sample {
    Uint8 *data;
    Uint32 dpos;
    Uint32 dlen;
} sounds[NUM_SOUNDS];


void mixaudio(
    void *unused,
    Uint8 *stream,
    int len)
{
    int i;
    Uint32 amount;

    for ( i=0; i<NUM_SOUNDS; ++i ) {
        amount =
          (sounds[i].dlen-sounds[i].dpos);
        if ( amount > len ) {
            amount = len;
        }
        SDL_MixAudio(stream,
          &sounds[i].data[sounds[i].dpos],
            amount,
            SDL_MIX_MAXVOLUME);
        sounds[i].dpos += amount;
    }
}


void PlaySound(char *file)
{
    int index;
    SDL_AudioSpec wave;
    Uint8 *data;
    Uint32 dlen;
    SDL_AudioCVT cvt;

    /* Look for an empty
       (or finished) sound slot */
    for ( index=0;
          index<NUM_SOUNDS;
          ++index ) {
        if ( sounds[index].dpos ==
            sounds[index].dlen ) {
            break;
        }
    }
    if ( index == NUM_SOUNDS )
        return;
```

```
    /* Load the sound file and
       convert it to 16-bit stereo
       at 22kHz */
    if ( SDL_LoadWAV(file,
        &wave, &data, &dlen) ==
        NULL ) {
        fprintf(stderr, "Couldn't "
            "load %s: %s\n",
            file, SDL_GetError());
        return;
    }

    SDL_BuildAudioCVT(&cvt,
      wave.format,
          wave.channels,
              wave.freq,
      AUDIO_S16,
          2,
              22050);
    cvt.buf =
        malloc(dlen*cvt.len_mult);
    memcpy(cvt.buf, data, dlen);
    cvt.len = dlen;
    SDL_ConvertAudio(&cvt);
    SDL_FreeWAV(data);


    /* Put the sound data
       in the slot (it starts
       playing immediately) */
    if ( sounds[index].data ) {
        free(sounds[index].data);
    }
    SDL_LockAudio();
    sounds[index].data = cvt.buf;
    sounds[index].dlen = cvt.len_cvt;
    sounds[index].dpos = 0;
    SDL_UnlockAudio();
}
```

> **Tip:**
> SDL audio facilities are designed for a low level software audio mixer. A complete example mixer implementation available under the LGPL license can be found in the SDL demos archive.

5. CD-ROM audio

- Opening a CD-ROM drive for use

  You can find out how many CD-ROM drives are on the system with the `SDL_CDNumDrives()` function, and then pick which one to use with `SDL_CDOpen()`. The system default CD-ROM is always drive 0.

  The CD-ROM drive may be opened for use even if there is no disk in the drive. You should use the `SDL_CDStatus()` function to determine the state of the drive.

  After you are done using the CD-ROM drive, close it with the `SDL_CDClose()` function.

  Example:

```
{
    SDL_CD *cdrom;

    if ( SDL_CDNumDrives() > 0 ) {
        cdrom = SDL_CDOpen(0);
        if ( cdrom == NULL ) {
            fprintf(stderr,
                "Couldn't open "
                "default CD-ROM: "
                "%s\n"
                SDL_GetError());
            return;
        }

        ...

        SDL_CDClose(cdrom);
    }
}
```

> **Tip:**
> You can get the system-dependent name for a CD-ROM drive using the `SDL_CDName()` function.

- Playing the CD-ROM

  CD-ROM drives specify time either in MSF format (mins/secs/frames) or directly in frames. A frame is a standard unit of time on the CD, corresponding to 1/75 of a second. SDL uses frames instead of the MSF format when specifying track lengths and offsets, but you can convert between them using the `FRAMES_TO_MSF()` and `MSF_TO_FRAMES()` macros.

  SDL doesn't update the track information in the `SDL_CD` structure until you call `SDL_CDStatus()`, so you should always use `SDL_CDStatus()` to make sure there is a CD in the drive and determine what tracks are available before playing the CD. Note that track indexes start at 0 for the first track.

SDL has two functions for playing the CD-ROM. You can either play specific tracks on the CD using `SDL_CDPlayTracks()`, or you can play absolute frame offsets using `SDL_CDPlay()`.

SDL doesn't provide automatic notification of CD insertion or play completion. To detect these conditions, you need to periodically poll the status of the drive with `SDL_CDStatus()`. Since this call reads the table of contents for the CD, it should not be called continuously in a tight loop.

Example:

```
void PlayTrack(SDL_CD *cdrom, int track)
{
    if ( CD_INDRIVE(
        SDL_CDStatus(cdrom)) ) {
        SDL_CDPlayTracks(cdrom,
            track, 0, track+1, 0);
    }
    while ( SDL_CDStatus(cdrom) ==
        CD_PLAYING ) {
        SDL_Delay(1000);
    }
}
```

> **Tip:**
> You can determine which tracks are audio tracks and which are data tracks by looking at the `cdrom->tracks[track].type`, and comparing it to `SDL_AUDIO_TRACK` and `SDL_DATA_TRACK`.

6. Threads

- Create a simple thread

  Creating a thread is done by passing a function to `SDL_CreateThread()`. When the function returns, if successful, your function is now running concurrently with the rest of your application, in its own running context (stack, registers, etc.) and able to access memory and file handles used by the rest of the application.

  Example:

```
#include "SDL_thread.h"

int global_data = 0;

int thread_func(void *unused)
{
    int last_value = 0;

    while ( global_data != -1 ) {
        if ( global_data !=
            last_value ) {
            printf("Data value changed "
                "to %d\n",
                global_data);
            last_value = global_data;
        }
        SDL_Delay(100);
    }
    printf("Thread quitting\n");
    return(0);
}


{
    SDL_Thread *thread;
    int i;

    thread = SDL_CreateThread(
        thread_func, NULL);
    if ( thread == NULL ) {
        fprintf(stderr, "Unable to "
            "create thread:  %s\n",
            SDL_GetError());
        return;
    }



    for ( i=0; i<5; ++i ) {
        printf("Changing value "
            "to %d\n", i);
        global_data = i;
        SDL_Delay(1000);
    }

    printf("Signaling thread to "
        "quit\n");
    global_data = -1;
    SDL_WaitThread(thread, NULL);
}
```

> **Tip:**
> The second argument to `SDL_CreateThread()` is passed as a parameter to the thread function. You can use this to pass in values on the stack, or just a pointer to data for use by the thread.

- Synchronizing access to a resource

  You can prevent more than one thread from accessing a resource by creating a mutex and surrounding access to that resource with lock (`SDL_mutexP()`) and unlock (`SDL_mutexV()`) calls.

Example:

```
#include "SDL_thread.h"
#include "SDL_mutex.h"

int potty = 0;
int gotta_go;

int thread_func(void *data)
{
    SDL_mutex *lock =
        (SDL_mutex *)data;
    int times_went;

    times_went = 0;
    while ( gotta_go ) {
        /* Lock  the potty */
        SDL_mutexP(lock);
        ++potty;
        printf("Thread %d using "
            "the potty\n",
            SDL_ThreadID());
        if ( potty > 1 ) {
            printf("Uh oh, somebody "
                "else is using the "
                "potty!\n");
        }
        --potty;
        SDL_mutexV(lock);
        ++times_went;
    }
    printf("Yep\n");
    return(times_went);
}


{
    const int progeny = 5;
    SDL_Thread *kids[progeny];
    SDL_mutex  *lock;
    int i, lots;

    /* Create the synchronization
       lock */
    lock = SDL_CreateMutex();

    gotta_go = 1;
    for ( i=0; i<progeny; ++i ) {
        kids[i] = SDL_CreateThread(
            thread_func, lock);
    }

    SDL_Delay(5*1000);
    SDL_mutexP(lock);
    printf("Everybody done?\n");
    gotta_go = 0;
    SDL_mutexV(lock);


    for ( i=0; i<progeny; ++i ) {
        SDL_WaitThread(kids[i], &lots);
        printf("Thread %d used the "
            "potty %d times\n",
            i+1, lots);
    }
    SDL_DestroyMutex(lock);
}
```

> **Tip:**
> You can make SDL surfaces thread-safe by passing `SDL_THREADSAFE` as a flag to `SDL_AllocSurface()`. If you do this, `SDL_LockSurface()` and `SDL_UnlockSurface()` serialize access so only one thread can access it at a time.

7. Timers

- Get the current time, in milliseconds

  `SDL_GetTicks()` tells how many milliseconds have past since an arbitrary point in the past.

  Example:

```
#define TICK_INTERVAL    30

Uint32 TimeLeft(void)
{
    static Uint32 next_time = 0;
    Uint32 now;

    now = SDL_GetTicks();
    if ( next_time <= now ) {
        next_time = now +
            TICK_INTERVAL;
        return(0);
    }
    return(next_time-now);
}
```

> **Tip:**
> In general, when implementing a game, it is better to move objects in the game based on time rather than on framerate. This produces consistent gameplay on both fast and slow systems.

- Wait a specified number of milliseconds

  SDL_Delay() allows you to wait for some number of milliseconds.

  Since the operating systems supported by SDL are multi-tasking, there is no way to guarantee that your application will delay exactly the requested time. This should be used more as a way of idling for a while rather than to wake up at a particular time.

  Example:

```
{
    while ( game_running ) {
        UpdateGameState();
        SDL_Delay(TimeLeft());
    }
}
```

  Tip:
  Most operating systems have a scheduler timeslice of about 10 ms. You can use SDL_Delay(1) as a way of giving up CPU for the current timeslice, allowing other threads to run. This is important if you have a thread in a tight loop but want other threads (like audio) to keep running.

8. Endian independence

- Determine the endianness of the current system

  The C preprocessor define SDL_BYTEORDER is defined to be either SDL_LIL_ENDIAN or SDL_BIG_ENDIAN, depending on the byte order of the current system.

  A little endian system that writes data to disk has it laid out: [lo-bytes][hi-bytes] A big endian system that writes data to disk has it laid out: [hi-bytes][lo-bytes]

  Example:

```
#include "SDL_endian.h"

#if SDL_BYTEORDER == SDL_LIL_ENDIAN
#define SWAP16(X)    (X)
#define SWAP32(X)    (X)
#else
#define SWAP16(X)    SDL_Swap16(X)
#define SWAP32(X)    SDL_Swap32(X)
#endif
```

  Tip:
  x86 systems are little-endian, PPC systems are big-endian.

- Swap data on systems of differing endianness

  SDL provides a set of fast macros in SDL_endian.h, SDL_Swap16() and SDL_Swap32(), which swap data endianness for you. There are also macros defined which swap data of particular endianness to the local system's endianness.

Example:

```
#include "SDL_endian.h"

void ReadScanline16(
    FILE *file,
    Uint16 *scanline,
    int length)
{
    fread(scanline, length,
        sizeof(Uint16), file);
    if ( SDL_BYTEORDER ==
        SDL_BIG_ENDIAN ) {
        int i;
        for ( i=length-1;
            i >= 0;
            --i )
            scanline[i] =
                SDL_SwapLE16(
                    scanline[i]);
    }
}
```

Tip:
If you just need to know the system byteorder, but don't need all the swapping functions, include SDL_byteorder.h instead of SDL_endian.h

5 Porting games to Linux, using DirectX sample "donuts" as example

1. Compile the source code

   - Create a Makefile:

```
--- Makefile ---

# The demos to be built
TARGET = donuts

include GNUmake

donuts donuts.exe: donuts.o
```

   - Samples of original Win32 source:

```
--- donuts.c ---

int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine,
                        int nCmdShow )
{
    MSG     msg;
    // save off application instance
    hInst = hInstance;

    while( lpCmdLine[0] == '-' )
    {
        lpCmdLine++;

        switch (*lpCmdLine++)
        {
        case 'e':
            bUseEmulation = TRUE;
            break;
        case 't':
            bTest = TRUE;
            break;
        case 'S':
            bWantSound = FALSE;
            break;
        case 'x':
            bStress= TRUE;
            bTest = TRUE;
            break;
        }
        while( IS_SPACE(*lpCmdLine) )
        {
            lpCmdLine++;
        }
    }

    ScreenX = getint(&lpCmdLine, 640);
    ScreenY = getint(&lpCmdLine, 480);
    ScreenBpp = getint(&lpCmdLine, 8);
```

```
    if( !initApplication(hInstance, nCmdShow) )
    {
        return FALSE;
    }

    if( !InitializeGame() )
    {
        DestroyWindow( hWndMain );
        return FALSE;
    }

    dwFrameTime = timeGetTime();

    while( 1 )
    {
        if( PeekMessage( &msg, NULL, 0, 0, PM_NOREMOVE ) )
        {
            if( !GetMessage( &msg, NULL, 0, 0 ) )
            {
                return msg.wParam;
            }
            if ( !TranslateAccelerator( hWndMain, hAccel, &msg ) ) {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
            }
        }
        else if ( bIsActive )
        {
            UpdateFrame();
        }
        else
        {
            WaitMessage();
        }
    }

} /* WinMain */
```

- Samples of modified Linux source:

```
--- donuts.c ---

/* Some defines to make our job easier */
typedef enum {
    FALSE = 0,
    TRUE
} BOOL;
typedef int DWORD;

/*
 * WinMain
 */
main(int argc, char *argv[])
{
    ScreenX = 640;
    ScreenY = 480;
    ScreenBpp = 8;

    if( !InitializeGame() )
```

```
    {
        exit(1);
    }

    dwFrameTime = SDL_GetTicks();

    while( !gDone )
    {
        UpdateFrame();
    }
    DestroyGame();
    exit(0);

