Internship at Snow B.V.
Redesigning FreeBSD's TTY layer

Ed Schouten

June 9, 2008

Thesis for Fontys University of Applied Sciences

Student:

Name:	Ed Schouten
Student ID:	2042669
Email address:	<ed@FreeBSD.org>
Course:	Embedded Systems
Period:	February 2008 to July 2008
Duration:	20 weeks

Employer:

Company:	Snow B.V.
Department:	Engineering
City:	Waardenburg, the Netherlands
Mentor:	J.P. Jansen, deputy director of engineering

School mentor:

Name:	P. Zwegers

Thesis:

Title:	Internship at Snow B.V. – Redesigning FreeBSD's TTY layer
Date:	June 9, 2008
Confidential:	no

Approval:

Signature from the company mentor:

Preface

The last five months I was given a second chance to work on the part of software engineering which interests me: UNIX kernel architecture. Like my first internship for my course on Embedded Systems was focused on making modifications to the network stack of the FreeBSD operating system, my dissertation was also all about data communication. During this internship I have been working on the serial data transfer framework called the TTY layer. Development was sponsored by the IT consultancy firm Snow B.V. in Waardenburg.

There are various people I would like to thank for supporting me during this project. First of all I would like to thank the people at Snow B.V., including my project mentor Jos Jansen and my project observer Joost Helberg for providing me this great opportunity to work on this assignment. I would like to thank my school mentor Patrick Zwegers for showing interest in the project. His feedback throughout the project has been of great use.

Last but not least I want to thank the developers and volunteers of the FreeBSD Project and the FreeBSD Foundation. With the help of Remko Lodder, Robert Watson and Philip Paeps, communication with the FreeBSD Project could not have been better.

Contents

Summary - English

During his dissertation for his course on Embedded Systems at Fontys University of Applied Sciences, Ed Schouten has been working at Snow B.V. in Waardenburg. Snow B.V. employs about 100 certified UNIX and network system experts, who are seconded to Top 500 companies in the Netherlands.

Snow B.V. has sponsored Ed Schouten to work on the Open Source FreeBSD operating system. Like most UNIX-like operating systems, FreeBSD has a kernel subsystem called the TTY layer. This subsystem is responsible for serving as a front-end for devices that provide serial line communication. Because this layer also integrates seamlessly with the UNIX process model, TTY's are often used to provide interactive user login sessions.

Even though the FreeBSD project was only founded in the 1990s, its TTY layer is at least 25 years old and has become unmaintained. This means it misses a lot of modern features that are already present in other subsystems throughout the kernel, such as scalability on multiprocessor (SMP) systems, low latency and hotplugging.

During this project a successful attempt has been made to rewrite the TTY layer that is part of the latest FreeBSD development code. The new implementation provides modern features. Because this TTY layer supports fine-grained locking, it is expected that its performance will now scale better on multiprocessor systems. The buffers that contain the data transmit and receive queues have also been changed to perform less copying of data. This means data throughput is also expected to be improved.

An important aspect of this assignment was to establish a good relationship with the FreeBSD project. Because the new implementation would be granted to the project, it was very imporant that it also matched the wishes of the FreeBSD project. During the internship a visit has been brought to the yearly BSDCan conference in Ottawa, Canada. During this conference Ed has given a presentation on the status of the new TTY layer, which was well received among the FreeBSD developers.

If further development goes according to plan, the new TTY layer will be part of the next major FreeBSD release, version 8.0.

Summary - Dutch

Tijdens zijn afstuderen voor de opleiding Technische Informatica op Fontys Hogescholen, heeft Ed Schouten stage gelopen bij Snow B.V. in Waardenburg. Snow B.V. heeft ongeveer 100 gecertificeerde UNIX en netwerk experts in dienst, die gedetacheerd zijn bij Top 500 bedrijven in Nederland.

Snow B.V. heeft Ed Schouten gesponsord om te werken aan het Open Source besturingssysteem FreeBSD. Net als de meeste UNIX achtige besturingssystemen, heeft FreeBSD een subsysteem in de kernel die de TTY laag genoemd wordt. Deze laag van de kernel is verantwoordelijk voor het dienen als abstractie bovenop apparaten die seriële communicatie verschaffen. Omdat deze laag naadloos integreert met het UNIX procesmodel, worden TTY's vaak gebruikt om interactieve login sessies te verschaffen voor gebruikers.

Ook al bestaat het FreeBSD project pas sinds de jaren '90, de TTY laag van het besturingssysteem is al minstens 25 jaar oud, waardoor deze niet meer actief onderhouden wordt. Dit betekent dat veel moderne verbeteringen die in andere subsystemen van de kernel geïmplementeerd zijn, niet aanwezig zijn binnen de TTY laag. Enkele voorbeelden hiervan zijn ondersteuning voor multiprocessorsystemen (SMP), optimalisaties voor lagere tijdsvertraging en de mogelijkheid om fysieke apparaten veilig te ontkoppelen.

Tijdens dit project is een succesvolle poging ondernomen om het TTY subsysteem van de laatste ontwikkelversie van FreeBSD te herschrijven. De nieuwe implementatie ondersteunt moderne functionaliteit. Omdat de nieuwe implementatie met hoge precisie gebruik maakt van synchronisatieprimitieven, wordt verwacht dat de prestaties op multiprocessorsystemen verbeterd zijn. Ook zijn de buffers die inkomende en uitgaande data bewaren verbeterd om minder data te kopiëren. Dit betekent dat de snelheid waarmee data verwerkt kan worden verhoogd is.

Een belangrijk aspect van de opdracht was om een goede band op te bouwen met het FreeBSD project. Omdat de nieuwe implementatie uiteindelijk aan het project beschikbaar gesteld zal worden, was het van groot belang dat het voldeed aan de wensen van het project. Tijdens deze stage is een bezoek gebracht aan de jaarlijkse BSDCan conferentie in Ottawa, Canada. Tijdens deze conferentie heeft Ed een presentatie gegeven over de voortgang van de nieuwe TTY implementatie. Deze presentatie is goed ontvangen onder de ontwikkelaars.

Wanneer verdere ontwikkelingen volgens plan gaan, zal de nieuwe TTY implementatie onderdeel uitmaken van de volgende grote FreeBSD release, versie 8.0.

Glossary

BSD license The BSD license is a Free software license. In its most simple form (often called the FreeBSD license), it only requires copyright notices to be left in the source code. When the software is only distributed in binary form, the documentation must display the copyright notice. The author of the code is in no way responsible for any damage.

devfs See *Device file system*.

Device file system On UNIX like operating systems, interaction with physical or virtual devices can take place by opening a special type of files called *device nodes*, usually stored in `/dev`. When opened, the kernel looks up the corresponding device by the node's *major* and *minor* numbers. On FreeBSD, `/dev` is a special file system whose contents are generated by the kernel, thus only showing devices that are actually available.

FreeBSD FreeBSD is a Free and Open Source (F/OSS) operating system. Together with Apple's Mac OS X – which also uses a lot of source code of the FreeBSD Project – it is the most commonly used 4.4BSD-Lite2 derived operating system.

GNU GPL The GNU General Public License is a Free, but somewhat restrictive software license. Unlike shareware and freeware software, tools covered by the GNU GPL will also include their source code, which allows modification and redistribution to the public. The GNU GPL requires the source code of applications to be included with its binaries.

Jail The FreeBSD kernel supports a mechanism called Jails. With Jails, a single FreeBSD kernel can run multiple instances of the FreeBSD Operating System. Like `chroot()`, it changes the root directory of a process. Because all the kernel's administrative interfaces cannot be used inside Jails, it is safe to allow processes to run as the root user inside the Jail without compromising the host system.

Kernel An operating system's kernel is responsible for scheduling tasks (processes) by performing time sharing. The FreeBSD kernel is monolithic, which means services like file systems, networking and drivers are run inside the kernel's address space.

Kernel space Kernel space refers to the memory region which normally cannot be addressed by user processes. This memory is used to store the kernel's code and data structures.

Mutex Similar to a binary semaphore, a mutex provides mutual exclusion to resources or data structures. Because a mutex provides the notion of an owning thread, techniques like priority propagation may improve the fairness of the process scheduler.

POSIX Because of the vast amount of different UNIX-like operating systems, the Portable System Interfaces for UNIX (POSIX) standard describes many core interfaces of the operating system, which vendors may choose to adhere. The latest edition of POSIX has been released in 2004. This release also includes new system library interfaces to interact with pseudo-terminals.

Pseudo-TTY Unlike regular TTY devices, pseudo-terminals are not connected to a physical device. Its input and output are exposed through a separate file descriptor. Pseudo-TTY's are used by terminal emulators (xterm, screen) and network login services (telnet, OpenSSH).

Semaphore Semaphores are used to temporarily prevent access to resources that are shared between multiple processes. A semaphore can be seen as a counter which can be modified by `Verhoog()` and `Prolaag()` actions. The `Verhoog()` action increments the counter, while `Prolaag()` tries to decrement the counter. When the counter has a value of 0, calls to `Prolaag()` will block until another process calls `Verhoog()`.

Signal Signals provide a way to asynchronously interrupt a running process on a UNIX system. Each process can install signal handlers, which are called when other processes or kernel services raise signals. Often used signals include `SIGTERM` which is called when a graceful shutdown of a process is requested, `SIGKILL` which forces a process to quit and `SIGSEGV` which is called when a process tries to access an invalid memory region.

Teletype Teletype writers were electromagnetic typewriters that were capable of transmitting and receiving data through a serial communications line. On UNIX like operating systems, TTY's are devices that allow users to perform interactive login sessions.

TTY See *Teletype*.

User space User space refers to the memory region which is used by a user process. Each process has its own unique address space, which means a virtual address in a process may not refer to the same physical memory when used inside another process.

Chapter 1

Introduction

The last couple of years we've seen big changes to desktop and server computer hardware. Where it had been a ritual for at least 30 years to store all computer processing logic in a single Central Processing Unit (CPU), it seems processor vendors are not capable of growing its processing power at a sustainable rate. Many computers nowadays ship with multiple CPU's. In some cases it is implemented by placing multiple processors on the motherboard, while in other cases it is performed by placing multiple CPU's on the same package ('Multi Core'). It is even possible to combine these techniques.

Just like any other operating system, the developers of the FreeBSD Project have already added support for Symmetrical Multiprocessor (SMP) systems. Around the year 2000, work started on adding SMP scalability to the operating system kernel itself, making it possible to implement concurrency inside the kernel's subsystems.

In the latest FreeBSD release, most subsystems of the kernel already provide SMP scalability. The TTY layer which provides support for serial communication still has to be converted. During the internship at Snow B.V. the TTY layer has been rewritten to provide better performance and scalability.

The second chapter of this document gives a background on Snow B.V. Chapter three describes the assignment in more detail and how it was executed. The fourth chapter contains a conclusion and recommendations.

Chapter 2

The company

The internship which is covered in this document was performend and executed at Snow B.V. This chapter will describe what Snow B.V. does and how it is organised internally.

Snow B.V. is specialised in UNIX system administration. Snow provides highly skilled and certified UNIX, NetApp and Cisco specialists, who are mainly seconded to Top 500 companies in the Netherlands.

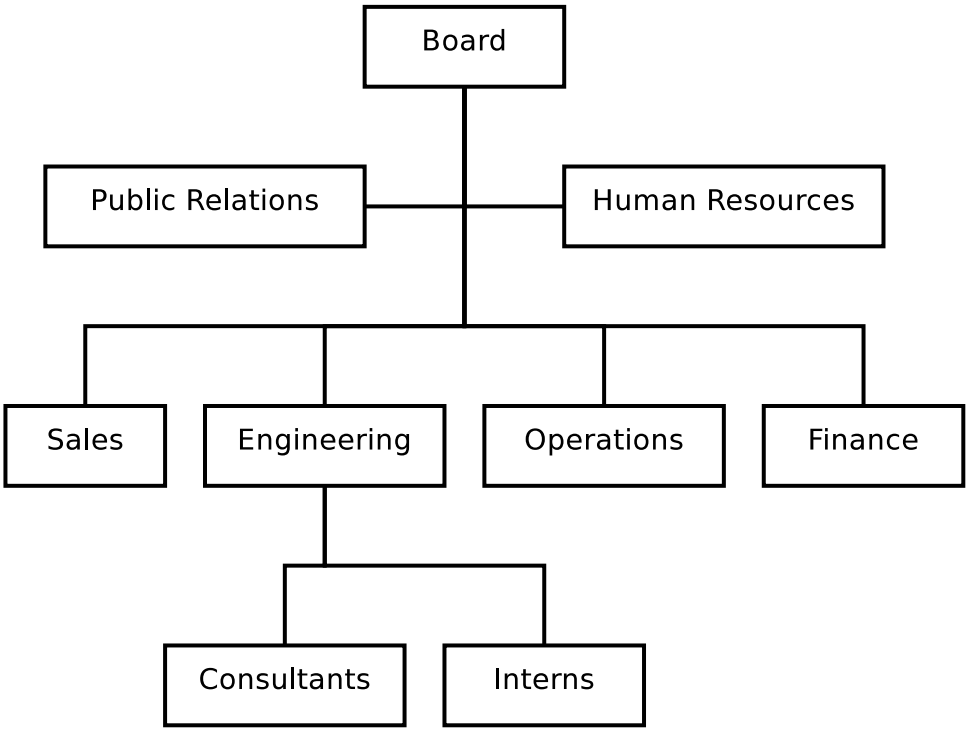
Snow was founded in 1997 by Hans Oey and Joost Helberg. It is named after the Snow Linux distribution, which they developed together during a snowy winter. Hans Oey and Joost Helberg were already active in the UNIX scene before they founded Snow. Hans has been the chairman of the Hobby Computer Club (HCC), while Joost has been the chairman of Vereniging Open Source Nederland (VOSN).