} /* WinMain */
```

2. Identify major subsystems

   Donuts uses DirectDraw for video and DirectSound for audio.

   The Plan: I can replace DirectDraw with SDL's graphic surface routines, and can implement the audio either using SDL's example mixer or by writing my own that supports panning.

3. Rewrite major components

   - Samples of original Win32 source:

```
--- donuts.c ---

BOOL InitializeGame( void )
{
    DDCAPS          ddcaps;
    HRESULT         ddrval;
    DDSURFACEDESC2  ddsd;
    DDSCAPS2        ddscaps;
#ifdef NT_HACK
    DDSURFACEDESC2  DDSurfDesc;
#endif
        LPDIRECTDRAW    pDD;

    score = 0;
    if( bTest )
        ShowLevelCount = 1000;

    if( bUseEmulation )
        ddrval = DirectDrawCreate( (LPVOID) DDCREATE_EMULATIONONLY, &pDD, NULL );
    else
        ddrval = DirectDrawCreate( NULL, &pDD, NULL );

    // Fetch DirectDraw4 interface
    ddrval = pDD->lpVtbl->QueryInterface( pDD, &IID_IDirectDraw4, (LPVOID*)&lpDD);
    if (ddrval != DD_OK)
        return CleanupAndExit("QueryInterface Failed!");

    if( ddrval != DD_OK )
        return CleanupAndExit("DirectDrawCreate Failed!");

    ddrval = lpDD->lpVtbl->SetCooperativeLevel( lpDD, hWndMain,
```

```
                            DDSCL_EXCLUSIVE | DDSCL_FULLSCREEN );
    if( ddrval != DD_OK )
        return CleanupAndExit("SetCooperativeLevel Failed");

    #ifdef NT_HACK
        DDSurfDesc.dwSize = sizeof(DDSurfDesc);
        ddrval = lpDD->lpVtbl->GetDisplayMode(lpDD,&DDSurfDesc);
        if(ddrval == DD_OK)
        ScreenBpp = DDSurfDesc.ddpfPixelFormat.dwRGBBitCount;
    #endif

    // set the mode
    ddrval = lpDD->lpVtbl->SetDisplayMode( lpDD, ScreenX, ScreenY, ScreenBpp, 0, 0 );
    if( ddrval != DD_OK )
        return CleanupAndExit("SetDisplayMode Failed!");

    // check the color key hardware capabilities
    dwTransType = DDBLTFAST_SRCCOLORKEY;
    ddcaps.dwSize = sizeof( ddcaps );

#ifdef DEBUG
    if( GetProfileInt( "Donuts", "force_dest_blt", 0) )
    {
        dwTransType = DDBLTFAST_DESTCOLORKEY;
    }
    bHELBlt = GetProfileInt( "Donuts", "force_HEL_blt", bHELBlt );
#endif

    // Create surfaces
    memset( &ddsd, 0, sizeof( ddsd ) );
    ddsd.dwSize = sizeof( ddsd );
    ddsd.dwFlags = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
    ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE |
                          DDSCAPS_FLIP |
                          DDSCAPS_COMPLEX;
    ddsd.dwBackBufferCount = 1;
    ddrval = lpDD->lpVtbl->CreateSurface( lpDD, &ddsd, &lpFrontBuffer, NULL );
    if( ddrval != DD_OK )
        return CleanupAndExit("CreateSurface FrontBuffer Failed!");

    // get a pointer to the back buffer
    ddscaps.dwCaps = DDSCAPS_BACKBUFFER;
    ddrval = lpFrontBuffer->lpVtbl->GetAttachedSurface(
                lpFrontBuffer,
                &ddscaps,
                &lpBackBuffer );
    if( ddrval != DD_OK )
        return CleanupAndExit("GetAttachedDurface Failed!");

    ddsd.dwFlags = DDSD_CAPS | DDSD_WIDTH | DDSD_HEIGHT;
    ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
#ifdef DEBUG
    if( bHELBlt )
        ddsd.ddsCaps.dwCaps |= DDSCAPS_SYSTEMMEMORY;
#endif
    ddsd.dwWidth = 320;
    ddsd.dwHeight = 384;
    ddrval = lpDD->lpVtbl->CreateSurface( lpDD, &ddsd, &lpDonut, NULL );
    if( ddrval != DD_OK )
```

```
        return CleanupAndExit("CreateSurface lpDonut Failed!");

    ddsd.dwHeight = 128;
    ddrval = lpDD->lpVtbl->CreateSurface( lpDD, &ddsd, &lpPyramid, NULL );
    if( ddrval != DD_OK )
        return CleanupAndExit("CreateSurface lpPyramid Failed!");

    ddsd.dwHeight = 32;
    ddrval = lpDD->lpVtbl->CreateSurface( lpDD, &ddsd, &lpCube, NULL );
    if( ddrval != DD_OK )
        return CleanupAndExit("CreateSurface lpCube Failed!");

    ddsd.dwHeight = 32;
    ddrval = lpDD->lpVtbl->CreateSurface( lpDD, &ddsd, &lpSphere, NULL );
    if( ddrval != DD_OK )
        return CleanupAndExit("CreateSurface lpSphere Failed!");
    // Set the background color fill color

    ddsd.dwHeight = 256;
    ddrval = lpDD->lpVtbl->CreateSurface( lpDD, &ddsd, &lpShip, NULL );
    if( ddrval != DD_OK )
        return CleanupAndExit("CreateSurface lpShip Failed!");

    ddsd.dwHeight = 16;
    ddrval = lpDD->lpVtbl->CreateSurface( lpDD, &ddsd, &lpNum, NULL );
    if( ddrval != DD_OK )
        return CleanupAndExit("CreateSurface lpNum Failed!");

    if( !RestoreSurfaces() )
        return CleanupAndExit("RestoreSurfaces Failed!");

    DL.next = DL.prev = &DL;            // null display list
    DL.type = OBJ_SHIP;
    DL.surf = lpShip;
    lastTickCount = GetTickCount();

#ifdef USE_DSOUND
    if(bWantSound)
    {
        InitializeSound();
    }
#endif

   // initialize DirectInput
   if(!InitInput(hInst, hWndMain))
      return CleanupAndExit("DirectInput initialization Failed!");

    if(bTest)
    {
        ProgramState = PS_ACTIVE;
        setup_game();
    }
    else
    {
        ProgramState = PS_SPLASH;
    }
    return TRUE;
}
```

- Samples of modified Linux source:

```
--- donuts.c ---

BOOL InitializeGame( void )
{
    SDL_Surface *donut_art;
    score = 0;

    // initialize SDL
    if ( SDL_Init(SDL_INIT_VIDEO|SDL_INIT_AUDIO) < 0 ) {
        fprintf(stderr, "Couldn't initialize SDL: %s\n", SDL_GetError());
        return(FALSE);
    }
    atexit(SDL_Quit);

    // set the mode
    screen = SDL_SetVideoMode(ScreenX, ScreenY, 0,
                    SDL_FULLSCREEN|SDL_HWSURFACE|SDL_DOUBLEBUF);
    if ( screen == NULL ) {
        fprintf(stderr, "Couldn't set %dx%dx%d video mode: %s\n",
                    ScreenX, ScreenY, ScreenBpp, SDL_GetError());
        return(FALSE);
    }

    // load the artwork
    donut_art = SDL_LoadBMP("donuts.bmp");
    if ( donut_art == NULL ) {
        fprintf(stderr, "Couldn't load artwork (donuts.bmp)\n");
        return(FALSE);
    }
    donuts = SDL_DisplayFormat(donut_art);
    SDL_FreeSurface(donut_art);
    if ( donuts == NULL ) {
        fprintf(stderr, "Couldn't convert artwork to display format\n");
        return(FALSE);
    }
    SDL_SetColorKey(donuts, SDL_SRCCOLORKEY,
                     SDL_MapRGB(screen->format, 0, 0, 0));

    // get the offsets into the artwork
    Yoff[OBJ_DONUT] = 0;
    Yoff[OBJ_PYRAMID] = Yoff[OBJ_DONUT] + 384;
    Yoff[OBJ_CUBE] =  Yoff[OBJ_PYRAMID] + 128;
    Yoff[OBJ_SPHERE] =  Yoff[OBJ_CUBE] + 32;
    Yoff[OBJ_SHIP] =  Yoff[OBJ_SPHERE] + 32;
    Yoff[OBJ_BULLET] =  Yoff[OBJ_SHIP] + 256;
    Yoff[OBJ_NUM] =  Yoff[OBJ_SHIP] + 256;

    DL.next = DL.prev = &DL;            // null display list
    DL.type = OBJ_SHIP;
    lastTickCount = SDL_GetTicks();

    InitializeSound();

    ProgramState = PS_SPLASH;

    return TRUE;
```

```
}
```

4. Polish the code

   - Remove `InitApplication()`, `MainWndproc()`, and `RestoreSurfaces()`
     — Not needed under Linux.
   - Remove menus and options
     — This simple example doesn't use them.
   - Remove `USE_DSOUND` C preprocessor define
     — We always use sound if available.

5. Add Linux specific features

   - Add icon and title caption

6. Optimize and debug

The original implementation used `SDL_FillRect()` to erase the screen. Under X11 DGA 2.0, there is no accelerated fill function, so writing directly to the video memory was very slow.

We can take advantage of the fact that only certain portions of the screen are updated to clear only those locations, greatly speeding up the framerate.

- Sample of original Linux source:

```
--- donuts.c ---
void EraseScreen( void )
{
    SDL_FillRect(screen, NULL, SDL_MapRGB(screen->format, 0, 0, 0));
}

void FlipScreen( void )
{
    SDL_Flip(screen);
}
```

- Sample of modified Linux source:

```
--- donuts.c ---
struct _dirty {
    int num;
    SDL_Rect rects[1024];
} dirty[2],
  *gdirty = &dirty[0],
  *odirty = &dirty[1],
  *tdirty;

void AddDirtyRect(SDL_Rect *rect)
{
    gdirty->rects[gdirty->num++] = *rect;
}

void EraseScreen( int completely )
```

```
{
    Uint32 black;

    black = SDL_MapRGB(screen->format, 0, 0, 0);
    if ( completely ) {
        SDL_FillRect(screen, NULL, black);
        SDL_UpdateRect(screen, 0, 0, 0, 0);
        gdirty->num = 0;
        odirty->num = 0;
    } else {
        int i;
        for ( i=gdirty->num-1; i>=0; --i ) {
            SDL_FillRect(screen, &gdirty->rects[i], black);
        }
        tdirty = odirty;
        odirty = gdirty;
        gdirty = tdirty;
    }
}