Snow's office is located in Waardenburg near Zaltbommel, though there are plans to move the office to Geldermalsen in July this year. Snow currently employs about 120 people. Because the employees of the engineering department are seconded to other companies, 20 people work at the office in Waardenburg.

The company is divided in the following departments:

- **Board:** The board of directors of Snow B.V.
- **Engineering:** Consultants who work at external companies or on internal projects between jobs.
- **Finance:** Tracking revenues and expenses of the company.
- **Human Resources:** Communicating with the staff, but also recruiting new people.
- **Operations:** Administrative tasks to support Snow's workflow.
- **Public Relations:** Responsible for external publicity.
- **Sales:** Finding new customers, but also communicating with existing customers.

The next page contains an organisational chart of the departments.


Board
Public Relations
Human Resources
Sales
Engineering
Operations
Finance
Consultants
Interns

Chapter 3

The assignment

During my internship at Snow B.V. I've reimplemented the TTY layer that is part of FreeBSD's operating system kernel. This chapter will first describe what FreeBSD is and how TTY's work. Section 3.3 will explain how several important aspects of the FreeBSD kernel are implemented, which are required to understand the problem with the old TTY layer, which is described in section 3.4.

3.1 The FreeBSD Project

The FreeBSD Project is an online project with about 250 developers, which maintains its flagship product, the FreeBSD operating system. FreeBSD is a UNIX-like operating system, which is based on the 4.4BSD-Lite2 source code. It runs on a variety of hardware architectures, including x86 (both 32 and 64-bits), Sun SPARC64, Intel Itanium (ia64) and IBM and Motorola PowerPC.

The FreeBSD operating system is designed to be used on servers, but it could also be used on desktops. Because it shares its programming interface with operating systems like Linux, it is capable of running many pieces of Open Source software, including services like Apache HTTPD and MySQL, but also desktop software like X.Org, GNOME, KDE, software from the the Mozilla Project and OpenOffice.org.

Unlike Linux, the FreeBSD Project not only develops an operating system kernel, but a full operating system which includes many standard utilities. Optional software can be installed through the FreeBSD Ports tree, which contains over 16000 optional software packages.

FreeBSD is used by big service providers, including the Yahoo! and Ilse search engines. Because most of its source is BSD-style licensed, it is also an attractive platform for embedded systems. Unlike the GNU GPL (the license used by Linux and its utilities), the BSD license does not force vendors to ship the source code. Major vendors using FreeBSD on their embedded products are Cisco and Juniper.

FreeBSD's source code is often used inside other software projects. There is a lot of migration of source code between the other BSD projects (NetBSD, OpenBSD and DragonFlyBSD). Apple's Mac OS X also includes a lot of software from the FreeBSD operating system to implement its UNIX API.

During this internship, software will be developed for FreeBSD CURRENT. In FreeBSD's development model, CURRENT always refers to the latest unreleased source code. In 2009 the source code will be branched. After extensive testing, this code will be released as FreeBSD 8.0. This means the new TTY layer will only be available to testers until the 8.0 version has been released.

3.2 The TTY layer's functionality

Like any other UNIX-like operating system, FreeBSD's kernel also contains a subsystem which is often referred to as the TTY layer. TTY's are front-ends for devices that can perform serial communication. On a regular FreeBSD installation, the following devices are represented as TTY's:

- The standard Intel 16550A serial ports, which are often integrated on the PC motherboard.
- The kernel's console driver, which implements a VT100 terminal emulator for the VGA card and an AT or USB keyboard.
- Any USB-to-serial cables, including USB modems and phone cables.

Apart from these physical devices, there is a driver which provides virtual pseudo-terminals. These pseudo-terminals allow user processes to emulate a TTY device, which will be explained in more detail later on.

A TTY is a very special kind of device. It is not a simple front-end for data transmission for serial ports. It provides many small features that are needed to make interactive user logins work on the UNIX operating system.

Buffering

One of the basic features of a TTY, is its built-in buffering mechanism. To improve system performance, data transmission is performed asynchronously. This means it is possible to write the output of a program to the serial interface at once. If the amount of bytes that are queued stays below the kernel's TTY buffer size (often called the *output watermark*), the application will not wait for the transmission to be finished.

Input is also buffered. This means the kernel is capable of storing multiple bytes of input, which can then be read by performing a single system call.

Flow control

Apart from the watermarks that cause processes that use TTY's to block when output exceeds its threshold, users can manually perform flow control by using the Ctrl+S and Ctrl+Q buttons. When Ctrl+S is pressed, output to the terminal is stopped. When Ctrl+Q is pressed, output to the terminal is resumed again. This allows users to temporarily stop terminal output when the output is too fast.

Text processing

The ASCII character set defines certain characters that may need special treatment when being transmitted or received on a TTY device. An example is the way newlines or carriage returns should be processed. Consider the following textbook example of a "Hello, world!" application written in the C programming language:

```
#include <stdio.h>

int
main(int argc, char *argv[])
{
        printf("Hello, world!\nGoodbye, world!\n");
        return (0);
}
```

When processed by standard terminal hardware, its output should be as following:

```
Hello, world!
             Goodbye, world!
```

The newline character (`\n`) is used in this example to terminate the lines. This character will cause the terminal to move its cursor one row down, but will not cause the cursor to be reset to the first column on the display. When the terminal is configured to perform output processing and newline to carriage return-newline translation, it will translate the `\n` character to `\r\n`. This will cause the cursor to be set to the first column before it is moved to the next row, making the output look as expected:

```
Hello, world!
Goodbye, world!
```

The kernel can perform the following processing actions:

- Newline to carriage return-newline translation: translate `\n` to `\r\n`.
- Carriage return to newline translation: translate `\r` to `\n`.
- Tab expansion: translate `\t` to multiple spaces, where the amount of spaces aligns the cursor to eight columns.
- End-of-text discarding: discard Ctrl+D characters.
- Handle characters with bad parity: when a character is received with bad parity, it can be dropped or escaped. Escaping it allows a user process to handle the parity error.
- Echo characters that have been received. When enabled, the terminal will display characters that have been inserted.

Process integration and signalling

The TTY code also integrates with the UNIX process model. Each terminal may be connected to a session, which has a session leader. The session leader is a process which has a leading role in controlling the processes that use a TTY. The session leader is usually a command shell (like the Bourne shell or the C shell).

A TTY may also have an associated foreground process group. The foreground process group is a chain of processes which are allowed to write to a terminal device. When a process tries to interact with a terminal when it is not part of the foreground process group, it is suspended using the `SIGTTIN` and `SIGTTOU` signals.

The reason why this mechanism is in place, is to make sure the user of the terminal is capable of determining which process it is interacting with. If multiple processes were capable of reading input at the same time, it would be unspecified which process would actually receive the data.

The kernel will also send various signals to the foreground process group when certain characters are received. When the user presses Ctrl+C, the processes in the foreground process group will receive the `SIGINT` signal, which may cause the process to quit. The Ctrl+Z key will suspend execution of the foreground process group. This allows users to return back to the shell at any time.

When the session leader process shuts down, access to the terminal is revoked to all processes that are currently interacting with it. When this mechanism would not be in place, processes would be capable of using the terminal, even after the user has logged out and the login prompt is shown again.

Line editing

One of the features that make TTY's really light-weight, is its line editing functionality. When a process only needs an entire line of input (i.e. shell command, username, password), it can switch the terminal to canonical mode. When a process tries to read terminal input, it will wait until a carriage-return or end-of-file character has been received.

Because the kernel will also perform character echoing when instructed, the user process will fully block until a full sentence has been inserted. The kernel also implements special line editing commands like backspace, the kill character (Ctrl+U) which deletes the entire input and the word erasion character (Ctrl+W) which only deletes a single word of text.

Line disciplines

Apart from interactive login sessions, serial ports can also be connected to a modem to connect to an Internet Service Provider (ISP). The FreeBSD TTY layer supports an abstraction called line disciplines. A line discipline specifies how data should be processed on arrival or transmission. FreeBSD supports the following line disciplines:

- `TTYDISC`: The default line discipline, which implements the text processing, line editing and signalling functionality described in the previous sections.
- `PPPDISC`: PPP line discipline (used with modems). This line discipline will not return data to user space processes, but will expose itself as a network interface, making it possible to process modem traffic without copying data to user space. This line discipline is planned for removal, because a more flexible user space PPP implementation is also available.
- `SLIPDISC`: Similar to the PPP line discipline, the SLIP discipline implements an efficient way to transfer IP traffic through serial interfaces. This line discipline is planned for removal, because it has been reported being broken.

- `NETGRAPHDISC`: Netgraph network traffic node type. This line discipline is capable of interacting with the Netgraph kernel networking subsystem.

Pseudo-terminals

Because modern computer systems also implement features like terminal emulators and network login services, the kernel also provides pseudo-terminals. These pseudo-terminals can be allocated by calling the `posix_openpt()` routine. This routine returns a file descriptor to the *master device*. Any data that is written to the master device, will be seen as character input. Any output on the terminal can be read from this file descriptor.

Pseudo-terminals are thus similar to pipes which are also provided by the FreeBSD kernel, but still provide the functionality which is normally available to the TTY through the slave device.

Conclusion

As can be read in the previous sections, a UNIX kernel implements a great set of features to provide outside interaction with local processes. Because these features are implemented within the operating system kernel, processes often do not need to be executed when data is received on the serial interface. This causes interaction to the TTY layer to be responsive, even if the system is under high load.

3.3 FreeBSD kernel programming interfaces

This section will describe some of the programming interfaces that are present inside the FreeBSD kernel. It is important to know how certain interfaces inside the kernel have evolved to understand some of the problems that exist inside the TTY layer.

3.3.1 Device nodes in FreeBSD 4 and before

Like most BSD-like operating systems, FreeBSD 4 and earlier had two structures inside the kernel, called `bdevsw` (*block device switch*) and `cdevsw` (*character device switch*). These structures contained a set of driver routines for a certain device class. All devices on the system were identified by a pair of major and minor numbers. The major number referred to a unique number that was assigned to each device switch. The minor numbers could be freely used by device drivers.

On a typical UNIX-like system, the `/dev` directory is filled with device nodes. These device nodes can be created by running the following command:

```
mknod /dev/ad0 b 3 0
```

This command will create a block device named `/dev/ad0`, which uses 3,0 as its major and minor numbers. When this device is opened, the kernel locates the proper `bdevsw` structure which corresponds with major number 3. When available, it calls the `open()` routine that is stored inside the device class. A typical `open()` routine may have looked like this:

```
static struct mydevice *mydevicelist[10];

static void
mydevice_open(struct bdev *dev, ...)
{
        u_int u;

        u = minor2unit(minor(dev));
        if (u >= 10 || mydevicelist[u] == NULL)
                return (ENXIO);

        return (0);
}
```

This design gave a device driver much freedom in how to implement device names, but it had a few fundamental design problems:

- Each driver had to validate if a minor number really refers to an existing device.
- When not properly configured, a device node in `/dev` may not refer to the correct device inside the kernel, which may cause serious security problems, loss of data, etc.
- The contents of `/dev` do not actually reflect the devices that are actually present on the system.

Most operating systems supported a shellscript called `/dev/MAKEDEV`. This script could be run to regenerate the directory's contents.

3.3.2 Device nodes in FreeBSD 5 and after

To remove the limitations of the old major/minor number design, FreeBSD 5 was the first version to include an in-kernel device filesystem. A device filesystem is a special kernel-generated filesystem that is often automatically mounted on the `/dev` directory on startup.

Unlike the previous model, device drivers explicitly call routines named `make_dev()` and `destroy_dev()` to create and remove device nodes from the `/dev` directory. On a typical FreeBSD system, the `/dev` directory only contains about 100 device nodes. All these device nodes refer to devices that actually exist.

Because there is no need to keep the contents of the `/dev` synchronised, it is a lot easier to implement more complex naming schemes. When one would create a software RAID set called `storage`, the `/dev/mirror/storage` device node will automatically be created by the kernel. When a USB-to-serial cable is plugged in to the system, `/dev/ttyU0` will automatically appear.

Because the buffer cache (responsible for caching disk buffers) has been merged into the page cache (responsible for paging physical memory), there was no real reason to support block devices. As of FreeBSD 5, block devices have been removed from the system. This means all block oriented devices are now presented as character devices. Unlike block devices, these nodes don't have any requirements with respect to alignment.

3.3.3 Locking data structures in FreeBSD 4 and before

To understand some of the problems with the design of the traditional BSD kernel, this section describes how actions are synchronised within the kernel, to protect data structures against concurrent access.

FreeBSD 4 and earlier use a programming interface that also appeared inside classic UNIX operating systems, called `spl` (Set Priority Level). These routines allowed programmers to change the interrupt mask of the processor. The hardware which ran the first versions of UNIX (PDP and VAX hardware) had a priority-based interrupt mask. When the processor ran in priority 3, it would only allow interrupts to be processed that had a priority of 3 and higher. This means critical hardware such as timers had a very high priority, to make sure they could be serviced as soon as possible. Hardware that does not need these guarantees (serial ports, disks) ran at a lower priority.

Data structures within the kernel were protected using these routines. The following code is a snippet from FreeBSD's `slip` driver (IP through serial lines):

```
static int
slopen(struct cdev *dev, register struct tty *tp)
{
        int s;

        ...

        s = splnet();
        if_up(SL2IFP(sc));
        splx(s);
}
```

The `splnet()` routine raises the priority of the processor when its current priority is lower than the priority required by the network stack. When the `if_up()` routine is finished, the `splx()` routine changes the priority back to the previous level.

It goes without saying that this mechanism is not suited for system architectures which contain multiple processors. Even if interrupts would be disabled on the active processor, another processor could still end up inside the kernel at the same time.

However, FreeBSD 3 and 4 did implement support for multiprocessor systems. When a processor needed to execute code within the kernel (interrupt, system call), it first checked whether another processor was already running inside the kernel. When this was the case, it stalled until the other processor left the kernel. This only scales when processes are computation intensive; when both processes cause interrupts to be generated or perform system calls, contention will increase.

3.3.4 Locking data structures in FreeBSD 5 and after

FreeBSD 5 had a unique development process. It had one of the longest development cycles in the FreeBSD Project's history. One of its major changes, was the introduction of fine-grained locking.

Fine-grained locking means multiple processors can execute code within the kernel at the same time. Access to data structures is synchronised using mutexes. These mutexes come in two flavours: sleep mutexes and spin mutexes.

Sleep mutexes should be used as much as possible. These mutexes can only be used from a thread context and will cause other threads to be scheduled when contended. The FreeBSD kernel also supports interrupt threads, which means most interrupt handlers supplied by drivers already run inside a thread context. Sleep mutexes also provide priority propagation. This means that a thread holding a lock that is needed by another thread with a higher priority will temporarily borrow the higher priority until the mutex is unlocked, which reduces the system's latency, but also prevents resource starvation. It is not possible to interrupt execution of a thread while holding a lock, which means it is not possible to call functions like `malloc()`, which may need to sleep to reclaim pages of memory.

Spin mutexes should only be used when not operating inside a thread context (say, within a hardware interrupt context). Spin mutexes are only used in places where a sleep mutex is unsuited, like the process scheduler. Because spin mutexes don't operate within a thread context, there is no priority propagation. Spin mutexes should only be held for a very short amount of time. The kernel will automatically crash when the processor has to wait an excessive amount of time to lock a spin mutex.

The kernel also implements other locks, including semaphores and shared/exclusive locks. They are not discussed here, because they are beyond the scope of this project.

To summarise the previous paragraphs, the following table contains a comparison between both the mutex types:

	Sleep mutex	Spin mutex
Usable in process context	+	+
Usable in hardware interrupt context	-	+
Simplicity of implementation	-	+
Priority propagation	+	-
Honours thread priority	+	-
Allows hardware interrupts to be serviced	+	-
Can be held for an extended amount of time	+	-

To ease migration to this new locking model, a special sleep mutex is used inside the kernel, called the Giant. All code that has not been converted to the new locking model is covered by the Giant. Even though FreeBSD 5 included this new locking model, most subsystems used the Giant lock to prevent concurrent access. In FreeBSD 6, both the Virtual File System (VFS) and network stack were modified to include their own per-device and per-object locks. FreeBSD 7 included even more multiprocessor safe subsystems.

One disadvantage of this model is that programming errors might lead to deadlocks. The FreeBSD kernel can be configured to detect possible deadlocks by generating a graph of all lock classes within the kernel. This system is called Witness.

It is very important to migrate code away from the Giant lock. The Giant lock makes it practically impossible to scale the FreeBSD kernel to hardware architectures which have many processors.

3.4 Problems with the old implementation

Now that we've explained the basic concept of the FreeBSD kernel and its TTY layer, this chapter will describe the design problems that exist within the old TTY layer.

Buffering

A TTY layer uses a queue of bytes to store characters that have been received on the serial interface, but have not been read by processes. It also has a similar queue to store characters that have been written by processes, but have not been transmitted on the serial interface.

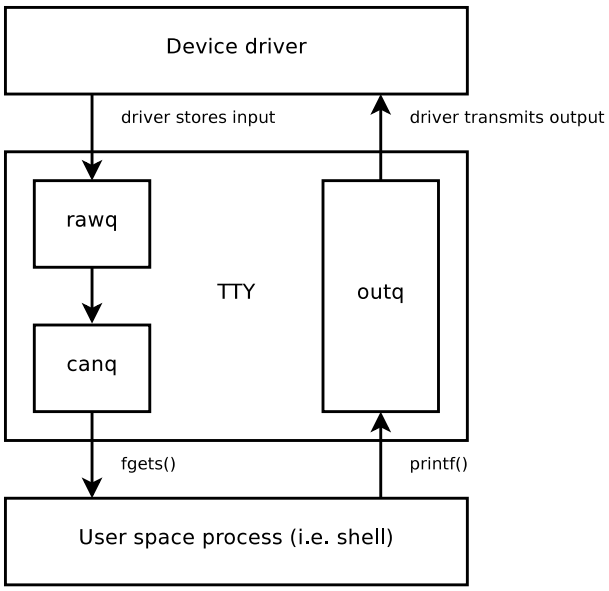
FreeBSD's TTY layer also has this byte queue, called *character lists*. Character lists are basically a linked list of buffer structures (*character blocks*). The character lists store the following data:

- Characters, each consuming 1 byte of space.
- Quoting bits, each consuming 1 bit of space. The quoting bits are used to mark characters with parity errors, literal processing, etc.

Each TTY has three character lists:

- *outq*: The output queue which stores the characters that have to be transmitted.
- *canq*: The input queue which stores lines of text that have already been terminated by a carriage return.
- *rawq*: The input queue which stores partial lines of text that have not yet been terminated by a carriage return.

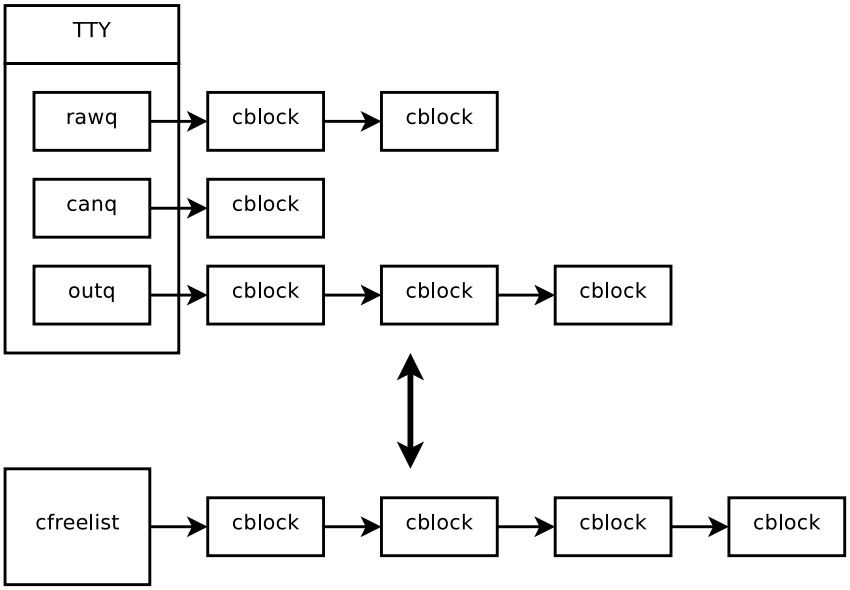
The following diagram shows how data flows through the TTY layer when the TTY is configured to operate in canonical mode. When canonical mode is disabled (often called 'raw mode'), the rawq will not be used.



The amount of buffer blocks associated with a character list depends on the baud rate of the device. When the baud rate is higher, data may arrive faster. To prevent loss of data, the list size scales linearly.

When the queue is empty or partially filled, unused blocks used inside the character list are stored on a global free list. When the baud rate changes, blocks are added to, or removed from the free list. This design is quite fragile. There are some known bugs, where a TTY may consume more blocks from the free list than it had allocated initially, thus stealing buffer space from other TTY's. The global free list does not scale well when there is a lot of activity on multiple TTY's at the same time. Each time a block would need to be allocated, a global lock would need to be acquired.

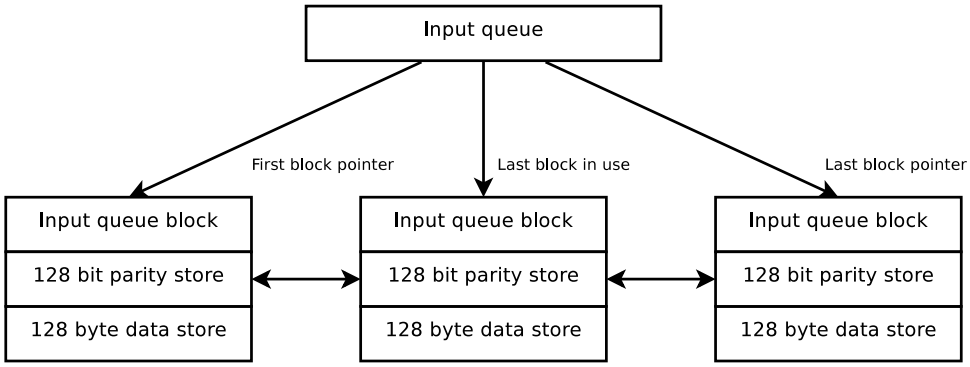
The following diagram shows an example of a TTY. As you can see, it still has some unused blocks that can be removed from the *cfreelist* when needed:



One way to prevent the use of a global free list, is to always store all allocated blocks in a single queue. The queue could then keep a reference to the last block in the queue which is actually used to store data. The list will hold at least three references to blocks and offsets within the queue:

- A reference to the first block in the list, which contains valid data.
- A reference to the block which should be used to store any new data. When the queue is empty, this reference points to the first block in the queue.
- A reference to the last block. When data is removed from the head of the queue, empty blocks will be moved to the end of the queue.

The next diagram shows a simple input queue which stores unused blocks inside the list structure:



The double input queue mechanism on input also has a design bug. Data often needs to be copied from the rawq to the canq. When a carriage return arrives, the line of input is terminated, which means it should be returned to the user. Below is a simplified version of the `catq()` routine, which concatenates the two queues together:

```
void
catq(struct clist *src, struct clist *dst)
{
        int chr;

        while ((chr = getc(src)) != -1)
                putc(chr, dst);
}
```

As you can see, the `catq()` routine performs the copying by reading a character at a time. This means it provides a linear $O(n)$ scalability, where n is the amount of characters that would need to be copied. This could easily be solved by merging the queues together, where the queue would provide the ability to set an additional reference where the last complete line ends. Changing this marker would be an $O(1)$ (constant time) operation. A constant time operation will reduce the latency of the kernel, making it perform better in real-time conditions.