void FlipScreen( void )
{
    SDL_UpdateRects(screen, odirty->num, odirty->rects);
    SDL_UpdateRects(screen, gdirty->num, gdirty->rects);
    odirty->num = 0;
}
```

Porting Linux to other ARM Platforms - A Brief Overview

Russell King
rmk@arm.linux.org.uk, http://www.arm.linux.org.uk/

Abstract

ARM Linux is continually being ported to many platforms, some of them well known like the Geofox, and some less well known like the Acorn RiscPC. As each platform gets added, the various interfaces are generalised, and therefore it becomes easier to port.

1 Introduction

ARM Linux has been around for five years, and over that time it has developed to incorporate a wide variety of ARM processors and machines.

This paper outlines the structure in the ARM kernel. It describes some, but not all of the interfaces.

2 What is ARM?

The ARM is a 32-bit reduced instruction set processor, which has been in existence for at least the past fifteen years. Over this time, the processor has developed both in terms of power and features, while still retaining it's low power consumption.

The market which these processors are targeted for has likewise changed over the years, and has ranged from the desktop machine to hand-held devices to embedded applications. Today, there are desktop machines, palmtops and mobile phones containing these processors to name a few.

Initially, when the ARM processor was first developed, it had a 26-bit address bus. This lasted for around four years, when it was replaced by a 32-bit address bus and a new memory management structure [mmu].

Likewise, there have been several generations of the core processor, and the variants that are produced. To give an example of this, the ARM6 core is at the heart of the ARM610 processors, and the ARM7 core is at the heart of the ARM710, ARM7100 and the CL-PS7110 processors.

The core processor is effectively just the execution unit, register bank and arithmetic logic unit. The variants add virtual write-back caches, write buffers, memory management and other peripherals to the processor. There is a set of rules which defines the architecture of the caches, buffers and memory management, which are defined by the ARM Architecture Reference Manual. For example, there is a set of defined instructions to control the virtual cache, and the architecture of the memory management.

Today, most ARM processors have a virtually mapped non-coherent cache. Since this cache uses virtual addresses, it needs to be specifically flushed when the page tables are altered. Also, the non-coherent nature of the cache means that it does not snoop other bus operations to remain up to date.

There are various options which the ARM variant can follow, for example the ARM710 has an 8kB combined instruction and data cache, whereas the StrongARM 110 has separate instruction and data caches, each of 16kB. The StrongARM 1100 on the other hand has one instruction and two data caches, one 8K and the other 512 bytes.

The memory management is a two level page table with page permission bits, cache control bits, and a few implementation defined options. Also, tied into the memory management is the idea of domains

[domains].

All areas of memory mapped by the page tables must be allocated one of sixteen domains. Associated with each domain are a set of permission bits which effectively override the page-level permission bits. These bits can define an area to be a 'client', where the page permission bits are used, a 'manager' where full access is possible, or 'no access'. Using these, it is possible to quickly control the wholesale access to a memory region without affecting the page table permissions.

As with many RISC architectures, has strict data alignment rules, which means that data should be aligned to it's natural boundary. Any memory access which would cross a 32-bit boundary is always contained within this 32-bit boundary. When this happens, the access may be optionally faulted, or may proceed and load rotated data.

During the development of the ARM processors several new instructions have been added, which include a 64-bit multiply in the ARM7M core, and half-word load and store instructions in the StrongARM processors.

Most ARM processors do not contain floating point instructions, however there is a definition for the floating point instruction set. There are, however, a few processors with floating point incorporated into them. When a processor encounters a floating point instruction, it attempts to pass it to a floating point coprocessor. Where the coprocessor does not exist, the instruction will fault, which allows it to be emulated in software.

All these various faults are directed through a set of vectors located in the first page of virtual memory.

3 Why port Linux to ARM?

Linux is capable of providing a very flexible environment, due to the use of the unix memory model and unix philosophies. Coupling this with the low power requirements of the ARM processor make it an attractive system for hand held computers or any environment which requires low power, low maintainence, high reliability and flexible processing capabilities.

The structure adopted by ARM Linux allows a huge amount of the existing code to be easily reused with minimal effort.

There are cross compilers available which allow development to be performed on other machines such as PCs, which can cut down the development time.

4 What Machines does ARM Linux currently run on?

ARM Linux currently runs on at least 14 models, which include the Acorn Archimedes and RiscPC, the Intel (which was Digital) EBSA285, Itsy, Nexus Technologies NexusPCI, Chalice Technologies CATS, NetWinder, Psion 5, Geofox and Empeg. There are even reports of people wishing to use ARM Linux on public touch screen display terminals!

This paper will only refer to the 32-bit category, which means any processor with an ARM6, ARM7 or StrongARM core. ARM Linux does, however, support the older 26-bit variety.

5 Where to Begin

Ok, so you've been asked by your boss to port Linux onto this board with an ARM processor on. Where do you begin?

First, do you have any technical documentation with the board? The documentation should include information such as the processor type and peripherals which have been fitted to the board. If not, then you may be able to gain the information by looking up the part numbers of the various devices on the board.

Secondly, check the latest kernel sources to determine what architectures are currently supported. The official web pages should have a complete list of the architectures, but they're not always up to date with the cutting edge. If you're not sure, send the me an email.

If the processor core or variant is already supported, then half your problems are already solved. If there is a similar architecture already in the kernel, then it may just require a small number of modifications. These sorts of decisions are unique to each and every type of hardware.

6 Structure

The kernel support for the processor cores is handled by a set of header files, which are split into two groupings for the 26-bit and 32-bit processors. All the common definitions for these families can be found in the include/asm-arm directory, whereas the processor core dependent parts are found in the include/asm-arm/proc-armo and include-asm-arm/proc-armv directories respectively.

7 Header Files

Most header files in the proc directories contain the various definitions and code fragments for each processor family, however there are some which are ARM specific. These are used to provide a level of independence in the arch/arm directory tree. Some of these will be examined in greater detail.

7.1 Processor Independent Assembler

There are two differences between the assembly required for each processor family. The procedure return code is subtly different, and the way the program status register (PSR) is accessed.

GCC fixes up these differences when it compiles C code automatically, but there is no such fixup for the assembler code. Instead, the assembler.h header file is included by all each assembler file in the arch/arm directory tree.

This header file defines a set of macros which allow these differences to be catered for. Some of these macros exist in several forms to allow the most efficient assembler code to be compiled in every case.

Briefly, the macros LOADREGS and RETINSTR are used for procedure return. MODE, SAVEMODE, RESTOREMODE, USERMODE and SVCMODE are used to switch, save and restore the processor privilege state. SAVEIRQS, RESTOREIRQS, DISABLEIRQS and ENABLEIRQS save, restore and affect the interrupt enable bits.

7.2 DMA Cache Handling

Any operation which affects data held external to the processor need to be specifically handled to ensure that any cached data remains valid.

With a virtual non-coherent cache, there are two possible situations which must be correctly handled. If a bus operation modifies some data held in memory and that memory location has been cached, then the cache will become out of date. However, if the processor writes to a cached location, the physical memory will become out of date.

Therefore, it is necessary to either purge or write back the cached data. Sometimes, it is desirable to write back and purge the cache. These operations are handled by the following macros.

dma_cache_inv purges a region of the cache. It may optionally write back any dirty data held in the cache. If the region overlaps a cache line, then the dirty data in that cache line must be written back. This is typically of use when performing a DMA from a device to memory.

dma_cache_wback writes back dirty data in the specified region to memory. It is preferable that the cache is not invalidated, but this may not be possible to achieve with every processor. This is typically used when the DMA is from memory to a device.

dma_cache_wback_inv writes back and purges the cached data in region. This is typically used when the DMA is from memory to a device, and you know that the data is no longer required to be held in the cache.

Since Linux has traditionally been used with coherent caches, these macros need to be added to device drivers as required.

7.3 Memory Initialisation

The core of the memory map initialisation is in arch/arm/mm/init.c, however there are elements of this which are processor core specific. This is handled by mm-init.h, and should at least contain two inline functions - setup_page_tables and mark_usable_memory_areas.

setup_page_tables is responsible for creating the page tables for the idle task which contains the kernel and IO regions. This mapping will also be included into all child tasks.

Initially, all pages are marked as a reserved page. The kernel then claims pages of memory after the end of its BSS segment. The remaining pages then need to be marked as non-reserved pages. The mark_usable_memory_areas is responsible for aligning the memory start pointer correctly (start_mem), and clearing the PG_reserved bit for all unused pages.

Depending on the processor core and machine, it may be necessary to also mark other areas free, but this depends very much on the memory layout.

7.4 Context Switching and the PSR

When a context switch is performed, the processor state is saved onto the kernel stack, and the stack pointer saved in the tss structure. The saved state information is described by the context_save structure, and is needed by some parts of the kernel to determine where a thread has blocked. Incidentally, in 2.3.11 and up, this structure is now known as the thread structure.

The PSR differences have to be catered for when setting up the task registers. This is handled in an inline function, start_thread, which initialises all the tasks registers.

7.5 Page Tables

When the ARM memory management system was designed, certain bits in the page tables were left to the processor implementor to define their use. Unfortunately, this uses all bits in the page table entries, leaving none available for software defined use. The Linux memory management system relies in being able to have a page young bit and a dirty bit associated with each page of memory.

For each page table, Linux keeps two copies. The first table holds all the necessary information for the Linux memory management, and the second table holds the processors view. It is up to the processor variant code to perform the correct translation via set_pte and set_pmd.

The cache and TLB management functions, as well as these macros are directed through to the processor variant code in arch/arm/mm/proc-*.S.

8 Processor Variant Code

All the processor variant code is held in the proc assembly files found in arch/arm/mm. Since the vast majority of processors only have minor differences, these files can contain several processor variants.

All the code contained within these files is called through a structure of function pointers. This allows the variant to be easily selected at boot time.

All function pointers are expected to be pointing at valid code, including ones which may not be valid for this variant. In this case, it must point at a suitable return instruction, normally 'mov pc, lr'.

Each function is expected to perform a certain minimum. For example, some processor variants are only capable of cleaning and purging the whole cache. In this case, all the flush_cache_* and clean_cache_* functions can point at the same routine.

Included in these files is the actual context switch function. This allows the cache handling and switch to be done in one call to this level.

When support is added for a new processor variant, the proc assembler files may need to be updated to handle the new cache operations, and the head-armv.S and setup-armv.S updated to select the correct set of function pointers.

9 Machine Peripherals

Now, let us move on to look at the support in ARM Linux for the various hardware architectures. Each different hardware is given it's own architecture directory.

Some architectures are combined into the same directory. For instance, the EBSA285, CATS and NetWinder all use the DC21285 [DC21285] host PCI bridge. Effectively they are all the same basic machine, but with different devices on the PCI bus. This allows one kernel to be built per family of machines, rather than per machine.

In the same way that the commonalities between processors are found in the top level include/asm-arm directory, the same exists for the architectures.

There are many architecture specific files, some of which will be briefly covered here.

9.1 Hardware Definitions

The hardware header file, hardware.h, should contain the definitions for the location of any fixed devices. For instance, in the EBSA285 port this file contains the base and size definitions for the PCI configuration space, the PCI IO space and so forth. It must also contain the definition of PAGE_OFFSET, the offset of the kernel's main system memory image.

9.2 DMA Numbering

dma.h defines all the DMA numbers for the architecture. Unfortunately, during the early days of Linux, the ISA DMA numbers 0 to 7 were hard coded into the device drivers. There would be a considerable amount of work required to fix this now, so it is preferable to keep these reserved for ISA in any architecture that may support these.

9.3 IO Code

The IO definitions can be found in the io.h header file. These may be either macros or inline functions, and may contain inline assembler.

In a similar way to the x86 io.h header file, it is possible to define both inline assembler and constant optimised IO macros if required, however it is not mandatory.

The EBSA285 architecture code gives a good illustration of the minimum number of macros required.

9.4 Memory Translations

The memory translations can be found in memory.h. These allow the kernel to translate a virtual system memory address to a physical system memory address, and vice versa. These are performed by the __virt_to_phys and __phys_to_virt macros respectively. Note that the address must point to an area of system memory. It should not be used for other translations.

Another set of translations called __virt_to_bus and __bus_to_virt are used to translate a virtual system memory address to an address seen on the IO bus. These are used to translate addresses for use with PCI bus master devices and DMA controllers.

9.5 Other Areas

There are other re-usable code fragments in the arch/arm tree. For instance, the DMA system is nicely layered so you can add ISA DMA support along with some other DMA support. Again, the EBSA285 illustrates how you can do this, which can be found in arch/arm/kernel/dma-footbridge.c.

Some architectures have LEDs, which can be used for debugging or other indications. There is a LED driver specifically for this, which consists of a single function called led_event. Using this, it is possible to have an LED which flashes in response to the timer interrupt (useful for machines with no display attached and/or debugging). The LED drivers can be found in arch/arm/kernel at the present time.

10 The Future

As can be seen, there are a lot of abstractions have been made in the ARM Linux kernel, which make it easier to add new architectures. With each architecture that is added to the kernel, more abstractions are added. Linux will continue to be ported to more and more ARM based architectures in the future, and the hope is that this has been a useful overview to some of the interfaces already present in the kernel.

11 More Information

If you wish to find out more information about the ARM port, there are several sites carrying information. Most of these can be accessed via the ARM Linux web pages at `http://www.arm.linux.org.uk/` or `http://www.arm.linux.org.uk/ rmk`.

Patches, code and an early RedHat-like distribution can be found on the official FTP site at `ftp://ftp.arm.linux.org.uk/pub/armlinux/`.

Lastly, there are several mailing lists for ARM Linux. Their details are also on the official ARM Linux web site.

12 Acknowledgments

Lastly, thanks should be extended to the following people, without whom, ARM Linux wouldn't be where it is today:

Werner Almesberger Philip Blundell Tim Chown Alan Cox Jeremy Doran Martin Ebourne Richard Earnshaw Dave Gilbert Linus Torvalds David Walker

References

[DC21285] Intel, *Core Logic for SA-110 Microprocessor*

[domains] ARM Ltd, *ARM Architecture Reference Manual*, ARM DDI 0100A (1996) p. 7-23

[mmu] ARM Ltd, *ARM Architecture Reference Manual*, ARM DDI 0100A (1996) p. 7-1 - 7-30

Inside the Lizard
A Look at the Mozilla Technology and Architecture

Mike Shaver
shaver@mozilla.org
Michael Ang
mang@subcarrier.org

Abstract

In October of 1998, the Mozilla project changed direction in a dramatic fashion. Spurred by cries for no-excuses standards compliance and improved architecture, the new roadmap focuses on the "NGLayout" (aka "Gecko") layout engine, cross-platform front end code and scriptable components.

This document discusses key aspects of the new architecture, including information on developing your own applications atop the Mozilla platform, writing custom components and porting to new platforms. Specifically, XPCOM, XPFE, XUL, RDF, XPConnect and porting issues will be addressed.

1 Introduction

After the initial release of the Mozilla code on March 31, 1998, there were many enthusiastic attempts to make that codebase (known as the "Classic" code) provide features such as incremental layout with reflow, fully compliant CSS and a Level-1 Document Object Model. The nature of the Classic code, though, made such tasks difficult and error-prone. (Not impossible, though: one of the authors of this paper worked on a project known as "Perignon" and made significant progress in improving the quality of the CSS implementation.)

The relative intractability of the Classic code, a desire to fully apply the lessons in cross-platform front-end development learned painfully over the first 4 releases of Navigator and the promise of the new layout engine (originally known as "Raptor", and subsequently as "NGLayout" or "Gecko") combined with pleadings from groups like the Web Standards Project to cause a radical shift: we would abandon much of the Classic code, rebuilding the browser and other applications (editor, mail/news) on top of Gecko. The user interface would be described in XML, HTML and JavaScript, and the application would be composed of XPCOM components linked with C++ and JavaScript glue.

2 Modular Code: XPCOM

The XPCOM system is a cross-platform variant of the DCE/Microsoft COM system for building binary components. We use it to provide clear "plugablity" and modularity boundaries between sections of code, and also to manage object lifecycles through reference counting.

XPCOM strongly separates interface from implementation, down to the binary level. These two abstractions map to two C++ classes: an interface class and an implementation class. The client interacts directly only with the interface class, which contains no implementation details and has a well-defined binary format.

Since clients of XPCOM know only about the interface to an object, the implementation can be safely upgraded or replaced, compiler differences are hidden, and objects can even be implemented in another language (e.g. JavaScript).

2.1 XPCOM techniques

XPCOM objects can expose multiple interfaces (somewhat similar to multiple inheritance). `QueryInterface` is XPCOM's equivalent of C++'s dynamic cast. To access an object through a particular interface, you `QueryInterface` for that interface.

Since the same object may be accessed through several different interface pointers, reference counting is used to keep track of the lifetime of objects. When a client receives an interface pointer, the reference count on the underlying object is increased by one. Once the client is done with the interface pointer, they call Release on the pointer, and the reference count is decremented. Once the reference count reaches zero, the object is destroyed. `nsCOMPtr` is a smart pointer implementation that takes care of these details automagically.

XPIDL is the cross-platform interface definition language used for XPCOM. XPIDL files are processed to generate C++ headers for XPCOM objects, runtime type information to call objects dynamically from JavaScript (and potentially other languages), and (Real Soon Now) online documentation.

2.2 XPCOM and JavaScript interoperation

Using XPConnect, it is possible for JavaScript to seamlessly interoperate with XPCOM without needing any glue code. XPConnect uses the runtime type information previously built from the XPIDL.

Code using XPCOM interfaces should not generally know or care what language is used to implement underlying objects. XPConnect allows JavaScript to interoperate seamlessly with XPCOM. JavaScript code can manipulate XPCOM objects, and JavaScript objects can present XPCOM interfaces to be called by other XPCOM code.

XPConnect uses the runtime information generated from XPIDL interface definition files when JavaScript calls into XPCOM. No glue code needs to be written, and this results in significant footprint savings.

Connection technologies for other languages are possible. (Sun engineers are currently investigating interoperation of XPCOM and Java.)

3 User Interface: XPFE and XUL

In the Classic code base, there were separate front ends written for Mac, Windows, and UNIX. Each platform's front end was maintained by a separate group of developers. Adding a simple new feature such as a button required duplicated efforts and coordination effort. More complex features often had subtle differences on the various platforms.

The pain of maintaining these separate front ends had to end.

3.1 XPFE

Mozilla now uses a Cross-Platform Front End (XPFE) that makes it possible to define user interface appearance and application behaviour using W3C standards instead of platform-specific C++.

XPFE is a toolkit that minimizes the amount of platform specific code needed for building applications like browsers and mail clients. XPFE makes available a standard set of widgets and graphics services to render UI elements, allowing pixel level control of UI if necessary.

3.2 XUL

XUL, which stands for XML-based UI Language and is pronounced "zuul", provides a way to specify UI appearance and application logic using an XML dialect, CSS, and JavaScript (or C++).

XUL is simply XML with specific meaning defined for a few element types, and into which HTML and JavaScript can be scattered. XUL makes it easy for developers with a knowledge of HTML and JavaScript to design UI: this opens up UI development to a whole new set of programmers. The JavaScript application logic can access XPCOM (and thus most libraries) using XPConnect, making it possible to build complete applications.

The combination of appearance and behaviour is called "chrome". Chrome is loaded from `.xul` files and associated JavaScript and CSS files; these files can be downloaded over the network at runtime.

Most of the UI for the browser is written in XUL, and is thus open for hacking. Adding a button on the toolbar that opens a tree control and populates it from a remote data source is simply a matter of writing some XML and JavaScript.

XUL lowers the bar for creating UI. One of the goals of XUL is to enable random hackers to create new features easily and share them with others. This will allow the UI to enjoy the same massively parallel development that has benefited other Open Source projects.

4 RDF

The Resource Description Framework[1] is a graph-based model for describing Internet resources (like web pages and email messages), and how these resources relate to each other. Mozilla's use of RDF stretches the common usage of "Internet resources", though, and adds resources like bookmarks, local file system information and address book content to the mix.

4.1 RDF in Brief

RDF allows arbitrary hierarchical content to be manipulated through a common API, and serialized in a standard syntax. This serialization syntax, known as RDF/XML, allows RDF-using applications to communicate their graphs to each other, and store those graphs persistently.

Within Mozilla, there are APIs provided to manipulate RDF nodes (resources) and assertions (links between nodes, like graph arcs or objects holding "pointers" to other resources). This allows Mozilla components to act as RDF "datasources" and perform operations on those datasources, without having to go through the RDF/XML serialization syntax.

[1] http://www.w3.org/RDF

4.2 Uses of RDF in Mozilla

RDF is used in Mozilla to represent many kinds of interesting data, from bookmarks and address books, to mail folders and remote search engines. Examples of RDF usage in Mozilla include:

- Bookmarks
- Mail folders
- Newsgroups
- Remote site maps
- "Sidebar" or "Flash panel" content
- Browser history and browsing profile
- Address book
- "Related Links"
- Local filesystems
- Remote FTP sites
- Web search engines

To display RDF data in Mozilla, the XUL tree widget is usually employed[2]:

```
<tree id="bookmarksTree"
      datasources="rdf:bookmarks">
  <treecol id="NameColumn"
    rdf:resource="Name"/>
  <treecol id="URLColumn"
    rdf:resource="URL"/>
  <treebody id="NC:BookmarksRoot">
  </treebody>
</tree>
```

The bookmarks source is specified by `rdf:bookmarks` and the resource used to "root" the tree is `NC:BookmarksRoot`. The assertions on the bookmarks graph named by `Name` and `URL` are used to populate the columns of the tree grid, and adding or removing new assertions to the RDF graph used will cause the tree widget to automatically update with new data. (Typesetting concerns required a small liberty to be taken with the example: URIs are used for property naming, and the Mozilla datasources would

[2] The RDF tree syntax is in flux at the time of this writing; please consult `mozilla/rdf/resources/bookmarks.xul` in a current source tree for an up-to-date example

have used `http://home.netscape.com/NC-rdf#Name` and `http://home.netscape.com/NC-rdf#URL` as the canonical `Name` and `URL` assertion names, respectively.) The tree widget uses the assertion named `http://home.netscape.com/NC-rdf#Child` to indicate "nesting", and will populate the top level of the tree with the `...NC-rdf#Children` of the specified root node.

The XPCOM interfaces are accessible through programmatic means as well, allowing you to create resources; add, remove and query assertions between resources; and access registered datasources. A simple example of using these interfaces can be found in `mozilla/rdf/resources/xpidl-test-1.xul`.

4.3 Creating New Datasources

To reflect another kind of resource into the RDF universe, a new RDF datasource must be implemented. The `nsIRDFDataSource` interface must be implemented directly if the datasource wishes to react to RDF operations in a specialized manner, such as adding a file to the filesystem when a new assertion is made, and is described in `mozilla/rdf/base/idl/nsIRDFDataSource.idl`. If the datasource does not require special handling of the `nsIRDFDataSource` API operations, XPCOM aggregation can be used to let the `nsIRDFInMemoryDataSource` handle most of the RDF operations. This approach is often used when the datasource is parsing data in a non-RDF format, and perhaps writes changes out to the original storage when the RDF engine has finished whatever modifications are performed.

When a datasource generates a graph to represent its chosen data, the selection of "vocabulary" is key: merging, sorting and other useful RDF features require that the participating datasources use the same assertion names. In addition to the `NC` vocabulary used for many of the Mozilla datasources, the RDF Schema Specification[3] and Dublin Core[4] provide other standard vocabularies for describing electronic resources.

Once your datasource is implemented, you must use the component manager to register your datasource with the RDF infrastructure. Detailed discussion of this process is beyond the scope of this document, but a good example can be seen in `mozilla/rdf/build/nsRDFFactory.cpp`.

A more detailed treatment of issues involved in RDF datasource implementation is available in the "datasource howto" [5].

[3]http://www.w3.org/TR/PR-rdf-schema/
[4]http://purl.oclc.org/dc/
[5]http://www.mozilla.org/rdf/doc/datasource-howto.html

5 Porting Issues

In order to port Mozilla to a new platform, three major components require attention. (It is assumed that there is an appropriately capable C/C++ toolchain available — gcc 2.7.2 and egcs are suitable. If the toolchain is not sufficiently capable, additional effort may be required: as an example, the OpenVMS port had issues with a linker that did not distinguish symbols on the basis of case.)

The foundation of Mozilla's portability is the Netscape Portable Runtime (NSPR). Applications built on NSPR can use system facilities such as threads, thread synchronization, I/O, interval timing, atomic operations, and several other low-level services in a platform-independent manner. Much of the work in porting Mozilla to a new platform is in porting NSPR.

The `widget` and `gfx` libraries provide implementations of the Mozilla cross-platform toolkit APIs for various platforms. At the time of this writing, implementations exist, in various states of completeness, for BeOS, GTK+, Macintosh, Motif, OS/2, Photon, Rhapsody, Windows, and Xlib systems, and a Qt implementation is under active development. The `widget` portion is responsible for providing standard toolkit widgets such as menus, buttons, scrollbars, text entries and checkboxes. `gfx` provides primitive graphics and rendering capabilities, including line drawing, image rendering and text display. The Mozilla source contains implementations of `gfx` widgets as well, which are rendered from primitives without the assistance of the platform's `widget` implementation. This permits Mozilla to fully support HTML4 rendering requirements, including partial opacity for form elements and use of animated images in backgrounds. The plan of record is to switch completely to the `gfx`-rendered widgets as soon as they are complete, likely by M9 or M10. However, in order to integrate Mozilla into a desktop environment such as KDE, it is necessary to port at least message passing, timers, and menus from the underlying toolkit (Qt in this case). There is an effort underway to make cross-platfrom menus, so the menu requirement will probably go away in some places.

`xptcall` is used by XPConnect and is required on all platforms. Porting `xptcall` requires knowledge of the platform's calling conventions at the assembly language level (this isn't as bad as it may sound). As of July, 1999, there are still several platforms (including some Linux platforms) that do not have a working `xptcall`. Significant detail on the topic of porting the `xptcall` library, as well as a list of completed and developing ports, can be found in `mozilla/xpcom/libxpt/xptcall/porting.html`.

6 Breaking News

An incomplete list of current projects in the Mozilla codebase at the time of this writing:

- The *Necko* project to develop an improved networking library for Mozilla is in the process of integrating its new code into the rest of the Mozilla codebase. During the integration process, the `--enable-necko` argument to `configure` will switch on the Necko implementation.
- Christopher Blizzard's Xlib-only `gfx` implementation is maturing rapidly, and can be meaningfully used as of the M8 release. Work is being done to make it possible to use the Xlib `gfx` implementation with other X-based `widget` implementations, which would reduce the porting burden for new X toolkits and allow all users of Mozilla on X, regardless of toolkit preference, to benefit from common work to improve rendering quality and performance.
- Ramiro Estrugo is working to make toolkits runtime-selectable on Unix-like platforms. This will allow the user to switch between GTK+, Motif, Xlib and others through a preference setting, rather than requiring separate binaries to be distributed. It is likely that `gfx` and `widget` will be individually selectable.

7 Acknowledgements

The authors would like to specifically thank the following for their documentation and instruction:

- Chris Waterson (*waterson@netscape.com*)
- David Hyatt (*hyatt@netscape.com*)
- Scott Collins (*scc@netscape.com*)
- John Bandhauer (*jband@netscape.com*)
- Warren Harris (*warren@netscape.com*)
- Christopher Blizzard (*blizzard@redhat.com*)

In addition, all the Mozilla developers, testers, documenters, kibbitzers and cheerleaders are deserving of heartfelt thanks for making Mozilla such an exciting project on which to work.

8 Additional Information

- Useful URLs.
  - Roadmap: `http://www.mozilla.org/roadmap.html`
  - XPCOM: `http://www.mozilla.org/projects/xpcom/`
  - XPFE/XUL: `http://www.mozilla.org/xpfe/`
  - RDF: `http://www.mozilla.org/rdf/doc/`
  - XPConnect: `http://www.mozilla.org/scriptable/`
  - Weekly status: `http://www.mozilla.org/status/`
  - "SeaMonkey" (Mozilla 5.0): `http://www.mozilla.org/projects/seamonkey/`
- Source browsing. The complete Mozilla source tree is browsable via the LXR web interface, and the hyperlinked version is updated approximately every 15 minutes. To view a source file through LXR, remove the leading `mozilla/` prefix and replace with `http://lxr.mozilla.org/mozilla/source`. For example, `mozilla/js/src/jsapi.c` becomes `http://lxr.mozilla.org/mozilla/source/js/src/jsapi.c` (modulo typesetting damage).
- Mailing lists and newsgroups. Mozilla hosts several dozen mailing list/newsgroup pairs related to a variety of topics. `http://www.mozilla.org/community.html` lists many of them, and the general forms `mozilla-`*`topic`*`@mozilla.org` and `netscape.public.mozilla.`*`topic`* hold for all. Messages sent to the mailing list or posted to the newsgroup appear in both places. New fora are announced on the Mozilla web site periodically, as they are created.

Linux kernel programming (& debugging)

Jes Sørensen
European Laboratory for Particle Physics - CERN
Jes.Sorensen@cern.ch

Linux kernel programming

(& debugging)

Ottawa Linux Symposium
07/23/99

Jes Sorensen
European Laboratory
for Particle Physics - CERN
http://www.cern.ch/
Jes.Sorensen@cern.ch

Agenda

- Kernel intro + versions
- Basic kernel programming rules
- Source layout
- Kernel configuration & compile, modules
- Kernel bootstrap & init
- printk()
- Memory allocation
- Address spaces & correct access
- Device probing & resource allocation
- Interrupts, bottom half's, timers
- User space access
- Waiting inside the kernel (delays)

Agenda (continued)

- Locking mechanisms
- System calls & privileges
- Kernel threads
- ioctls, sysctl & /proc
- Portability across architectures
- Multi kernel version supporting code
- SMP, atomic access, spin locks
- Data alignment & performance
- initdata
- OOPS (and other aspects of debugging)
- Changes in 2.3.x
- Other information sources

Kernel intro & versions

- Stable (even) and developer (uneven):
  - 2.2.x current stable release (previous 2.0.x)
  - 2.3.x current developer release (expect radical changes every now and then)
  - This talk will mainly focus on 2.2.x
- Kernel maintained by Linus Torvalds, some modules available as separate patches from third parties (eg. PCMCIA, some non Intel architectures, etc.)
- Object Oriented C and assembly
- All code is GPL (Gnu general Public License) or GPL compatible. The normal BSD license is not GPL compatible! - Dual licensing is possible!

Basic rules

- No libc in the kernel, only few basic functions
- Use C (no C++ !!) whenever possible, only inline assembly for architecture specific
- Avoid unncessary #ifdef's (#ifdef <arch> !!)
- No floating point usage
- Only put things in the kernel that really need to go into the kernel (think twice!)
- Runtime options preferred to compile time options (flexibility)
- Use private header files for things that are not supposed to be public (ie. not in include/)
- Write in English (code and comments)

Kernel programming tools

- gcc + binutils (as, ld etc)
- gdb
- your favorite editor (emacs, vi etc.)
- grep
- find
- coffee
- diff
- patch

- Nothing special - really!

Kernel source layout

- **Documentation/**
- **Makefile**
- **arch/{alpha,i386,sparc,...}**
- **drivers/{block,char,net,scsi,...}**
- **fs/{ext2,fat,nfs,ntfs,ufs,...}**
- **include/{asm-<arch>,net,linux}**
- **init/**
- **ipc/**
- **kernel/**
- **lib/**
- **mm/**
- **modules/**
- **net/**
- **scripts/**

How to build a kernel

- Fresh build:
  - cd <source-dir>
  - make config (menuconfig/xconfig)
  - make dep
  - make install
  - make modules
  - make modules_install

- Applying patches:
  - cd <source-dir>
  - patch -p1 -d. < patch-file
  - make oldconfig
  - make dep
  - ....

Kernel bootstrap (1)

- arch/<arch>/kernel/head.S (assembly):
  - Initial minimum setup:
    - Shutdown remains of previous operating system
    - Initial mmu setup
    - Copy kernel to final location
    - Jump to start_kernel()

Kernel Bootstrap (2)

- init/main.c:
  - Initialize basic kernel modules:
    - architecture specific
    - memory management
    - interrupts/traps
    - scheduler
    - console
    - various caches (filesystem, directory etc)
    - virtual memory subsystem
    - SMP
    - busses (PCI, MCA, SBUS etc.)
    - devices
    - filesystems
  - mount root
  - exec init process

Kernel Bootstrap (3)

- Early initialization of memory subsystem, interrupts etc.
- Drivers, filesystems (what's the difference?) etc. will have a fully functional system - no need for special treatment for module vs statically compiled drivers

Device drivers and the VFS (1)

- Device drivers, file systems etc. (modules) are registered in the VFS (network devices are not directly in the VFS, but similar)
- Module can provide just a subset of the possible functions (no sense to mmap net-device)
- Most common functions & behavior:
  - **probe/init**: detect presence of hardware, allocate control structures, register driver
  - **open**: activate hardware/filesystem, allocate + init run time resources etc.
  - **close**: disable hardware/filesystem, release run time resources etc.

Device drivers and the VFS (2)

- **read**: prepare & provide data to caller, read from device etc.
- **write**: store/proces data provided by caller, write to device (net-devices have start_xmit)
- **mmap**: memory map raw device registers, control structures or actual device data
- **ioctl**: module specific io control; set buffer size, set volume, read device geometry etc. etc.
- **interrupt handler**: process hardware interrupts (modules supporting real hardware devices)
- **init_module**: call probe and other module init (loadable module specific)
- **cleanup_module**: release resources allocated by init_module and probe/init (loadable module specific)

printk() - can't live without it

- **printk()** - works more or less like printf() in user space. syslog priorities can be put in front of the content to be printed (recommended):