Locking

FreeBSD's TTY layer is one of the last major components of the kernel that still uses the Giant lock exclusively. Most layers inside the kernel support per-instance flags to determine whether the Giant should be picked up, to provide a controlled phase-out. The network stack supports a flag called `IFF_NEEDSGIANT`. When set, the network stack will lock the Giant before calling into the network interface driver. The Virtual File system (VFS) supports the `MNTK_MPSAFE` flag to mark file system drivers which can operate without the Giant lock. The TTY layer does not support a similar construction. This has a couple of disadvantages:

- The kernel provides many drivers that are only exposed through the TTY layer, namely the console driver, input and serial port drivers. They cannot be made multiprocessor safe without removing the Giant lock from the TTY layer first.
- Other frameworks inside the kernel that call into the TTY layer (the USB and the network stack) still depend on the Giant lock indirectly.

The TTY layer should be able to use per-TTY locks. In the old TTY design, there are only two cases where an interlock should be required to synchronise between all TTY objects:

- The kernel keeps a list of all TTY's that are registered in the kernel. This list is needed to generate statistics. The `pstat` utility is capable of printing a table with terminal statistics.
- The old TTY character lists share one common free list.

When the character lists would be redesigned to include a per-character list free list, an interlock would only be required during construction and destruction of TTY objects.

Having per-TTY locks will likely improve performance on systems with many TTY's. A realistic example would be a shell server with many users. Many users could be logged in at the same time, or could be running jobs in the background using `screen`. When each terminal would have its own sleep mutex, threads could enter the TTY layer at the same time without blocking.

Hotplugging

Classic BSD systems used a list of statically defined TTY's. On startup, the devices were probed and attached and did not get removed from the system during its lifetime. Since the advent of pseudo-terminals and buses like USB, devices can be removed from the system.

FreeBSD does not safely handle hotplugging of TTY devices. Removing a USB-to-serial device while the TTY is in use, may cause the system to crash. Pseudo-terminals are never deallocated, but are recycled. Not being able to remove TTY's from the system has a couple of disadvantages:

- Even though TTY structures are not big, the associated input and output buffers may consume about 40 KB of memory.
- Pseudo-terminals don't completely reset their state when recycled, making the TTY's behave inconsistent when not initialized properly.

TTY's cannot be destroyed, because there is no reference counting. This means the TTY layer cannot determine whether the device is still being used.

Legacy features

One of the problems with UNIX-like operating systems in its entirety is that it's always possible to do the same thing in multiple ways. Even though the original UNIX was written around 1970, the first UNIX standard appeared in 1988. This caused many operating systems to become incompatible.

One of the examples of an incompatible programming interface within UNIX, is the interface that allows applications to change terminal attributes (baud rate, control characters, character processing). There are three different interfaces available:

- `sgtty`: The traditional UNIX 'set/get TTY' interface. It is still available inside BSD-style operating systems.
- `termio`: The System V Terminal I/O interface. This interface is not available inside FreeBSD. System V-like operating systems like Linux and Solaris still support this interface.

- `termios`: The Terminal I/O Standard interface, described in POSIX. All modern UNIX-like operating systems support this interface.

FreeBSD emulates the `sgtty` interface by converting it to `termios` calls. The interface is incomplete and should eventually be removed. The new terminal layer should at least think of a way to remove the interface from the kernel, making `termios` the exclusive interface for changing terminal attributes.

Conclusion

The previous sections described only some of the improvements that could be made to the TTY layer. Even though the TTY layer is not often seen as an important feature of a UNIX-like kernel, it is used quite often.

The Design Document (see appendix B) describes more improvements that could be made.

3.5 Communication

This section will describe which parties were involved during the execution of the assignment. This internship was somewhat special, because not only the employer and the school were involved, but also the FreeBSD Project.

Snow B.V.

During the project I mainly communicated with people at Snow B.V. There were two colleagues who were related to the project:

- Jos Jansen: customer and mentor.
- Joost Helberg: project observer.

All documents that were written during the project were sent to both persons. Jos Jansen provided feedback and approval to the documents. Joost Helberg was familiar with the TTY implementation of System V-like operating systems.

Fontys Hogescholen

Fontys also assigned me a mentor, who assisted me during my internship, Patrick Zwegers. During the internship, he paid two visits to the office to discuss the project plan, the thesis and the presentation. He also provided feedback to the documentation.

The FreeBSD Project

Because Snow B.V. allowed the new TTY layer to be contributed to the FreeBSD Project, it was very important that the new TTY layer would meet the standards that are required by the project. Already before work started at the office, contact had been made with the FreeBSD Project about the assignment.

Three developers of the FreeBSD project provided assistance during the project:

- Remko Lodder. Remko maintains the Dutch translation of the FreeBSD Handbook. He is also an employee of Snow B.V.
- Robert Watson. Robert is one of the members of the FreeBSD Core Team. The FreeBSD Core Team is responsible for determining the project's direction.
- Philip Paeps. Philip is a FreeBSD developer from Belgium. He became my mentor at the FreeBSD project after I became a developer myself.

The FreeBSD Project provided the following tools, which turned out to be of great use throughout the project:

- The project offered access to its Perforce Version Control System (VCS) to store the source code which was written during the project. This allowed interested parties to test the code during its development. Other developers were also capable of subscribing themselves to repository commit messages, making it possible to discuss any implementation related issues.
- The project provides public mailing lists, which were used to ask questions or discuss the proposed design.

Various conferences were visited during the project. In February a visit was made to FOSDEM in Brussels, where Robert Watson gave a lecture. In May Remko Lodder and I went to BSDCan in Ottawa, where a short talk was given on the project's status. All expenses to the conference were paid by Snow B.V. and the non-profit FreeBSD Foundation.

After the conference, access was given to FreeBSD's master CVS repository, making it possible to integrate the code into the core operating system.

3.6 Project execution

Now that a background on the assignment is given, this section will discuss how the project was executed. The project was divided in the following phases:

- Planning.
- Research.
- Design.
- Implementation.
- Documentation.

These phases will now be described in more detail.

3.6.1 Planning phase

Like most projects, the planning phase was used to write a Project Plan (see appendix A). The Project Plan was sent to both mentors. Even though the initial version was finished within a week, it has been modified throughout the project afterwards. There are a couple of reasons why the Project Plan had to be changed:

- At the beginning of the project, it was still unknown how much time was needed to implement certain requirements. This made it very hard to write an accurate schedule in advance.
- In the ninth week, I started planning a visit to BSDCan in Ottawa. Because this event lasted one week, the planning had to be adjusted.

Even though Snow B.V. uses the PRINCE2 project management technique for its internal projects, this Project Plan was somewhat different from the Project Initiation Document (PID). Normally, the PID is used to decide if the project is going to be executed. This had already been determined for this project in advance.

3.6.2 Research phase

After the first version of the Project Plan was sent to the mentors, research on the old TTY layer had already begun. During the research phase, only unofficial documentation was written. There were a couple of reasons why no official documentation of the old TTY layer was made:

- The Design Document that was written was already going to describe many major aspects of the TTY layer.
- Even though the new TTY layer has many similarities with the old implementation, its main purpose was to comply with standards like POSIX, which is already properly documented.
- The old TTY layer was already described in the book *The Design and Implementation of the FreeBSD Operating System*.

Because research on the TTY layer had already begun before work started at the office, the research phase only lasted 2 weeks.

3.6.3 Design phase

After enough knowledge was available on the old TTY layer, a design document was written, describing all shortcomings of the old code. The document described how the new TTY layer was going to be different from the old layer (see appendix B). It also described in what phases the new TTY layer was going to be implemented in FreeBSD's Perforce repository.

Because the document was going to be submitted to the FreeBSD Architecture mailing list, the document had to be easy to read. The FreeBSD Project has many volunteers. When the document would be too long or complicated, it would be less likely other people would read it.

It only took two days before most feedback was received and processed. After an updated version was posted, it was archived on the FreeBSD Wiki page.

3.6.4 Implementation phase

Now the developers were aware of the plans to rewrite the TTY layer, work was started on replacing the TTY layer. This was the longest phase of the project. It already started in the third week.

Because the project was executed using some of the principles of PRINCE2, two documents were written at the end of the week and submitted to the mentors and the observer:

- Highlight Report: This report described what work was done during the week, to be able to monitor if the project is still going according to plan. See appendix C for an example.
- Exception Report (optional): This report described any problems that were discovered during the implementation phase. Only one Exception Report was written. See appendix D for an example.

Implementation went according to plan. In the beginning, it turned out work was going ahead of schedule, which was described in the Project Plan. Half way through the project, the plan was tightened, to make room for the visit to BSDCan in Canada.

3.6.5 Documentation phase

After already 10 weeks, work started on writing the documentation that was needed to properly finish the project. Because the visit to BSDCan and preparations for the presentation would take some time, it was necessary to start on this earlier than initially planned.

Two forms of documentation were written:

- The thesis, which was written for Fontys, but also people who were interested in knowing more about the assignment.
- Documentation for the FreeBSD operating system (manual pages). Because system interfaces had slightly changed, technical documentation for the operating system had to be adjusted as well.

Even though the thesis does not entirely match the goals of the 'Closing a Project' (CP) stage of PRINCE2, it does contain similar content, including 'Identifying Follow-on Actions' (CP2) and 'Evaluating a Project' (CP3).

3.7 Delivered product

During the internship a new TTY layer for the FreeBSD kernel has been developed. The new TTY layer is mostly binary compatible with the old TTY layer, which means that it can easily be integrated into the FreeBSD operating system.

This section will describe the new functionality of this implementation.

Core TTY layer

When the implementation phase of the project started, a second TTY layer was added to the kernel. This allowed a new TTY layer to be developed, without rendering the system to an unusable state. This approach turned out to work pretty well. Drivers could be ported to the new TTY layer one by one. When all standard drivers had been ported to the new layer, the old implementation was removed from the kernel entirely.

This new TTY layer has many advantages over the old design:

- The new TTY layer allows each TTY to have its own lock. As a compatibility mechanism, drivers can override this lock. This means existing drivers that still need the Giant lock can use this lock to protect the TTY as well.
- TTY's are properly reference counted and garbage collected. Unlike the previous implementation, it is no longer possible to turn self-created devices into TTY's. TTY's are frontends, which means creation of device nodes and integrating to the system is completely handled by the TTY layer.
- The new TTY objects provide a better abstraction. It should no longer be needed to modify the datastructures of TTY's directly.

The core TTY layer is responsible for the integration with processes on the system. It exposes the device nodes in the device filesystem, but it also keeps track of the relations with processes, such as the session leader and the foreground process group.

TTYDISC line discipline

One of the most complex pieces of code in the TTY layer is the new standard `TTYDISC`. The interface of the line discipline has been improved. It has the following improvements when compared to the old implementation:

- Because the programming interface has improved, drivers contain less duplicate code. The old line discipline interface had no frontend to read data from the output queue from within the drivers.
- The old line discipline interface had no mechanism for the driver to inform that all received data was delivered to the TTY layer. This caused the TTY layer to generate many spurious wakeups. The new line discipline now has a special operation to delay wakeups until all data has been processed.
- The new `TTYDISC` line discipline is a little bit more efficient in its bandwidth usage, by printing less redundant (invisible) characters to the screen.

The `TTYDISC` line discipline should be mostly compatible with the functionality described in POSIX, but also contains some of the extended functions of the old FreeBSD `TTYDISC`.

Input and output buffer queues

During the project, much time was spent on developing the new input and output buffer queues for the TTY layer, which were meant to replace the old character list queues.

The output queue was designed to be a simple and light weight buffer queue. This output queue is used by the line disciplines to store outbound data. The queue only allowed data to be removed from the beginning and stored at the end (FIFO).

The input queue was designed to implement all operations that are needed by the `TTYDISC` line discipline:

- Support for various markers, including a reference to the position where the current line of input starts.
- Support for removing characters from the end of the queue, used to implement backspace.

- Support for iterators to implement features like line reprinting (Ctrl+R).
- Support for quoting bits, used to store parity and verbatim input (Ctrl+V).
- Support for locating characters within the buffers.

As an additional feature, both buffer queues are capable of returning data back to a userspace process using a single copy pass. The old TTY layer was not capable of safely copying data from the buffers to userspace. This means it first had to copy data in 100 byte chunks to a separate buffer before returning data to userspace. In typical cases, the new TTY layer does not need this step.