```
printk(KERN_INFO "%s: Firmware running\n",
    dev->name);
```

- printk() priorities:
  - **KERN_EMERG**
  - **KERN_ALERT**
  - **KERN_CRIT**
  - **KERN_ERR**
  - **KERN_WARNING**
  - **KERN_NOTICE**
  - **KERN_INFO**
  - **KERN_DEBUG**

Memory allocation

- kmalloc()/kfree():
  - Similar to malloc()/free(), however limited to a maximum request of 128KB
  - Guarantee contiguous block of physical memory
- get_free_pages(gfp, order) / free_pages(order):
  - Allocates power of 2 blocks of pages
  - **get_free_page(gfp)** == **get_free_pages(gfp, 0)**
  - Commonly used gfp masks:
    - **GFP_DMA** (to be accessibly by bus-master device)
    - **GFP_KERNEL**
- vmalloc()/vfree():
  - Allocates block of virtual memory which is not guaranteed to be contiguous physical memory

Kernel Address Spaces

- Physical memory
  - 'Real' host memory
- Bus shared memory (eg. PCI shared mem)
  - Use **ioremap()** to make accessible to kernel. Access with **read[bwl]()**/**write[bwl]()** as it may be presented in strange ways (eg. Alpha sparse memory)
- Virtual kernel memory
  - Memory as seen by the running kernel and in which the kernel is running
  - Virtual kernel address != physical address!!
- ISA/PCI style I/O address space
  - Only accessible via **in[bwl]()**/**out[bwl]()**

Address conversion

- Physical addresses - physical addresses are <u>only</u> of interest to the mm subsystem!!
  - **virt_to_phys()**: Convert a virtual kernel space address to a physical address
  - **phys_to_virt()**: Convert a physical address to a virtual kernel space address
- Bus addresses are physical addresses as seen by devices at the other end of a bus (PCI bus-master devices).
  - **virt_to_bus()**: Convert a kernel virtual address to a bus address
  - **bus_to_virt()**: Convert a bus address to a kernel virtual address
  - Bus addresses are the ones you will normally use in device drivers!
  - <u>Bus address != physical address</u> !!

Device Probing (ISA)

- Resource information for a device driver:
  - Interrupts
  - Device Shared Memory
  - I/O ports

- Probing for ISA devices:
  - Register layout for your device is known. Write to the assumed registers and read back and check if the device reacts as expected - if not try next I/O port
  - Interrupt probing: Enable one interrupt at a time, make the device generate an interrupt
  - Alternatively let the user specify the information
  - ISA device probing is dangerous!

PCI device probing

Manufacturer and product codes known for device:

```
struct pci_dev *pdev = NULL;
struct my_private *priv = kmalloc(sizeof(struct my private),
                                  GFP_KERNEL);

if (!pci_present())
    return -ENODEV;

while ((pdev = pci_find_dev(PCI_VENDOR_ID_ESSENTIAL,
                            PCI_DEVICE_ID_ESSENTIAL_ROADRUNNER,
                            pdev))) {

    priv->irq = pdev->irq;
    priv->io = pdev->base_address[0] & PCI_BASE_ADDRESS_IO_MASK;
    priv->shared1 = pdev->base_address[1];
    priv->shared2 = pdev->base_address[2];

    ....

    my_device_init();

    ....
}
```

Interrupt Allocation

- Interrupts (irqs) are reserved with **request_irq()**. Sharing irqs is recommended for sane devices (PCI etc).
- ISA does not support shared irqs!
- Linux 2.0.x does not support shared irqs!

```
static void my_dev_init(struct device *dev, struct my_priv *priv)
{
    if (request_irq(dev->irq, my_int, SA_SHIRQ, priv->name, dev)) {
        printk(KERN_WARNING "%s: Requested IRQ %d is busy\n",
               dev->name, dev->irq);
        return -EAGAIN;
    }
 ...

static void my_int(int irq, void *dev_id, struct pt_regs *ptregs)
{
    struct device *dev = (struct device *)dev_id;
    priv = (struct my_priv *)dev->priv;

    if (!(inb(priv->io + MY_IRQ_REG) & MY_INT_PENDING))
        return;
```

Resource Allocation (2)

- There is no method for reserving device shared memory! Multiple drivers can map the same shared memory region and access it simultanously -- implement your own locking if necessary!
- I/O ports are allocated with **request_region()**:

```
request_region(my_ioaddr, MY_REGION_SIZE, dev->name);
```

Interrupt handlers

- Interrupts 'block' other parts of the kernel while running (and other interrupts) - problem for timing critical things (serial etc.)
- Reduce the work in the interrupt handlers to a bare minimum:

    1) acknowledge the interrupt
    2) do your work
    3) get out of the interrupt handler

bottom half handlers

- Low priority non-timing-critical work that can run with other interrupts enabled should be put in a 'bottom half' handler
- Use a generic bottom half handler if possible or create one for a specific purpose if necessary (number of bh's is limited)
- bh's are run last by the interrupt handler with all interrupts re-enabled
- See include/linux/interrupt.h for bh's currently defined

Using the generic bh handler

```
struct tq_struct immediate;

static void my_bh_handler(struct my_data *data) {
    ... do something ...
}

void my_queue_bh(struct my_data *data) {

    /* Initialize task queue structures */
    immediate.next = NULL;
    immediate.sync = 0;
    immediate.routine = (void *)(void *)my_bh_handler;
    immediate.data = data;

    queue_task(&immediate, &tq_immediate);
    mark_bh(IMMEDIATE_BH);
```

Timers

- The kernel has a global timer known as 'jiffies' ticking at **HZ** ticks per second (HZ = 100 on most architectures)
- To avoid busy waiting use the timer software interrupt (example while waiting for a disk drive motor to spin up)
- Once a timer has expired it will not run again
- Do not add the same timer again before it has expired!

```
struct timer_list mytimer;

static void my_timer_func(unsigned long data) {
    printk(KERN_WARNING "My timer expired!\n");
}

void my_set_timer(int delay, struct my_data *data) {
    init_timer(&mytimer);
    mytimer.data = (unsigned long)data;
    mytimer.function = my_timer_func;
    mytimer.expires = jiffies + (delay * HZ);
    add_timer(&mytimer);

!! Alternative for modifying a running timer: !!
    mod_timer(&mytimer, delay * HZ);
```

User space access

- Range copy similar to BSD copyin/copyout:
  - **copy_from_user(*to, *from, size)**
  - **copy_to_user(*to, *from, size)**
- Single word (char/short/int(/long)):
  - **get_user(dest, *useraddr)**
  - **put_user(source, *useraddr)**
- Single word (non address checking)
  - **__get_user(dest, *useraddr)**
  - **__put_user(source, *useraddr)**

- Checking versions all return zero for succesful access, non-zero for an error (**copy_{to,from}_user()** returns the amount of data not copied on error)

Waiting inside the kernel

- Long delay loops inside the kernel are a <u>bad</u> solution! - To be avoided whenever possible!
- Use a wait queue to wait for events, like device ready etc. (described later)
- Use **udelay()** for short delays (usecs)
- If absolutely necessary (for jiffies waiting):
  - **time_before()**, **time_before_eq()**
  - **time_after()**, **time_after_eq()**

```
void some_secs_delay(unsigned long delay)
{
    unsigned long myjif;

    myjif = jiffies + delay * HZ;
    while (time_before(jiffies, myjif));
}
```

Locking mechanisms

- Linux provides various locking mechanisms used to protect data structures etc:
  - semaphore
  - wait queue
  - spin lock + read/write lock (SMP special - covered later)
- All mechanisms are thread + SMP safe
- Classic 'disable all interrupts' (sledgehammer approach) is to be avoided whenever possible - never in hot path code

Semaphores

- The semaphore is a traditional mutex
- If a semaphore is 'taken' the task will sleep until it becomes available (unlike spin locks)
- Use semaphores in 'task-related' code such as system calls etc. Rarely used in device drivers and never in interrupt handlers!
- Take semaphore with **down()** or **down_interruptible()** - for killable tasks use **down_interruptible()** (ie. in syscalls)
- release a semaphore with **up()**
- Init semaphore with **MUTEX** or **MUTEX_LOCKED**

Semaphore example

```
struct semaphore my_sem = MUTEX;

asmlinkage int sys_my_syscall(struct my_struct *userdata)
{
    int errno = 0;
    struct my_struct mydata;

    if (down_interruptible(&my_sem))
        return -ERESTARTSYS;

    if (copy_from_user(userdata, &mydata,
                       sizeof (struct my_struct)))
        errno = -EFAULT;

    ... do something with data ...

    up(&my_sem);
    return errno;
}
```

Wait queues

- Wait queues are used to make a task sleep until a certain event happens, eg. device receives data and an interrupt is generated
- Exist as both interruptible and non-interruptible
- Used with poll() VFS function call

```
struct wait_queue *myqueue = NULL;

void wait_on_my_queue_inside_kernel(){
    interruptible_sleep_on(&myqueue);
}

void my_event_notify(){
    wake_up_interruptible(&myqueue);
}
    /* poll() example */
unsigned int my_poll(struct my_device *dev,
                     struct file *file, poll_table *wait){
    int mask = 0;
    poll_wait(file, &myqueue, wait);
    if (some_condition)
        mask |= (POLLIN | POLLRDNORM);
    return mask;
}
```

System calls

- System calls are defined by a unique syscall number in include/asm/unistd.h
- A pointer to the system call function is put at the corresponding position in sys_call_table normally found in arch/<arch>/kernel/entry.S
- API defined between user and kernel space

```
arch/i386/kernel/entry.S:
ENTRY(sys_call_table)
 .long SYMBOL_NAME(sys_ni_syscall)
 .long SYMBOL_NAME(sys_exit)
 .long SYMBOL_NAME(sys_fork)

include/asm-i386/unistd.h:
#define __NR_exit    1
#define __NR_fork    2

arch/i386/kernel/process.c:
asmlinkage int sys_fork(struct pt_regs regs)
{
    return do_fork(SIGCHLD, regs.esp, &regs);
}
```

Privileges

- Linux provides POSIX (draft) defined capabilities support
- Used to determine whether a user proces is authorized to perform a task in the kernel, (eg. reboot or set the address on a network interface)
- See include/linux/capability.h for capabilities

```
asmlinkage int sys_reboot(int magic1, int magic2,
                          int cmd, void * arg)
{
    char buffer[256];

    /* We only trust the superuser with rebooting the system. */
    if (!capable(CAP_SYS_BOOT))
        return -EPERM;
```

Kernel threads

- Kernel threads are 'ordinary' processes running in kernel space
- Scheduled as normal processes
- Use for things that must be run regularly (daemon type tasks)
- Can run system calls as normal user space processes
- Can run executables from disk with **execve()** and the thread then becomes a normal user process

ioctl's

- ioctl's are generally used to get/set configuration and get info about general system settings
- <u>slow</u>, not for performance critical code
- registered by a device driver, filesystem etc.
- Uses unique identifiers (ex. SIOSIFFLAGS)

```
int my_ioctl(struct device *dev, struct ifreq *rq, int cmd){
    int error = -EOPNOTSUPP;

    switch(cmd){
    case SIOCDST:
        ... do something ...
        if (ok)
            error = 0;
        break;
    case SIOCDSTE:
        ... do something else ....
        if (ok)
            error = 0;
        break;
    }
    return error;
}
```

sysctl

- sysctl is an interface to make tuning parameters run time configurable
- Make parameters sysctl configurable if possible - compile time options are evil!
- Examples of sysctl parameters:
  - Socket buffer sizes (default+max)
  - Max shared mem
  - VM settings (page cache tuning etc).
- Entries registered in **kernel/sysctl.c, include/linux/sysctl.h** for defines
- Default handlers for strings and integer lists (read-only and read-write)
- Special cases covered with own handlers

sysctl example

```
int bdflush_min[N_PARAM] = {0, 10, 5, 25, 0, 1*HZ, 1*HZ, 1, 1};
int bdflush_max[N_PARAM] = {100,5000,2000,2000,100,600*HZ,
                            600*HZ,2047, 5};
int bdf_prm = {40, 500, 64, 256, 15, 30*HZ, 5*HZ, 1884, 2};

static ctl_table vm_table[] = {
    {VM_FREEPG, "freepages",
    &freepages, sizeof(freepages_t),
    0644, NULL, &proc_dointvec},

    {VM_BDFLUSH, "bdflush",
    &bdf_prm, 9*sizeof(int),
    0600, NULL, &proc_dointvec_minmax,
    &sysctl_intvec, NULL,
    &bdflush_min, &bdflush_max},

    {0}
};
```

/proc

- /proc is a virtual filesystem
  - provides kernel information to user space
  - good alternative to ioctl's for getting general system info
  - can be used to set certain system settings
  - /proc/sys interface to sysctl (also accessible via sysctl binary)
- /proc entries are registered dynamically by kernel modules - see **fs/proc/root.c**, **fs/proc/array.c** and **include/linux/proc_fs.h**

```
static struct proc_dir_entry proc_root_partitions = {
    PROC_PARTITIONS, 10, "partitions",
    S_IFREG | S_IRUGO, 1, 0, 0,
    0, &proc_array_inode_operations
};

void proc_root_init(){
    proc_base_init();
    proc_register(&proc_root, &proc_root_loadavg);
    proc_register(&proc_root, &proc_root_partitions);
    ....
```

/proc output function

- output function called from fs/proc/array.c

```
static long get_root_array(char * page, int type, char **start,
                           off_t offset, unsigned long length){
    switch (type) {
    case PROC_PARTITIONS:
        return get_partition_list(page);
    ....

int get_partition_list(char * page){
    struct gendisk *p;
    char buf[32];
    int n, len;

    len = sprintf(page, "major minor  #blocks  name\n\n");
    for (p = gendisk_head; p; p = p->next) {
        for (n=0; n < (p->nr_real << p->minor_shift); n++) {
            if (p->part[n].nr_sects && len < PAGE_SIZE - 80) {
                len += sprintf(page+len, "%4d  %4d %10d %s\n",
                p->major, n, p->sizes[n], disk_name(p, n, buf));
            }
        }
    }
    return len;
}
```

Multi architecture & portability

- Linux is a multi architecture OS
- Writing the code to be cross-architecture is very important
- Do not expect 'normal' data types to have a fixed size or byte order
- If data types need to match hardware expectations use fixed sized types
- Byte order conversion macros are available
- Not all archs guarantee cache coherency
- Not all archs guarantee write ordering
- Not all archs have their devices attached to the I/O ports of an ISA/PCI bus

Linux data types

bits/type	32 bit arch	64 bit arch
char	8	8
short	16	16
int	32	32
long	32	64
long long*	64	64

(* depreciated inside the kernel)

Name	Exported *	Size(bits)
s8 /u8	__s8 /__u8	8
s16/u16	__s16/__u16	16
s32/u32	__s32/__u32	32
s64/u64**	__s64/__u64	64

(* for export in header files
- no POSIX type conflict)
(** 64 bit architectures only)

Pointers and bit operations

- Cast/store pointers to/in unsigned long only!
- Bit operations always operate on longest native entity: unsigned long (ie. 32 or 64 bit)!
- Never use bit operations on char/short! (bitops on char * is a classic mistake)
- Atomic and SMP safe
  - **set_bit(int bitnr, void * addr)**
  - **clear_bit(int bitnr, void *addr)**
  - **test_bit(int bitnr, void *addr)**
  - **test_and_set_bit(int bitnr, void *addr)**

Kernel versions

- Some features are not available in all kernel versions, data structures may get new elements etc.
- To make a driver multi-version portable one can check the kernel version
- The kernel defines **LINUX_VERSION_CODE**, (major << 16) + (minor << 8) + release

```
#if (LINUX_VERSION_CODE > 0x20155)

#include <linux/init.h>

#else

#define __init
#define __initdata

#endif
```

Byte order handling

- To check for byte order:
  - **#ifdef __BIG_ENDIAN**
  - **#ifdef __LITTLE_ENDIAN**
- Byte order conversion:
  - **cpu_to_le{32,16}(), le{32,16}_to_cpu()**
  - **cpu_to_be{32,16}(), be{32,16}_to_cpu()**
- 64 bit versions available on 64 bit architectures only
- Classic network macros available as well **ntohl()**, **htonl()**, **ntohs()**, **htons()** - use these for networking only!

Cache coherency

- Some architectures are not cache coherent, and/or do not have data+instruction cache inter connected
- Explicit cache handling in software required:
  - Data segment to be accessed via DMA
- A code segment previously accessed as data by a CPU (copied with memcpy() etc)
- Cache handling functions (virtual addresses):
  - **flush_page_to_ram(u_long pageaddr)**
  - **flush_icache_range(u_long start, u_long end)**

Write ordering

- Some archs do not guarantee write ordering (Alpha + PPC) and gcc may cache data in registers (even x86!)
- **mb()** explicitly ensures write ordering
- **volatile** != write ordering
- Wrong vs correct example:

```
static inline void write_wd33c93_cmd(wd33c93_regs *regp,
                                     unsigned char cmd)
{
    writeb(WD_COMMAND, &regp->SASR);
    writeb(cmd, &regp->SCMD);
}

static inline void write_wd33c93_cmd(wd33c93_regs *regp,
                                     unsigned char cmd)
{
    writeb(WD_COMMAND, &regp->SASR);
    mb();
    writeb(cmd, &regp->SCMD);
    mb();
}
```

SMP

- Two or more processors may try to access a data structure at the same time
- Control concurrent access to avoid race conditions and their dangerous results
- Avoid 'just' disabling all interrupts, it is slow as it requires synchronization across all processors!
- Alternative solutions:
  - atomic data types (**atomit_t** with atomic operations, use for counters etc.)
  - **xchg()** atomic swap of data at a given address with a new value - works for 8, 16, 32 (& 64 on 64 bit archs) data types
  - spin locks

spin locks

- Linux provides two versions of spin locks, a normal mutex lock and a read/write lock
- spin locks are fast but do busy wait!
- Use to restrict access to critical data structures, device registers etc.
- Usable in interrupt handlers
- Never call anything that may sleep or take a page fault while holding a spin lock
- Available with or without disabling interrupts on local processor (inside irq handler, the irq is not reenabled - use non-disable version):
  - **spin_lock()/spin_unlock()**
  - **spin_lock_irqsave()/spin_unlock_irqrestore()**

spin lock example

```
void ace_spin_init(struct ace_private *ap){
#ifdef __SMP__
    spin_lock_init(&ap->lock);
#endif
}

static void ace_interrupt(int irq, void *dev_id,
                          struct pt_regs *ptregs){
    struct ace_private *ap;
    struct ace_regs *regs;
    struct device *dev = (struct device *)dev_id;

    ap = (struct ace_private *)dev->priv;
    regs = ap->regs;

    spin_lock(&ap->lock);
    /*
     * In case of PCI shared interrupts or spurious interrupts,
     * we want to make sure it is actually our interrupt before
     * spending any time in here.
     */
    if (!(readl(&regs->HostCtrl) & IN_INT)){
        spin_unlock(&ap->lock);
        return;
    }
    ....
```

Data alignment

- Data types are not the same size on all archs
- Compiler does not perform the same level of padding in data structures on all archs
- Do not expect the compiler to align entries on four byte boundaries!
- Explicitly pad chars/shorts if it is required that the following entry is on a certain boundary
- Pack data in a sensible matter in data structures, ie. longs together with longs, shorts with shorts whenever possible

Data alignment (2)

- Data structures with elements accessed concurrently by two processors or CPU + device may cause cache line stalls
- If elements are independent (eg. indices for a TX and a RX ring for a network device) perform explicit cache line alignment

```
struct something {
    volatile u32 evt_prd
                  __attribute__ ((aligned (L1_CACHE_BYTES)));
    volatile u32 rx_ret_prd
                  __attribute__ ((aligned (L1_CACHE_BYTES)));
    volatile u32 tx_csm
                  __attribute__ ((aligned (L1_CACHE_BYTES)));
}
```

__initdata

- Data structures and functions that are only used at boot time can be marked as **__initdata** and **__init** functions.
- Will be put in a special ELF section and added to the pool of free memory after the kernel has booted
- Good for device driver probe functions and firmware data

```
static int probed __initdata = 0;

int __init some_device_probe (struct device *dev))
{
    if (probed)
        return -ENODEV;
    probed++;

    ....
}
```

OOPS

- The kernel produces a register + stack dump in case of a detected crash inside the kernel (NULL pointer access etc)
- When possible dumps will appear in syslog, otherwise only on the console
- Stack trace addresses can be converted to symbolic addresses using **ksymoops** and the symbol table generated by the kernel compile (**System.map**)
- Symbol tables for loadable modules can be obtained when the modules are loaded:
  - **insmod -m <module-name> > symfile**
- Keep your System.map!

OOPS example (1)

```
Unable to handle kernel NULL pointer dereference at virtual address 00000000
current->tss.cr3 = 073d1000, %cr3 = 073d1000
*pde = 00000000
Oops: 0002
CPU:    1
EIP:    0010:[<c01b2ff8>]
EFLAGS: 00010286
eax: c01b2fe8   ebx: c7f80000   ecx: c733df48   edx: c020c2cc
esi: c020c2cc   edi: c020c2cc   ebp: c8000000   esp: c733dedc
ds: 0018   es: 0018   ss: 0018
Process ifconfig (pid: 397, process nr: 29, stackpage=c733d000)
Stack: 00000000 00001043 ffffffff c015e83d c020c2cc 00001002 c020c2cc 00000000
       c015f386 c020c2cc c7ad7ae0 00001043 c7ad7b04 c733df58 c017a950 c020c2cc
       00001043 00008914 00008914 bffffb34 c7570b80 00001043 00000000 00000001
Call Trace: [<c015e83d>] [<c015f386>] [<c017a950>] [<c017c7cb>] [<c015b121>]
Code: c7 05 00 00 00 00 42 00 00 00 83 bb c4 76 01 00 00 75 1d 8b
```

OOPS example (2)