There are cases where unbuffered copying to userspace cannot guarantee proper serialisation. When multiple processes would perform read operations on the same TTY device, unbuffered reads can only be performed when a process reads enough data to empty the entire queue.

In practice, this does not seem to occur very often, which can be explained as followed:

- Many userspace applications provide large read buffers, which can hold multiple kilobytes of data. Because this is often larger than the amount of data in the kernel TTY buffers, there is no need for buffered copies.
- Even if the read buffer is small, it is often scheduled fast enough to process any user input.

To determine if the unbuffered copying would be useful in real-world scenarios counters have been added to the TTY layer, which can be printed using the `sysctl` tool. On the machine that was used during development, the following numbers were observed:

	Input path	Output path
Buffered reads	202	1
Unbuffered reads	1063	91952
Total reads	1265	91953
Effectiveness	84,0 %	100,0 %

As you can see, the effectiveness of the unbuffered copying mechanism is quite high. In the output path we can see an almost perfect score. Terminal emulating software, such as `screen` and OpenSSH, seem to be using buffers which can easily hold the data which is generated by applications.

On the input path, the approach seems to be less effective. After performing more research, it seems shells like `bash`, `tcsh` and `zsh` only use a 1-byte read buffer. This means buffered reads can occur in two situations:

- A key has been pressed, which generates a key scancode which is larger than 1 byte (Unicode, escape sequences).
- Text input is not processed fast enough by the shell, which can be caused by high system load.

Even though shells should be changed to not perform such suboptimal behaviour, the difference in performance on such small amounts of data is hard to measure.

Pseudo-terminal interface

Because the design of the pseudo-terminal driver had many problems and was also impossible to use with the new TTY layer, two new pseudo-terminal drivers were written:

- The standard `pts` driver, which implements a Linux-style pseudo-terminal device naming. The `pts` driver implements proper garbage collecting, which means pseudo-terminals are no longer recycled. This driver is also binary compatible with Linux, which means unmodified Linux binaries can now use pseudo-terminals on FreeBSD.
- The `ptycompat` driver, which implements the old BSD-style pseudo-terminal device naming. This driver was built on top of the `pts` driver. This driver shall be used by older C libraries. Tests have shown that it is even compatible with FreeBSD 5.2.1's C library (released in February 2004).

The `ptycompat` driver is expected to be used temporarily, to ease the migration. It may be removed in the far future.

Drivers

The following drivers have been ported to the new TTY layer:

- The `uart` serial interface driver.
- The `syscons` system console driver.
- The `ucom` USB to serial communication driver.
- The `ofw_console` OpenFirmware debugging console driver.

The `uart` and `ofw_console` driver do not depend on the Giant lock anymore. The other drivers still depend on the Giant lock. Making individual drivers multiprocessor safe is outside the scope of this project.

Missing features

The new TTY layer supports most features that are used most often. Most users could use the new TTY layer as it is now. Below is a list of features that are missing, for completeness:

- Non-standard line disciplines have not yet been ported to the new TTY layer.
- Three USB drivers still need to be ported.
- Four ISA card drivers using the TTY layer need to be ported.
- Miscellaneous debugging tools like the `dcons` Firewire debugging console driver still need to be ported.

These drivers have not been ported to the new TTY layer, because there was no physical hardware to test the changes on. The ISA drivers may be removed from the tree. There is a slight chance people still use these devices.

Integration with the FreeBSD codebase

During the last month of the project, access was gained to FreeBSD's CVS code repository. During this period many changes from the Perforce branch were merged back. These changes can be divided in the following groups:

- Bug fixes: during the project bugs were found in various subsystems of the kernel. These could already be fixed without importing the new TTY layer.
- API extensions: subsystems like the Device Filesystem (devfs) were slightly extended to support new features, used by the new TTY layer.
- Abstraction: changes to the old TTY layer were also made, to ease the process needed to integrate the new TTY layer.

Conformance to the goals of the project

Below is the list of goals that were defined at the beginning of the project. All goals have been met.

- Gain more knowledge on the design of the old TTY layer that exists within the FreeBSD kernel.
- Gain more knowledge on the multiprocessor support and synchronisation primitives that exist within the FreeBSD kernel.
- Improve the performance and scalability of the FreeBSD TTY layer.
- Apply elements of PRINCE2 to the management tasks of this project.
- Gain more knowledge on the organisational structure of the FreeBSD project, including communication with other developers on the team.

Chapter 4

Conclusion and recommendations

As can be read in the previous chapters, a successful attempt has been made to rewrite the TTY layer for the FreeBSD operating system, which includes improvements to its performance, scalability and stability. Now that a basic proof of concept version is available for testing, any missing features and bugs should still be fixed before it can be integrated to the FreeBSD operating system. The target is to complete this by the end of the summer this year, which means the upcoming major release of FreeBSD (version 8.0) will hopefully include these changes.

An advantage of having a multiprocessor optimized TTY layer is that additional project can be started to improve the scalability of various other subsystems within the kernel, one example being the kernel's system console. Right now the TTY layer has only been designed to provide locking on a per-TTY basis, which may not always be sufficient for systems that require high throughput and real-time responsiveness. These projects are well beyond the scope of this project and could be executed by other FreeBSD developers or volunteers in the future.

Improvements to FreeBSD's performance on multiprocessor systems has had a high priority within the project since the release of FreeBSD 5.0. This project is yet an example of the work that has been done to make FreeBSD one of the best performing operating systems on server class hardware to date.

Evaluation

The project I have been working on has been executed even more successful than I had expected beforehand. While previous attempts to improve the TTY layer were focused on refactoring the existing TTY code, I already discovered during my research phase that this would not be the correct approach. The existing implementation had too many problems.

Because the new TTY layer has a somewhat similar design and uses the same terminology as the old implementation, we get the best of both worlds. We now have a new TTY layer that implements the features we want today, existing FreeBSD developers can easily get themselves familiar with the new code.

During this project I discovered communication is a very important aspect within the FreeBSD community. Because I already announced my plans to the FreeBSD project almost a month before I started working at the office, the project was already well aware of my plans.

BSDCan also took place at a rather perfect time. Remko and I went to BSDCan in the fifteenth week of the project. At this time I already knew everything about the existing TTY layer and almost all basic features of the new TTY layer were already implemented. This meant that most developers saw that I had good intentions with my project.

In my opinion this project has showed that I am capable of creating, formulating, planning and executing a project. I have also demonstrated that I am willing to discuss my work with other people of the project, even though discussing this may sometimes be hard. It is not always easy to find a consensus within a project which has the size of the FreeBSD.

I have full confidence that – based on the decisions that I have made – my work will eventually be integrated into the FreeBSD operating system.

Bibliography

- *The Design and Implementation of the 4.4BSD Operating System*
  Marshall Kirk McKusick, Keith Bostic, Michael J. Karels and John S. Quarterman
  ISBN: 0-201-54979-4
  - Chapter 4: Process Management
  - Chapter 6: I/O System Overview
  - Chapter 10: Terminal Handling
- *The Design and Implementation of the FreeBSD Operating System*
  Marshall Kirk McKusick and George V. Neville-Neil
  ISBN: 0-201-70245-2
  - Chapter 4: Process Management
  - Chapter 6: I/O System Overview
  - Chapter 10: Terminal Handling

Appendix A

Project Plan - Dutch

Plan van aanpak
FreeBSD TTY refactoring

Ed Schouten

15 april 2008

Snow B.V.
Koeweistraat 12
4181 CD Waardenburg
Postbus 72
4180 BB Waardenburg
tel. 0418–653 333
fax 0418–653 666

info@snow.nl
http://snow.nl
k.v.k. 17101631
bank 68.05.08.430
giro 5377513

Versiebeheer

Auteur	Datum	Versie
Ed Schouten	5 februari 2008	1.0
Ed Schouten	6 februari 2008	1.1
Ed Schouten	26 februari 2008	1.2
Ed Schouten	27 februari 2008	1.3
Ed Schouten	15 april 2008	1.4
Opmerking: in versie 1.4 is alleen de tijdsverdeling bijgewerkt, om tijd te reserveren voor BSDCan.		

Distributie

Naam	Functie	Email
Jos Jansen	Opdrachtgever	jos@snow.nl
Joost Helberg	Toezichthouder	joost@snow.nl
Patrick Zwegers	Toezichthouder	p.zwegers@fontys.nl

Akkoord

Naam	Datum	Paraaf
Jos Jansen		
Joost Helberg		
Patrick Zwegers		

Inhoudsopgave

Hoofdstuk 1

Inleiding

Voor mijn stageopdracht binnen Snow B.V. heb ik gekozen om onderzoek te doen naar een subsysteem binnen het FreeBSD operating system, namelijk de TTY laag, welke verantwoordelijk is voor het verschaffen van terminals, waarmee een persoon interactief gebruik kan maken van een UNIX systeem. Dit document zal een achtergrond geven van de huidige situatie van de TTY laag, welke verbeteringen er kunnen plaatsvinden en hoe de uitvoer van de opdracht gefaseerd zal worden.

Snow B.V. is een IT bedrijf die opgericht is in 1997. Snow B.V. houdt zich bezig in diverse sectoren, namelijk systeembeheer, netwerkbeheer en advies op het gebied van security. Snow B.V. Het bedrijf heeft over de jaren veel expertise opgebouwd over systemen die gebruik maken van UNIX-achtige besturingssystemen, maar ook de diensten die hierop veelal toegepast worden.

FreeBSD is een Open Source besturingssysteem dat ontwikkeld word door honderden ontwikkelaars, maar ook duizenden vrijwilligers. FreeBSD heeft een vergelijkbare functionaliteit als het Linux besturingssysteem en kan ook een groot aantal applicaties draaien die ook onder Linux beschikbaar zijn, zoals de GNOME desktop en applicaties van Mozilla.

Doel document

Dit document geeft inzicht in het projectresultaat, het bereik van het project, de randvoorwaarden en de globale kosten.

Hoofdstuk 2

Inleiding en achtergrond project

Zoals de meeste UNIX-achtige besturingssystemen, beschikt FreeBSD over een TTY (Teletype) implementatie. TTY's zijn een categorie apparaten waar vanuit een applicatie tegenaan geprogrammeerd kan worden. Deze apparaten kunnen terminals of andere seriële hardware zijn. Een voorbeeld van dit soort apparaten zijn seriële poorten waaraan fysieke terminals verbonden zijn.

TTY's kunnen onder UNIX gebruikt worden om interactief met processen te werken. Op een standaard UNIX-achtig systeem worden zogenaamde `getty`'s gedraaid die een login prompt weergeven op de betreffende TTY. Wanneer de gebruiker inlogt, wordt zijn eigen shell gestart, waarmee de gebruiker zelf processen kan starten.

De shell gebruikt de TTY om bepaalde functionaliteit te implementeren, zoals toetsen waarmee applicaties geforceerd afgesloten kunnen worden (`^C` en `^\`), uitvoer van de terminal tijdelijk te onderbreken (`^S` en `^Q`), applicaties tijdelijk te stoppen (`^Z`) en tekstinvoer te kunnen realiseren zonder interactie van processen (`^W`, `^U` en backspace toets). Andere applicaties kunnen diverse parameters van de TTY aanpassen. Zo zetten applicaties die om een wachtwoord vragen de echo-functionaliteit van de terminal uit, om te zorgen dat het ingevoerde wachtwoord niet in beeld verschijnt.

Naast de fysieke terminals bestaat er ook een virtuele soort TTY's, namelijk PTY's (Pseudo-teletypes). Deze apparaten zijn niet verbonden met fysieke apparatuur, maar kunnen aan beide kanten aangestuurd worden door processen. Enkele programma's die gebruik maken van PTY's, zijn terminal emulators zoals `xterm` en `screen`, maar ook loginservices zoals `sshd` en `telnetd`.

Omdat PTY's zijn geïmplementeerd via de TTY laag, betekent dit dat een programma zonder aanpassingen gebruik kan maken van zowel TTY's als PTY's. Dit betekent dat er vrijwel geen verschil is tussen het gebruik van een fysieke

terminal (de VGA system console of een VT100) of een terminal emulator zoals `xterm`.

2.1 FreeBSD's TTY implementatie

Omdat FreeBSD een zeer lange geschiedenis heeft die zelfs terug gaat naar het ontstaan van UNIX, is het geen wonder dat er nog bepaalde stukken code binnen deze laag al enkele decennia oud zijn. Dit betekent dat de code zich onderhand bewezen heeft en ook vrij robuust werkt. Helaas heeft de huidige implementatie enkele nadelen.