```
[jes@hippie linux]$ ksymoops -m /boot/Symbol.map-2.2.3 < oops-file

Unable to handle kernel NULL pointer dereference at virtual address 00000000
current->tss.cr3 = 073d1000, %cr3 = 073d1000
*pde = 00000000
Oops: 0002
CPU:    1
EIP:    0010:[<c01b2ff8>]
EFLAGS: 00010286
eax: c01b2fe8   ebx: c7f80000   ecx: c733df48   edx: c020c2cc
esi: c020c2cc   edi: c020c2cc   ebp: c8000000   esp: c733dedc
ds: 0018   es: 0018   ss: 0018
Process ifconfig (pid: 397, process nr: 29, stackpage=c733d000)
Stack: 00000000 00001043 ffffffff c015e83d c020c2cc 00001002 c020c2cc 00000000
       c015f386 c020c2cc c7ad7ae0 00001043 c7ad7b04 c733df58 c017a950 c020c2cc
       00001043 00008914 00008914 bffffb34 c7570b80 00001043 00000000 00000001
Call Trace: [<c015e83d>] [<c015f386>] [<c017a950>] [<c017c7cb>] [<c015b121>] [<c012f910>]
Code: c7 05 00 00 00 00 42 00 00 00 83 bb c4 76 01 00 00 75 1d 8b

>>EIP: c01b2ff8 <ace_open+10/130>
Trace: c015e83d <dev_open+25/80>
Trace: c015f386 <dev_change_flags+4a/f8>
Trace: c017a950 <devinet_ioctl+244/544>
Trace: c017c7cb <inet_ioctl+33f/3a0>
Trace: c015b121 <sock_ioctl+21/28>
Trace: c012f910 <sys_ioctl+14c/178>
Trace: c0108c00 <system_call+34/38>
Code:  c01b2ff8 <ace_open+10/130>  00000000 <_EIP>: <===
Code:  c01b2ff8 <ace_open+10/130>     0: c7 05 00 00 00 00  movl $0x42,0x0 <===
Code:  c01b2ffd <ace_open+15/130>     5: 00 42 00 00 00
Code:  c01b3002 <ace_open+1a/130>     a: 83 bb c4 76 01  cmpl $0x0,0x176c4(%ebx)
Code:  c01b3007 <ace_open+1f/130>     f: 00 00
Code:  c01b3009 <ace_open+21/130>    11: 75 1d           jne   c01b3028 <ace_open+40/130>
Code:  c01b300b <ace_open+23/130>    13: 8b 00           movl (%eax),%eax
```

Changes in 2.3.x

- Mainly the declaration of sempahores and wait queues
  - semaphores:

```
Before:
    struct semaphore sem = MUTEX_LOCKED;
    somestruct->sem = MUTEX;

Now:
    DECLARE_MUTEX_LOCKED(sem);
    init_MUTEX(&somestruct->sem);
```

  - wait queues:

```
Before:
    struct wait_queue wait = {current, NULL};
    struct wait_queue *xdc_wait = NULL;
    init_waitqueue(&thread->wqueue);