Veel redundante code

Een van de dingen waarop BSD-achtige systemen in het verleden verschilden van SystemV-achtige systemen, was de manier waarop TTY's geconfigureerd konden worden (baud rate, echo, flow control, etc). Zo gebruikten BSD-achtige systemen de `sgtty` interface en SystemV-achtige systemen de `termio` interface.

Beide interfaces beschikten over ongeveer dezelfde functionaliteit, maar `termio` gebruikte een iets logischere naamgeving. Uiteindelijk is er in 1988 in POSIX een standaard-interface gedefinieerd die `termios` heet. Deze interface heeft veel weg van `termio`, maar geeft makers van besturingssystemen meer vrijheid in de manier waarop dit geïmplementeerd wordt.

Toen het FreeBSD project in 1993 opgericht werd, was er zowel `sgtty` en `termios` aanwezig. De `termios` interface werd namelijk geïntroduceerd in 4.3BSD Reno. Dit zorgt voor veel redundante code binnen de kernel. In de werkelijkheid is te zien dat `sgtty` vrijwel nooit meer gebruikt wordt. Het is dus te overwegen om deze interface uit te faseren, zodat applicatieschrijvers uitsluitend de gestandaardiseerde interface kunnen gebruiken.

Implementatie voldoet niet aan de wensen van nu

Voordat workstations te vinden waren op ieder bureau, was vaak te zien dat dure UNIX systemen voorzien waren van meerdere fysieke terminals die tijdens de levensduur van het systeem altijd verbonden zijn.

Dit staat eigenlijk haaks op het gebruik van terminals op een modern systeem, waarbij de gebruiker zelfs in een minuut tijd meerdere terminals maakt en weer sluit. Op een gemiddeld desktopsysteem kan een gebruiker ervoor kiezen om alleen maar met PTY's te werken die gealloceerd worden wanneer de gebruiker

het wil. Op shellservers kunnen vele mensen tegelijk inloggen en uitloggen. Ook kunnen apparaten tegenwoordig zomaar aangesloten en ontkoppeld worden, bijvoorbeeld via USB.

Dat de huidige TTY implementatie hier niet op berekend is, is duidelijk te zien in de PTY en USB code, waar met de grootste moeite gezorgd moet worden dat TTY's en bijbehorende buffers opgeruimd worden. De PTY code dealloceert zelfs nooit terminals, maar zet ze in een lijst, zodat ze later opnieuw gebruikt kunnen worden. Dit kan uiteraard verbeterd worden door correcte reference counting te gebruiken op TTY objecten.

Implementatie maakt gebruik van de Giant lock

De hardware waarop UNIX en BSD systemen van oorsprong op werden ontwikkeld, waren de PDP computers. Deze systemen beschikken uiteraard niet over meerdere processoren en uitgebreidere interruptafhandeling, zoals die te vinden zijn in de computerhardware van tegenwoordig. Wanneer er op een klassiek UNIX-achtig systeem in de kernel een kritieke sectie uitgevoerd werd, maakte de programmeur gebruik van functies om interrrupts in en uit te schakelen, zoals `spltty()`, `splhigh()` en `splx()`. Tijdens de ontwikkeling van FreeBSD 5 zijn deze functies vervangen door een locking model dat gebruik maakt van mutexes om zo zeer precies datastructuren in de kernel af te schermen van interrupts en andere processoren.

Helaas is de kernel te groot om in één keer een transitie te maken naar dit model, dus over de laatste jaren zijn steeds meer subsystemen binnen de kernel (de netwerkstack, de diskaansturing, etc.) vrij gemaakt van de zogenaamde Giant lock. De Giant lock is een enkele mutex die bedoeld is om alle code af te schermen die nog niet beschikt over correcte locking. Op dit moment is de TTY laag nog een van de weinige veelgebruikte subsystemen die hier nog gebruik van maakt.

Het nadeel van het gebruik van de Giant lock is dat een systeem met meerdere processoren met oponthoud te maken krijgt wanneer meerdere TTY's gelijktijdig gebruikt worden, maar ook wanneer er gelijktijdig gebruik gemaakt wordt van een irrelevant subsysteem dat ook de Giant gebruikt.

Het zou mogelijk moeten zijn om iedere TTY zijn eigen lock te geven, wat de performance, maar ook tijdsgaranties van de kernel zou moeten verbeteren.

2.2 Onderzoeksvragen

Uit de vorige paragrafen kunnen we de volgende onderzoeksvragen afleiden, waarbij een proof of concept implementatie ontwikkeld moet gaan worden:

- Welke programmeerinterfaces voor applicaties buiten de kernel bieden een redundante functionaliteit? Wanneer deze interfaces in de praktijk ongebruikt blijken te zijn, maak ze dan onbeschikbaar zonder bestaande applicaties onbruikbaar te maken. FreeBSD's software management systeem (FreeBSD Ports) kan gebruikt worden om te onderzoeken of de interfaces daadwerkelijk ongebruikt zijn.
- Ontwerp een nieuw model voor de TTY laag waarmee TTY's dynamisch gealloceerd kunnen worden, maar ook veilig gedealloceerd kunnen worden, zodat ongebruikte PTY's opgeruimd kunnen worden en apparatuur veiliger gehotplugd kan worden.
- Ontwerp een nieuw model voor de TTY laag waarmee TTY's simpel, maar toch vrij precies gelockt kunnen worden om zo de performance op multiprocessorsystemen te verhogen.

Hoofdstuk 3

Projectdefinitie

Tijdens dit project zal er een proof of concept implementatie ontwikkeld gaan worden van een nieuwe TTY implementatie die voordelen biedt tegenover de oude implementatie op het gebied van performance en functionaliteit. Deze implementatie moet zowel geschikt zijn voor fysieke apparatuur als pseudo-apparatuur zoals PTY's.

3.1 Doelstellingen

3.1.1 Leerdoelen

Het uitvoeren van de stage zal enkele doelstelling hebben die te verdelen zijn in verschillende categoriën, namelijk technische en procesgerichte leerdoelen.

Technische leerdoelen

- Vergroten van de kennis van de werking van TTY laag binnen de FreeBSD kernel.
- Vergroten van de kennis van het model voor multiprocessor-ondersteuning en synchronisatieprimitieven binnen de FreeBSD kernel.

Procesgerichte leerdoelen

- Procesmatig werken met PRINCE2.

- Kennis opdoen van de organisatie van FreeBSD, met name het communiceren met andere ontwikkelaars.

3.1.2 Projectdoelen

Het doel van het project is om de TTY implementatie binnen de FreeBSD kernel te verbeteren, om zo de prestaties van het systeem in veel gebruikelijke, maar ook veel eisende situaties te verbeteren.

3.1.3 Bereik

De ontwikkeling van de nieuwe TTY laag zal enkel plaatsvinden op de vendor branch van FreeBSD in de Perforce repository. Deze vendor branch bevat dezelfde code als de CVS HEAD, namelijk die van `CURRENT`.

3.1.4 Belangrijkste resultaten

Hieronder volgt een opsomming van de documenten en producten die opgeleverd zullen worden tijdens dit project:

- Plan van aanpak, met bijbehorend communicatieplan
- Ontwerpdocument voor het FreeBSD project
- Wekelijkse rapportages
- Stageverslag
- Ontwikkelde sourcecode, minstens beschikkend over een driver voor een virtueel TTY apparaat (PTY's) en een fysiek TTY apparaat (`uart(4)` of `sc(4)`).

3.1.5 Afbakening

De implementatie zal alleen ontwikkeld worden voor de FreeBSD vendor branch en zal niet voor `RELENG_7` of ouder ontwikkeld worden. Ook zal tijdens dit project alleen maar aan drivers gewerkt worden die beschikbaar zijn op het testsysteem.

3.1.6 Randvoorwaarden en beperkingen

Tijdens dit project zullen enkele middelen beschikbaar moeten zijn, om te kunnen garanderen dat er aan dit project gewerkt kan worden:

- Een test- en ontwikkelsysteem dat in staat is om FreeBSD zonder problemen te draaien.
- Toegang tot FreeBSD's Perforce repository, om progressie te kunnen volgen.

3.1.7 Relaties met andere projecten

Dit project zal zelfstandig uitgevoerd worden van andere projecten die binnen Snow bekend zijn.

Hoofdstuk 4

Business case

De huidige implementatie is een belemmering op de prestaties van FreeBSD en zorgt voor hinder in het optimaliseren van andere subsystemen binnen de kernel.

Hoofdstuk 5

Referentiemateriaal

De volgende documentatie kan geraadpleegd worden tijdens het uitvoeren van dit project:

- The FreeBSD Project
  `http://www.FreeBSD.org/`
- The Design and Implementation of the FreeBSD Operating System
  Marshall Kirk McKusick, George V. Neville-Neil
  ISBN: `978-0201702453`

Hoofdstuk 6

Kwaliteitsverwachtingen

De implementatie zal aan de volgende kwaliteitseisen moeten voldoen, om geïntegreerd te kunnen worden in het FreeBSD besturingssysteem:

- De implementatie zal aan codestandaarden moeten voldoen die binnen het FreeBSD project gesteld worden.
- De implementatie moet opgeleverd kunnen worden in kleinere stukken, om zo het auditeren van de nieuwe implementatie te versimpelen.

Hoofdstuk 7

Geïdentificeerde risico's

Tijdens dit project zijn de volgende risico's van toepassing:

- De implementatie voldoet niet aan de wensen van het FreeBSD project.
- De stagiair is niet in staat de implementatie te schrijven.
- De stagiair is niet in staat de implementatie binnen het tijdsbestek aan de gestelde eisen ver genoeg af te ronden.

Hoofdstuk 8

Planning

Voor dit project zijn 20 werkweken (100 dagen) uitgetrokken. Deze zijn verdeeld over enkele fasen.

8.1 Faseverdeling

Initiële documentatie

In de eerste fase van de stage zal documentatie geschreven worden die de werkwijze beschrijft, namelijk het Plan van aanpak (dit document). Dit document zal ook opgestuurd worden naar Fontys Hogescholen, omdat de stagebegeleider ook hiervan op de hoogte moet zijn.

Onderzoeksfase

Nadat het Plan van aanpak is goedgekeurd door Snow en Fontys Hogescholen het document heeft ontvangen, kan er onderzoek gedaan worden naar de huidige implementatie.

Tijdens deze fase zal er wel documentatie geschreven worden, maar deze hoeft niet opgeleverd te worden. De documentatie zal puur dienen als hulpmiddel. Ook zal er bestaande documentatie gelezen worden, zoals paragrafen uit ‘The Design and Implementation of the FreeBSD Operating System’ die van toepassing zijn.

Ontwerpfase

Nadat er inzicht gecreëerd is van de bestaande implementatie, kan een een ontwerp gemaakt worden, waarin beschreven zal worden hoe de nieuwe implementatie zal functioneren. In dit project is het belangrijk dat dit document ook zal beschrijven in welke volgorde de implementatie gerealiseerd zal worden.

De documentatie zal in ieder geval gedeeltelijk in het Engels geschreven moeten worden, want dit zal het mogelijk maken dat andere FreeBSD ontwikkelaars het ontwerp kunnen goedkeuren. Een ander voordeel is dat dit document uiteindelijk als basis zou kunnen dienen voor documentatie voor anderen.

Implementatiefase

Wanneer andere FreeBSD ontwikkelaars de mogelijkheid hebben gehad om feedback te geven, kan het implementeren beginnen. Om te valideren dat de implementatie naar behoren werkt, zullen er diverse bestaande applicaties gedraaid worden die gebruik maken van de nieuwe code, zoals terminal emulators en applicaties die de seriële lijn gebruiken.

Wanneer er tijdens de implementatie bugs ontdekt worden, zullen er meteen tests aangelegd worden om de bugs te kunnen reproduceren. Deze test-suite kan uiteindelijk ook gebruikt worden om te testen of de implementatie conform is aan specificaties zoals POSIX en het gedrag van andere besturingssystemen zoals Linux.

Omdat de implementatie zal gaan dienen als een Proof of Concept, hoeft de code niet geheel afgerond te zijn aan het einde van deze fase. Het streven is om aan het einde van deze fase een implementatie te hebben die op zijn minst bruikbaar is voor diverse terminal emulators en loginservices.

Afronding

Wanneer de implementatiefase afgerond is, zal er nog afrondende documentatie geschreven moeten worden, waaronder een stageverslag en een afsluitende presentatie die zowel op het bedrijf als op Fontys Hogescholen gehouden moet worden.

Er is een grote kans dat de implementatie aan het einde van deze fase nog niet afgerond is. Dit is niet erg, omdat er na de stageperiode nog verder gewerkt kan worden, omdat de code publiekelijk toegankelijk zal zijn. Er zal wel een beknopt document in het Engels geschreven moeten worden waarin behandeld zal zijn hoe de implementatie verder afgerond moet worden.

8.2 Tijdsverdeling

Voor de bovenstaande fasen is de volgende tijdsverdeling gekozen:

	Tijdsduur	**Begindatum**	**Einddatum**
Initiële documentatie:	4 weken	4 februari	29 februari
Onderzoeksfase:	1 week	11 februari	15 februari
Ontwerpfase:	1 week	11 februari	15 februari
Implementatiefase:	15 weken	18 februari	30 mei
Eerste versie TTY objecten	2 weken	18 februari	29 februari
Eerste versie PTY driver	2 weken	18 februari	29 februari
Implementatie buffers en line discipline – milestone 1	4 weken	3 maart	28 maart
Eerste opzet `uart(4)` en `ucom(4)` drivers – milestone 2	2 weken	31 maart	11 april
Integratie procescode en connectie state	2 weken	14 april	25 april
Bezoek BSDCan, inclusief voorbereidingen	2 weken	28 april	16 mei
Opruimen en documenteren nieuwe code – milestone 3	2 weken	19 mei	30 mei
Afronding:	3 weken	2 juni	20 juni

Hoofdstuk 9

Beheersingsmechanismen

Iedere week zal een rapportage gestuurd worden naar Jos Jansen, waardoor bepaald kan worden of de snelheid waarop het project verloopt binnen de toleranties ligt.

9.1 Toleranties

Net als de andere projecten binnen Snow, zal een tolerantie van 10% van toepassing zijn. Indien uit de wekelijkse rapportage afgeleid kan worden dat de afwijking boven de vastgestelde tolerantie ligt, moeten de opdrachtgever en de toezichthouder hiervan op de hoogte zijn.

9.2 Uitzonderingsprocedure (exceptions)

De uitzonderingsfase treedt op wanneer een fase of het gehele project over de tolerantiewaarde komt. Met de opdrachtgever zullen zaken besproken worden, zoals de aanleiding en de oplossing hiervoor.

Hoofdstuk 10

Oplevering

Na contact te hebben opgenomen met het FreeBSD Core Team, hebben zij toegang tot Perforce verstrekt. Perforce is een versiebeheersysteem dat het FreeBSD project gebruikt om experimentele ontwikkelingen in op te slaan. Dit betekent dat de ontwikkelde implementatie direct opgestuurd kan worden naar het FreeBSD project zelf. Dit heeft enkele voordelen:

- Andere ontwikkelaars kunnen assisteren bij het implementeren en testen. Er is geen beschikking over alle hardware die gebruik maakt van de TTY laag.
- Er kan makkelijk feedback worden gegeven op de implementatie, omdat andere Perforce gebruikers mail ontvangen bij iedere commit.
- Er kan geen misverstand ontstaan over de richting die tijdens de implementatie gekozen wordt. De code wordt niet in een keer opgeleverd. Er is dus een grotere kans dat het werk aan de wensen voldoet, wat integratie in FreeBSD zelf waarschijnlijker maakt.
- Het werk kan minder snel verloren gegaan.

Hoofdstuk 11

Communicatieplan

Het onderstaande schema behandelt welke communicatie er plaats zal vinden met de docentbegeleider van Fontys Hogescholen.

Onderwerp	**Datum**	**Medium**	**Actie**
Plan van aanpak	6 februari	email	
Communicatieplan	12 februari	email	
Eerste bedrijfsbezoek	22 februari		Goedkeuring documentatie
Definitieve Plan van aanpak	28 februari	email	
Eerste concept stageverslag	30 april	email	Commentaar docent
Tweede bedrijfsbezoek, inclusief proefpresentatie	31 mei		
Tweede concept stageverslag	31 mei	email	Commentaar docent
Definitieve stageverslag	30 juni	post	

Appendix B

Design document - English

Redesigning the FreeBSD TTY layer

Ed Schouten

April 15, 2008

Contents

1 Introduction

Like many UNIX-like operating systems, the FreeBSD operating system includes an interface that is often referred to as the *TTY layer*. This layer represents a category of devices that have one thing in common: serial lines, which may or may not be connected to a computer terminal. When connected to a terminal device, a user has the ability to log in and run commands in a shell.

This document will describe alterations that could be made to the existing TTY layer that is part of the FreeBSD operating system, which could eventually lead to a more robust and multiprocessor-optimised system.

The changes that are described in this document will eventually be implemented in the `mpsafetty` branch on FreeBSD's Perforce server. Work on the initial implementation will be funded by Snow B.V. (`http://www.snow.nl/`).

2 Limitations of the current implementation

Like most other BSD-like operating systems, FreeBSD's TTY implementation is based on the original 4.4BSD Lite code. The code has evolved a lot since the start of the FreeBSD project, though the functionality has basically remained the same over the years.

Even though the code is quite robust, it has some structural design issues that prove to be a limitation to FreeBSD's potentials.

Inefficient locking model

One of the things FreeBSD excels in when compared to the other BSD's, is the performance of its multiprocessor implementation. On FreeBSD, data structures in the kernel are protected using fine-grained locking. This means that unlike the original priority based synchronisation scheme, synchronisation is performed with mechanisms like mutexes. When properly implemented, this will lead to a system where unrelated threads won't block on each other, even if both threads are executing similar code within the kernel.

Because changes like these cannot be implemented without breaking compatibility with existing code, the FreeBSD operating system places all legacy code within a mutex called the Giant lock. At this moment the TTY layer is one of the last remaining often used frameworks which still uses the Giant lock to protect its internal structures. A smarter approach would be to give each TTY device its own mutex.

Code lacks any form of abstraction

At this moment the TTY code within the kernel acts as if it is a library, which means that most of the device drivers take care of creating and destroying the device nodes, but call into the TTY layer on

certain events. This causes a lot of redundant code within the device drivers.

A better solution would be if the TTY layer would act as a frontend, with hooks at proper locations which enable device drivers to plug their own functionality into the TTY layer, instead of wrapping around it.

Reference counting is missing

One of the things that has really changed since the start of the FreeBSD project, is that users nowadays connect and disconnect a lot of devices to their systems and expect them to work without rebooting first. Of course, creating new TTY devices on the fly isn't a problem with the current implementation. Unfortunately, destruction of TTY devices isn't really handled safely, which causes numerous strange bugs to appear in those situations.

Some drivers that have to deal with TTY destruction frequently (the PTY driver) never destroy the actual TTY objects, but just recycle old ones by keeping a free list. When the TTY is reused, a `revoke()`-like mechanism is used to make sure the TTY isn't used by strangers.

Conclusion

Even though the FreeBSD TTY layer has proven to work in practice, it could undergo many structural changes to become even better. Unfortunately such changes cannot be implemented without breaking compatibility with the current TTY driver architecture.

The next chapter will describe the proposed changes in detail, followed by a plan on how the changes are implemented without causing too much breakage for drivers and userspace applications.

3 Proposed changes to the TTY layer

This chapter will describe the problems with the TTY layer in more detail, including solutions to solve them. The reader is assumed to have knowledge about the inner workings of the TTY layer.

3.1 General TTY interface

Introduce a TTY device class structure

A lot of frameworks in the kernel use structures filled with function pointers to define classes. An example of this is the Virtual File Switch (VFS) which has a table of `vfsops` and `vnops` to determine what code should be run on vnodes and mounts. This allows the VFS to operate on different filesystem types like UFS, FAT32 and NTFS.

The TTY layer needs a similar construction to expose different types of terminal devices through a consistent programming interface. On FreeBSD, `struct tty` contains a list of function pointers that represent the operations that are implemented in the device driver. It is better to split this off into a `ttydevsw`, which will have a similar functionality as the well-known `cdevsw`. This gives us some advantages:

- Removing the list from each TTY instance consumes less memory.
- When enforcing that each driver implements all routines, the TTY code has no need to check for its existence.
- The structure can easily be extended to store per-class flags.
- When the device is being disconnected, the class can be set to `NULL` to ensure no operations in the device driver are going to be called.

Disallow direct device node creation

Most of the TTY device drivers create their own device nodes. When we want to implement proper reference counting and safe destruction of TTY devices, we must make the generic TTY code the owner of the device nodes it creates. This way the TTY code can remove the device nodes from the device file system when nobody is using the TTY device anymore.

This will also cause all the TTY cdev operations to be marked static, because they will have no reason to be called outside the TTY code.

Add new reference counting routines

TTY's basically have three angles in which reference counts should be tracked:

- They should keep track whether they are opened or not.
- The TTY may have abandoned by its driver.
- Sessions refer to TTY devices to perform accounting and to signal processes when the session leader shuts itself down.

The current implementation already performs some basic reference counting on TTY devices, but some dangerous situations are still present:

- Inside `sessrele()`, the routine that deallocates sessions, the backreference from the TTY to the session structure isn't cleared, which may cause panics to happen when running `stty(1)`.
- Some device drivers, like the PTY driver don't use the existing `ttygone()` interface to report that the master device has been abandoned. It has its own interface, which leaks TTY's in numerous situations.

That's why a new reference counting scheme should be applied, using the following functions:

- `ttyrel_sess()` should be called from `sessrele()` to lower the reference count on the TTY and remove the backreference to the session structure. Note that there isn't a `ttyref_sess()`, because TTY's and sessions are linked together through the ioctl `TIOCSCTTY`.
- `ttyrel_gone()` should be called from the device driver. This function can only be called once of course. This will lower the reference count on the TTY and unset the `ttydevsw` as stated previously.
- `ttyref()` and `ttyrel()` shall still exist and will be used only within the TTY layer to adjust the reference count when character device nodes are opened and closed.

This means when the reference count drops to zero, the device has been abandoned by the driver and processes that use the node, which allows the kernel to safely destroy the TTY and its device nodes.

Remove sgtty as a public programming interface

Apart from the POSIX `termios` interface – which can be used by applications to change a variety of parameters of a TTY device – the FreeBSD kernel still implements the legacy `sgtty` interface.

Almost a year ago work has begun to make all applications in FreeBSD Ports use the POSIX interface, in order to phase out the old interface. This work is almost finished; there are less than 5 ports left in the entire tree that still make use of this interface.

The new implementation should at least make the old interface inaccessible when compiling new binaries, to make sure the interface can eventually be removed from the source code.

A patch has already been committed to the Perforce tree, which removes the interface, but still is binary compatible. This patch should be merged back to CVS as soon as possible, because this would lower the barrier to just remove the entire interface in the Perforce branch.

Split off the termios line discipline

The current implementation supports various line disciplines, which allows you to use TTY's not only for running a shell. An example is the PPP line discipline (`PPPDISC`), which allows Point-to-Point connections to be established through a TTY device, like modems.

The most often used line discipline, is the `TTYDISC`, which is the standard discipline, which is used for regular terminal interaction. This discipline is integrated into the TTY code, but could be split off to make the line discipline interface more abstract.

3.2 PTY interface

Move posix_openpt() into the kernel

The current PTY implementation (both `/dev/pty*` and `/dev/pts/*`) suffers from the problem that a simple `stat()` on a nonexistent device will already create the device, even if the user has no intention to use it.

One solution would be to move TTY allocation into a system call named `posix_openpt()`, which will return the PTY master device. The master device will become a seperate filetype (like sockets, pipes, etc). The master device will not be visible in devfs. There is no need for exposing the master device. Other operating systems like Linux don't expose them either.

Optional: implement grantpt() through ioctl()

Currently `grantpt()` is implemented through a setuid application which changes the ownership of the slave to the real user ID of the calling process. It would be a lot cleaner if we would implement this in the kernel.

Implementing this in kernelspace has the advantage that signal handlers won't be affected. POSIX currently states it is implementation dependent whether the routine generates `SIGCHLD`'s when invoked, but it would be nicer if we can offer an implementation that doesn't do so.

Move prison checks into devfs

The current PTY code has some checks to make sure that the PTY can only be opened in the same jail as it was created in. In theory there could be other types of devices that could use such a functionality: all device drivers that construct devices on demand.

We already store the credentials when creating a device node, which means we only need to check the prison against the prison of the calling process to determine if the process is privileged to open the device.

Whether or not this will be implemented, still has to be discussed.