Now:
    DECLARE_WAITQUEUE(wait, current);
    DECLARE_WAIT_QUEUE_HEAD(xdc_wait);
    init_waitqueue_head(&thread->wqueue);
```

Other sources of info

- The **Documentation**/ directory contains a lot of valuable information
- Linux Documentation Project (LDP) http://metalab.unc.edu/LDP/
- The Linux Kernel, David A Rusling, 1996-1999,
  - ftp://metalab.unc.edu
  - /pub/Linux/docs/linux-doc-project/linux-kernel/
- Linux Kernel Internals, 2nd. edition, Beck et al. Addison Wesley 1998
- Linux Device Drivers, Alessandro Rubini, O'Reilly & Associates Inc. 1998
- Use the source Luke

FreeS/WAN - IPSec for Linux

Richard G. Briggs
FreeS/WAN
rgb@conscoop.ottawa.on.ca

FreeS/WAN IPSec for Linux

---

or, "How did I get here?"

Ottawa Linux Symposium
07/24/99

Richard Guy Briggs

http://xs4all.nl/~freeswan/
rgb@conscoop.ottawa.on.ca

Agenda

- Levels of Security
- Authentication vs. Encryption and Denial of Service
- Network Shapes
- Tunnel vs. Transport Mode
- System Layers
- Setup/Configuration
- Challenges/Solutions
- Conclusions
- Other Sources of Information

Levels of Security

Cyberspace	Paper Equivalent
Internet	Postcard
PGP	Sealed Letter
SSH/SSL	Daily Corporate FedEx
IPSEC	Inter-City Armoured Brinks Truck

Authentication vs. Encryption

E-Communication		Analogy to Sex	
Authentication	**Encryption**	**Who**	**Where**
none	none	stranger	public
yes	none	spouse	public
none	yes	stranger	home
yes	yes	spouse	home

Denial of Service

- If you are being actively distracted, you won't be very (re)productive

Network Shapes

- Classical VPN Tunnel
- Road Warrior Tunnel
- Host-to-Host
  - Transport or Tunnel mode
- Extruded Subnet Tunnel
- Masqueraded Subnet Tunnel

Tunnel vs. Transport Mode

- Transport Mode is the way most packets cross the internet:
  - [eth [ip [tcp/udp [data]]]]
  - [eth [ip [esp [tcp/udp [data]]]]]
  - [eth [ip [esp [xxxxxxxxxxxx]]]]

- Tunnel Mode is the way Mobile-IP works and the way VPNs are constructed
  - [eth [ip-in [tcp/udp [data]]]]
  - [eth [ip-out [ip-in [tcp/udp [data]]]]
  - [eth [ip-out [esp [ip-in [tcp/udp [data]]]]]]
  - [eth [ip-out [esp [xxxxxxxxxxxxxxxxxx]]]]

System Layers

Function	Layer
Who are we talking to?	DNS Secure
Set up keys	IKE/ISAKMP
Protect information	IPSEC

Setup/Configuration

1. Unpack the distribution
2. Read the Documentation
3. Get a kernel (2.0.37 or 2.2.10)
4. Compile it
5. Make it work (this is actually important!)
6. Verify functionality of the network
7. Compile/install utilities and patches
8. Configure/install kernel
9. Setup run-time configuration
   - addresses (hosts and networks)
   - keys (manual or automatic)

Challenges/Solutions of KLIPS(1)

- Where do we 'jack in' to the net code?
- How do I deal with a stalled transmitter?
- How do we get packets out of here?
- What if my network is not ethernet?
- How do I recover from Oopses?
- How do I debug kernel modules?
- `/proc` filesystem only hands you <4k page
- Don't use GFP_KERNEL from interrupt code
- How do I tell the kernel config my defaults?
- What is the easiest way to manage 14 transforms?

Where do we 'jack in' to the net code?

- Firewall code
- Virtual devices
- IP-Chains

How do I deal with a stalled transmitter?

- Don't use the device's `dev->hard_start_xmit()` function
- Use `dev_start_xmit()` instead

How do we get packets out of here?

- Initially, we could only send transport mode packets to local networks
- Using "Stoopid Routing Tricks"[sic](tm), we were able to extend 1.0 further than we intended

What if my network is not ethernet?

- The FreeS/WAN was originally set up for a fixed `hard_header_len` of 14, ethernet
- It is now dynamic and will deal with any well-behaved interface with a predictable `hard_header_len`
- Known problems with some buggy PPP

How do I recover from Oopses?

- Problem: Buggy code caused an oops, preventing access to the module, only solution: reboot
- Solution: Clear the device open map on unload in the `netlink` code

How do I debug kernel modules?

- `printk()` is your friend, use it with appropriate `syslog` priority levels
- compile-time debug switches
- run-time debug switches via custom interfaces such as `netlink`, which we are already using
- `sysctl` variables, accessible via `/proc/sys/*`

`/proc` fs only hands you <4k page

- *really* bad things happen if you write more than 4kB to the buffer that it hands you
- keep track of how much has been written to the buffer and return when it is full
- it has an offset variable to remember where to continue

use `GFP_ATOMIC` from interrupts

- use `GFP_KERNEL` for `kmalloc()`s in code called from user space
- use `GFP_ATOMIC` for `kmalloc()`s in code called from interrupts such as incoming and outgoing socket buffers

How to set kernel config defaults?

- patch or append to `arch/*/defconfig` if it is a fresh kernel, never been configured
- append to `.config` if it has, because the above won't get looked at again

What is the easiest way to manage 14 transforms?

- don't, you will end up fixing the same bug 14 times
- rearrange the code to manage combinations of 5 encryption and 5 authentication algorithms with one path through

Challenges/Solutions of KLIPS(2)

- Finding your way around the Linux kernel - etags/ctags, cflow
- Unloading modules, static linked KLIPS
- Deleting SAs, deleting eroutes
- Memleak-deluxe package for detecting kernel memory leaks
- Route-stealing -- internal loop
- Bypass for IKE (UDP/500)
- For Alpha, use %p instead of (uint)%d, use long instead of int
- IPIP tunnel dependance, built from it, include it
- What to do with oversize packets/with DF set? Send ICMP MTU *and* fragment
- PF_KEYv2 sockets
- Extending PF_KEYv2 or SDB? OpenBSD vs. IPChains
- 2.2.x kernels, fragmentation issues

Conclusions

Problems	Solutions
Privacy	Encryption
Modification, Replacement, Spoofing	Authentication
Trust	DNSSEC

- Proprietary O/Ss might be better documented, but not always correctly. This can be very frustrating.
- The Linux kernel is not that well documented yet, but it is getting there, and the source is the most authoritative.
- If something is broken, it can be fixed.

Other Sources of Information

- Author:
  - http://conscoop.ottawa.on.ca/rgb/
  - rgb@conscoop.ottawa.on.ca
- Project Motivation:
  - http://www.cygnus.com/~gnu/swan.html
- Source:
  - http://xs4all.nl/~freeswan/
  - http://www.flora.org/freeswan/

Other Sources of Information(2)

- IPSEC:
  - http://ietf.org/html.charters/ipsec-charter.html
  - RFC2401-2412, 2451, 1828, 1829, 2104, 2085
  - Internet Drafts: draft-ietf-ipsec-* and others
- Mailing lists:
  - ipsec@lists.tislabs.com
  - linux-ipsec@clinet.fi
- DNSSEC:
  - http://ietf.org/html.charters/dnssec-charter.html
  - RFC2065, 2137
  - Internet Drafts: draft-ietf-dnssec-* and others
- Mailing lists:
  - dns-security@lists.tislabs.com

Other Sources of Information (3)

- Cryptography:
  - Schneier, Applied Cryptography, Wiley & Assoc.
- Networking:
  - Kirch, Linux Network Administrator's Guide, O'Reilly and Associates (www.oreilly.com)
  - Stevens, TCP/IP Illustrated, Vols. I, II, III, Addison-Wesley
- Kernel hacking:
  - Rubini, Linux Device Drivers, O'Reilly and Associates (www.oreilly.com)
  - The Linux Kernel Hackers' Guide
  - The Linux Kernel
  - The Linux Kernel Module Programming Guide
  - Linux Kernel mailing list
  - The Linux kernel source itself

Linux-7k

Werner Almesberger
École Polytechnique Fédérale de Lausanne
Werner.Almesberger@epfl.ch

Linux-7k

Werner Almesberger
EPFL ICA

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

July 8, 1999

Why Linux ?!?

- Powerful and flexible tools
- Easy cross-development and cross-testing
- Seamless integration with Linux desktop
- Long-term platform stability
- Friendly user interface
- Hack value

Platforms

- Psion S5 (8 MB)
  Primary development platform
- Geofox One (8, 16 MB)
  Usually lags behind in a few features
- Psion S5mx/S5mx PRO (16/24 MB) **?**
  Apparently hardware very different from Psion S5
  Position of Psion plc. not clear yet

Linux-7k Who is Who

Some of the names you may see on the linux-7110 mailing list:

Alan Cox General things (isn't he everywhere ?)

Edward Bailey Got things rolling

Otfried Cheong "ArLo" boot loader

Philip Blundell ARM Linux, GCC/glibc

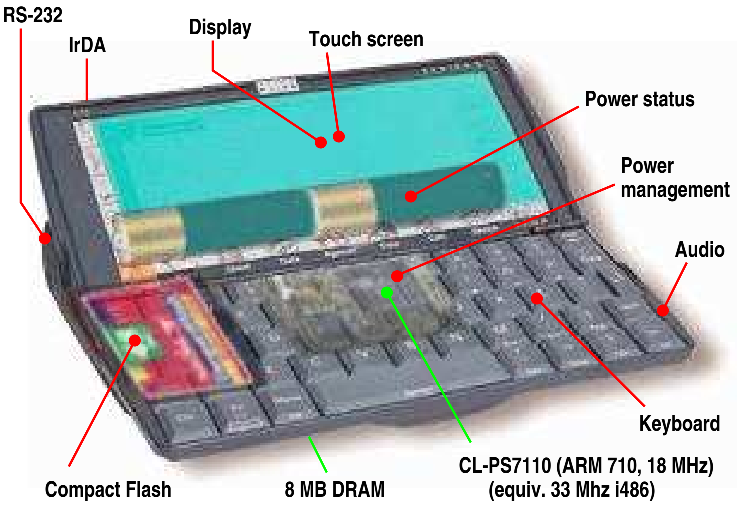
Roger Gammans Geofox; Keyboard; MICROWIRE

Russell King ARM Linux

Werner Almesberger Psion; General parts

... there are of course many others.

Hardware overview



History

March '98: Project begins

August '98: ArLo ready to boot kernels

October '98: Cross-development tools (egcs-1.1b, binutils-2.9.1.0.15)

October '98: First boot with serial console (2.1.125 kernel)

January '99: Keyboard; LCD incl. VT; CompactFlash (very slow); ADC

February '99: Upgrade to 2.2.1

March '99: fbcon; audio; sticky keys; power status

March '99: First field test: SLC'99

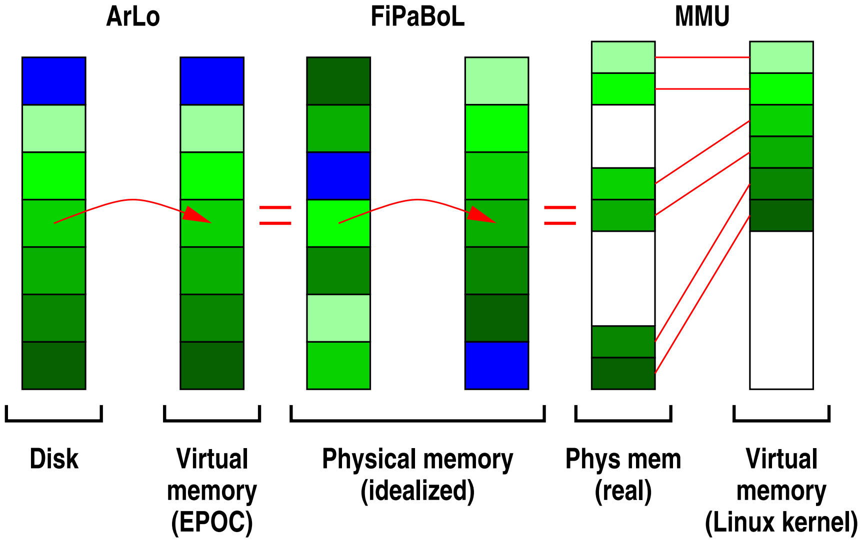
April '99: SLIP works; IR port; power management; Geofox touch pad

May '99: Psion touch screen; CF now at 700 kB/sec

May '99: Started to move things into 2.3

June '99: "Correct" support for CompactFlash; power management with standby

Memory



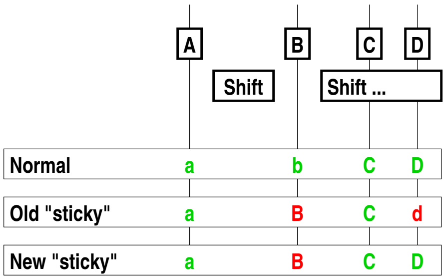
Boot process

- ArLo (runs under EPOC)
  - Load kernel and initrd into EPOC virtual memory
  - Set up map with physical page addresses
  - Switch to privileged CPU mode
  - Jump to first page
- Boot (is prepended to kernel)
  - Determine free memory
  - Move kernel to physically contiguous pages
  - Start kernel
- Kernel initialization (`head-armv.S`)
  - Set up page tables for kernel virtual memory
  - Activate MMU
  - Jump to `start_kernel`

Werner Almesberger Linux-7k 7

Keyboard

"Sticky" shift keys are needed for single-handed operation.



Architecture-specific keys:

- PDAs have many special keys, e.g. for power, backlight, contrast
- Semantics vary slightly among platforms
- Solution: add new key type for architecture-specific functions

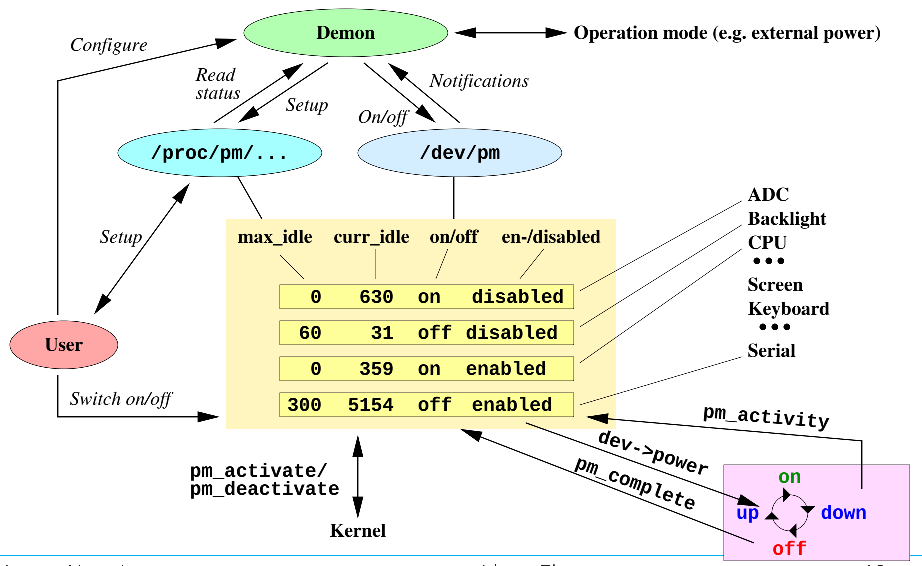
Werner Almesberger Linux-7k 8

Kernel-based PCMCIA

- PCMCIA package needs:
  - Kernel, initrd
  - `cardmgr`, a shell, etc.
  - libc
  - Recent modutils
- Problem: at the beginning, all we have is the kernel
- Kernel-based PCMCIA:
  - Card Information Structure (CIS) parser
  - A bit of architecture-specific glue
  - The IDE driver

Werner Almesberger Linux-7k 9

Power management



Werner Almesberger Linux-7k 10

What works well

- Architecture-independent kernel services (processes, memory management, file systems, networking, etc.)
- Keyboard
- Display (fbcon)
- Audio
- CompactFlash
- Generic power management

What needs work

- Psion: Touch screen still off by 1-2 millimeters
- ADC: sometimes locks up when power saving (still ?)
- MICROWIRE and ADC use delay loops (changing)
- Serial driver drops characters at higher speeds

Beyond the kernel

- Need up to date binary distribution for ARM3 (people working on RH 6)
- or: emulator for ARM4 instructions (status unknown)
- Installation procedure
- Windowing system
- GUI toolkit (GNOME ?)
- Organizer applications (agenda, phone book, Internet tools, etc.)

Future kernel work

- Integration into 2.3
  - Architecture-independent parts can go directly to Linus
  - ARM-specific parts go via Russell
  - Novel components (e.g. power management) go to `linux-kernel` first
- Fix various event mechanisms (still a few `udelay`s and even `mdelay`s)
- Many minor details (e.g. device numbers, file locations)
- Converge with other PDA projects on power management architecture

Related work

- Linux CE (Cassiopeia (SH3); Nino, Everex (MIPS))
  `http://www.linuxce.org/`
- μClinux (extended PalmPilot)
  `http://www.uclinux.org/`
- NetBSD/hpcmips (Everex, MIPS)
  `http://www.freebsd.org/%7Eimp/netbsd-hpcmips-en.html`
- Itsy
  `http://www.research.digital.com/wrl/itsy/`

Resources

- Project Web page:
  `http://www.calcaria.net/`
- This presentation:
  `ftp://lrcftp.epfl.ch/pub/people/almesber/slides/ols99-7k.ps.gz`
  `ftp://lrcftp.epfl.ch/pub/people/almesber/slides/ols99-7k.tar.gz`

Keynote Address: Brook's Law Revisited: Software Engineering Lore in a free software world

Alan Cox
Building #3

Abstract

Alan Cox has been working on Linux since 0.95. His first major project was the Linux Networking which he took over from Ross Biro. Alan maintained the Linux networking stack up to 2.0.30 when it was handed over to Dave Miller , Alexey Kuznetsov and others. During his time as network maintainer Alan wrote much of the original Linux protocol stacks for AX.25 Appletalk and IPX.

With hardware from Caldera Alan did the original Linux SMP port for Linux 2.0. Since then he has started the Linux Macintosh port, designed the Video4Linux layer and modularised the Linux sound support as well as writing several drivers.

Alan now maintains the 3c501 network card driver, the 2.0 kernel tree and works on maintaing and merging code into the Linux 2.2 tree.

Introduction by, Gavriel State, Corel Corp

Drinking from the firehose: can we optimize Linux to handle gigabit ethernet?

Michael C. Richardson
Sandelman Software Works Inc.
mcr@sandelman.ottawa.on.ca http://www.sandelman.ottawa.on.ca/

Abstract

As data speeds get higher and higher, while CPU and memory speeds are not increasing at quite the same rate it is becoming less and less practical to build firewalls, bridges, routers and even file servers out of generic hardware with general purpose operating systems. Unix workstations were long since replaced with by dedicated software routers, and the trend recently has been to replace most of the switching functions done in software by dedicated ASICs.

However, ASICs are hard for end users to change, and Linux has been very successful at providing a platform on which to build network devices. How can software only solutions keep up with gigabit Ethernet, OC48 and even higher speeds made possible via wave division multiplexing? A typical Linux box can not deal with 100Mb/s. The answer is indisputably smarter NIC cards. The disputed part of the answer is "how smart?" (and how expensive).

The Solidum PaxPort.100 is a 100Mb/s intelligent NIC card that offloads the CPU from sorting packets. A Linux driver has been produced and it will be discussed. Some experience debugging drivers that must move a lot of data will be related. More challenging still is our plan to further integrate packet classification technology into the Linux networking stack.

Gb+ Ethernet designs are in development.

Memory & File Management in the Linux Kernel

Pat Beirne
Corel Corporation

Abstract

Ever wonder how memory is given to applications? Where does it come from? Where does it go afterwards? This is tutorial for programmers working in and around the Linux kernel.

Topics include:

- memory management
- file mapping
- dynamic loading and linking
- the /dev/zero device
- inter-app security
- virtual memory and swap devices

These topics may sound diverse, but are bound together by a single concept: file mapping. By the end of the session, you should have a clear understanding of:

- how memory is allocated (kernel vs glibc management)
- the real time requirements for memory allocation
- how shared libraries are shared
- how memory is prepared for a task
- the subtleties of mmap()

This is an intermediate level tutorial. It is more of a ”this is how it works” rather than a ”howto”.

Some C experience is required.

The GNOME Mailer, utilizing the Bonobo Component Model

Miguel de Icaza
International Gnome Support

Abstract

The GNOME Mailer will support all major mail backend formats and protocols, will provide an intuitive and attractive user interface, and will support sophisticated message organization and threading technologies. The mailer will also integrate tightly with the other groupware components, including the contact manager and the calendar. Our intent is to beat Microsoft Outlook.

The GNOME Mailer is not a research project; it would be inappropriate for us to devise an entirely new design concept and wait to find out whether or not it works. Our design is based on a number of proven elements of other mail systems, including the DEC Pachyderm research project, Sun's JavaMail API definition, and Mozilla's Grendel mail program.

The design of the mailer is centered around the Bonobo component model. The GNOME Mailer will be built out of a number of self-contained well-defined components, each of which solves a very specific problem. These components are then glued together and collectively act as the mailer.

There are two main parts to the GNOME Mailer: The Mail Service Component and the user interface component, or frontend. The mailer's frontend communicates with the mail service across a process boundary via CORBA. The user interface is thus decoupled from the actual mail storage. Splitting the user interface from the mail storage makes it possible to plug in any number of mailer frontends. So, for example, it will be possible to outfit the mailer with a text-based interface, a web-based interface, or an IMAP proxy which would allow legacy mail clients access to the powerful semantics of our mail storage service, described in greater detail below.

The Mail Service Component handles access to the mail backends – the files and network servers which actually store the mail. Its interface provides methods for fetching mails and mail folders and for manipulating the backends. The Mail Service Component has a native backend which can store mail in a way particularly tailored to the GNOME Mailer. This is described more below.

The mail frontend, which includes the GUI, is made up of a number of components. One component, the Mail Summary View component, handles displaying mail folders and allowing the user to manipulate them. The Mail Message View component is designed to display a single mail message. To do this, it uses a large number of small components which are each good at displaying one particular type of mail object, be it text, attached HTML, an attached image, or things like business cards. These small components are referred to as Mail Object Components, and are embedded into the Mail Message View component using the Bonobo embedding facilities. Finally, the Mail Composition Component provides the GUI that is used to compose messages and send them. The Mail Composition Component communicates with the Mail Service Component to actually send the mail.

The job of the main GNOME Mailer program is to tie all of these components together in a coherent way. The main program handles component initialization and sets up the intercomponent communication

paths. This part of the mailer will be written in a widely-known scripting language which will be outfitted with CORBA bindings. Using a scripting language for the main program makes it easy for hobbyists to begin hacking on the mailer quickly, and to get a complete view of the mailer's design. The main program manages the behavior of the user interface, dispatching to the proper interface methods when the user triggers an event and manipulating the graphical display as appropriate. Controlling the UI behavior in a script makes it especially easy for hackers to tweak the user interface.

IP QoS (Quality of Service)

Jamal Hadi
Nortel Networks

Abstract

The tutorial will introduce, via a live demo, configuration and setup of a hidden gem in Linux 2.2.* networking – Linux Traffic Control. Linux TC was written by Alexey Kuznetsov. It provides administrators more control over who uses their bandwidth. If you have heard of the new hype term QoS – then TC provides the framework for QoS which is more flexible and powerful than any known OS in the market today.

Consider a Service Provider, who wishes to control which visitors to a web site are more important than others (free versus paying customers) and should get a relatively faster and better service. Before TC, this would have needed very expensive equipment. With TC, the control could be done within the server via a configuration script which could be activated via a crontab (bandwidth gets cheaper after midnight ;-¿).

The granularity of control is very powerful. For example, depending on the setup, it could be at the IP, TCP, UDP etc i.e control could be provided either by application (FTP gets less bandwidth than HTTP) or per IP address or ranges or a mixture of all these parameters.

TC truly takes Linux to new breaking ground unexplored by any other OS.