3.3 Clists

Improve programming interface

The TTY code stores its character buffers in so-called *clist*s. Each TTY has three lists; two to store incoming characters (the *rawq* and *canq*) and one to store outgoing characters (the *outq*).

The clists can be altered by the functions described in `tty_subr.c`. Unfortunately, a lot of device drivers tamper with the clists directly, which makes it easier to break existing drivers when changing the clist code. The programming interface needs to be extended to make it possible to use clists without needing to dive into its internals.

Function names that are currently used are sometimes hard to understand for people that don't have much experience with the TTY code (i.e. `q_to_b()`).

catq() is incredibly inefficient

When the TTY is configured to canonicalise terminal input, it uses two queues to store terminal input. When a user inserts a carriage return, the canq is appended to the rawq, making the entire line available to the application.

Unfortunately the clist code doesn't allow cblocks to be sparsely filled, which means fragmentation needs to get removed each time the queues are concatenated.

Output queue has quoting bits

The current TTY interface allows applications to enable parity checking on terminal input. One feature that's often used in terminal emulators is the `PARMRK` switch, which adds a special escape sequence in front of characters that had bad parity on input.

Because the clist code is meant to be generic for both the input and output paths, the quoting bits – that are used to store characters with bad parity – are also used in the output queue, even though they remain unused. This causes a loss of 1 bit per byte of storage in the cblocks.

Input and output queues should be split up

In order to solve the previously mentioned problems, there should be two different TTY queue mechanisms in the kernel:

- A lightweight queueing mechanism for storing output characters, which does not support quoting bits or removing previously added characters, called the *outq*.
- A heavier queueing mechanism that allows markers to be set for canonicalisation, but doesn't need to concatenate multiple queues. This will cause data to never be copied. This queue is responsible for storing input characters, thus called the *inq*.

Not going to be implemented: lazy allocation

The current TTY implementation already preallocates data buffers when the device is opened. Unfortunately, we cannot fix this in an easy way. We need to preallocate the buffers on both the queues:

- Drivers like `sio(4)` call `l_rint()` from within an interrupt context, which means we cannot use lazy allocation on the input queue.
- When the terminal is configured to echo characters, input needs to be stored on the output queue as well.

3.4 Locking

Spin or sleep mutexes

Unfortunately, there are a lot of drivers that use the TTY layer that require spin mutexes. The current implementation solves this by making the specific driver responsible for locking down the TTY. The `sio(4)` driver has a `sio_lock` which is acquired before entering the TTY layer through the device node or when an interrupts occurs.

In the new implementation, we can give each TTY a flag to store whether this TTY needs a spin or a sleep mutex. This way drivers can just call `tty_lock()` and `tty_unlock()` to lock the mutex, which is a spin mutex for `sio(4)` and a sleep mutex for `pty(4)`. The driver can also use this mutex to protect its own data.

Just like the current locking model, the TTY will be locked when calling into the driver. This means that a driver may still have its own locks, but they must be unlocked when locking the TTY object when, for example, calling into the entry points of the line discipline. Not doing so may cause a LOR[1] to occur!

Making the process code Giant free

The current state of the TTY layer forces us to pick up Giant inside routines like `exit1()`, because the relation of TTY's towards processes is quite close.

There are essentially two fields in the TTY structure that are responsible for describing the relation towards the processes, namely `t_pgrp` and `t_session`. The `t_pgrp` points to the foreground process group – the process group which is allowed to interact with the user through data I/O and signals – and `t_session` points to the session, which may or may not have a session leader.

The `t_pgrp` field is very often used inside the TTY code. This means it is a good idea to lock this field using the per-TTY mutex. The `t_session` field, however, suffers from a chicken-and-egg problem. Because the session structure has a reference to the TTY as well, there is no correct way to update the fields when we need to change this relation.

In almost all the cases where we need the TTY's `t_session` or the session's `s_ttyp`, we already hold the `proctree_lock` SX lock[2]. This means we're better off locking down these fields through this global lock.

4 Roadmap

The changes that are described in the previous chapter are quite hard to implement without breaking the current driver model. Because the TTY drivers often provide the ability to serve as system and debugging console, a very small flaw may cause the console to break down, making it very hard to debug the system.

This is why I have decided to implement these changes in a new TTY layer, which will live next to the existing one. This has a couple of advantages:

- The new implementation can still be debugged using a TTY device that is supported by the old layer.
- Existing drivers can still be used during development.
- Changes can be applied to the new TTY layer without breaking all drivers at once.

All drivers should be ported before the new implementation is integrated into FreeBSD itself, which means the old TTY layer will be completely replaced by the new layer.

The following roadmap will be used to implement the new layer:

Remove source compatibility

To ensure that existing applications will work with the new TTY implementation, the `sgtty` source compatibility should be removed from the FreeBSD system in CVS. It is very important that this shall be done as soon as possible to ensure a painless integration of the new implementation.

[1] Lock Order Reversal

[2] Shared/eXclusive lock: reader/writer lock without priority propagation

Renaming the old TTY layer

Because the new TTY layer will have a lot of routines and structures which have similar names, the old TTY layer should be renamed to `otty`. The TTY layer also exposes various sysctl's which should be renamed, to make sure the new layer doesn't register those as well.

Only the `sc(4)`, `sio(4)` and `pty(4)` drivers will be altered to work with the old layer, because development will be done on standard PC hardware.

The new `/dev/pts`-style PTY driver shall be removed from the kernel. The new TTY layer will also feature a new PTY driver with similar naming, so the old PTY driver should be used in the mean time.

Implement basic TTY features

When the old layer is moved aside, a new TTY layer will be implemented. In order to test the new implementation, a new `/dev/pts`-style PTY module will be implemented, which can allocate TTY's using a new `posix_openpt2()` system call. This means the new TTY layer can be tested using patched terminal emulators, like `xterm` and `screen`.

Remove the old PTY driver

When the new PTY driver works like it should, the `posix_openpt2()` call is renamed to `posix_openpt()`. The old `/dev/pty`-style PTY driver shall also be replaced by a new driver that uses the new TTY layer, which shall be called `pty_compat` and will serve as a compatibility driver when users are migrating to the new TTY layer. This driver will still allow TTY's to be leaked, but will only be used for migration purposes.

Port the serial port driver

After the virtual PTY device drivers have been implemented, a physical device driver like `sio(4)` should be ported to the new TTY layer. This is a very important step, because when this driver works, we know the new layer also works from within interrupt contexts.

Port the console code to make logging work

When we have a working physical device, we can change the existing `/dev/console` driver to work with new TTY's.

Port the system console driver

The last driver that is often used on standard PC hardware in combination with VGA, is the system console driver `sc(4)`. This driver should also be ported to the new TTY layer to make the system as usable as it was before.

Port other drivers to the new layer

After the system is usable again, using the new TTY layer, all the remaining drivers should be ported to the new TTY layer, like `ucom(4)` and the console drivers used on other architectures. This will take a long time and also requires interaction with other FreeBSD developers that own specific hardware.

Remove the old TTY layer

After all useful drivers have been ported to the new TTY layer, the old layer can be removed from the kernel.

5 Conclusion

As stated in the previous chapters, implementing a new TTY layer will have a lot of advantages for the FreeBSD operating system. The new TTY layer will take a long time to develop, but its architectural improvements will most likely improve its stability in time.

I am going to work on the TTY layer during my internship at Snow B.V., which will only last for 20 weeks. Because 20 weeks is not sufficient to implement all the proposed changes, work will continue after my graduation.

Appendix C

Sample Highlight report - Dutch

Snow B.V.
Koeweistraat 12
4181 CD Waardenburg
Postbus 72
4180 BB Waardenburg
tel. 0418-653 333
fax 0418-653 666

info@snow.nl
http://snow.nl
k.v.k. 17101631
bank 68.05.08.430
giro 5377513

Highlight Report

Snow B.V.

15 april 2008

Versiebeheer

Versie	Datum	Gewijzigd door
1.0	11 apr 2008	e.schouten@snow.nl

Distributie

Naam	Functie/rol	E-mail
Jos Jansen	Opdrachtgever	jos@snow.nl
Joost Helberg	Toezichthouder	joost@snow.nl
Patrick Zwegers	Waarnemer	p.zwegers@fontys.nl

Akkoord

Naam	Paraaf	Datum
Jos Jansen		
Joost Helberg		
Patrick Zwegers		

Inhoudsopgave

1 Doel van dit document

Het verstrekken van een samenvatting van de voortgang van de laatste week. Deze samenvatting kan gebruikt worden om te kijken of de uitvoer van het project nog op schema ligt.

2 Status

De terminal implementatie is nu vrij ver gevorderd. Er is nog een functie die echt ontbreekt, namelijk ondersteuning voor de carrier/connection lijnen. Dit betekent dat de TTY laag op dit moment niet tijdens het openen van het apparaat blijft wachten tot er een carrier beschikbaar is.

3 Voortgang week 10

Deze week is er weer hard gewerkt aan de code van de TTY implementatie. Daarnaast zijn er ook diverse zaken geregeld voor mijn bezoek aan BSDCan in Ottawa, Canada, zoals het regelen van een vliegticket en een slaapplek.
Op maandag is er gewerkt aan de `ucom(4)` driver voor FreeBSD. Deze driver zorgt ervoor dat USB-to-serial kabels gebruikt kunnen worden onder FreeBSD, zoals bijvoorbeeld de Prolific PL2303 (`uplcom(4)`). Het porten ging redelijk snel, alleen werd er al snel ontdekt dat een subtiele wijziging in het ontwerp in de TTY laag ervoor zorgde dat de driver snel crasht.
In de oude TTY implementatie kan de driver een flag aanzetten om aan te geven dat het apparaat bezig is (`TS_BUSY`). De nieuwe terminal laag legt deze verantwoordelijkheid in de driver. De TTY laag geeft meldingen dat er activiteiten zijn. De driver is zelf verantwoordelijk voor het bepalen wanneer de activiteiten afgehandeld worden. Dit is opgelost door de `ucom(4)` driver een eigen busy-vlag te geven.
Op dinsdag zijn enkele wijzigingen teruggedraaid. Er is namelijk een script geschreven waarmee makkelijk patches tegen FreeBSD gemaakt kunnen worden. Hieruit bleek dat er enkele wijzigingen zijn doorgevoerd die teruggedraaid kunnen worden. Ook is de `pstat(8)` applicatie aangepast om fatsoenlijk terminal statistieken uit kernel coredumps te halen.
Woensdag zijn enkele crash bugs in de kernel gerepareerd. Ook is er ondersteuning toegevoegd om break aan en uit te zetten (`TIOCSBRK` en `TIOCCBRK`). Ook is er een opzetje gemaakt voor de modem funcionaliteit (Carrier Detect, Data Terminal Ready, etc). Helaas worden nog niet de correcte vlaggen ingesteld bij het openen van de TTY's.
Donderdag heb ik eindelijk een functie toegevoegd aan de TTY laag die al heel lang op de TODO lijst stond, namelijk ondersteuning voor `read()`'s met timers (`VTIME` en `VMIN`). Hiermee kan aangegeven worden dat `read()` moet wachten tot een bepaalde hoeveelheid bytes is ontvangen, of totdat een bepaalde tijd is verstreken. Na dit toegevoegd te hebben, werkte Minicom naar behoren. Het is nu mogelijk om vanuit een systeem met de nieuwe TTY laag een verbinding te maken naar een ander systeem. Omdat de correcte modem vlaggen niet worden ingesteld, detecteert het andere systeem de connectie helaas niet.

Op vrijdag heb ik vlaai uitgedeeld op kantoor, omdat ik zaterdag jarig ben. Aan het einde van de morgen heb ik een Sun Ultra 5 onder mijn bureau gevonden. Deze heb ik ook voorzien van FreeBSD, zodat ik de `ofw_console(4)` driver kon porten. De TTY laag lijkt nu correct te functioneren onder SPARC64.

4 Vooruitblik

Ondanks dat er nog bepaalde zaken op de TODO staan (modem ondersteuning), zal de hoeveelheid tijd die besteed wordt aan het programmeren de komende weken langzaam aan afnemen. Er moet begonnen worden aan het stageverslag en ook zal een presentatie voorbereid worden voor in Ottawa.

4.1 Mogelijke issues

Op dit moment zijn er geen issues bekend.

Appendix D

Sample Exception report - Dutch

Snow B.V.
Koeweistraat 12
4181 CD Waardenburg
Postbus 72
4180 BB Waardenburg
tel. 0418-653 333
fax 0418-653 666

info@snow.nl
http://snow.nl
k.v.k. 17101631
bank 68.05.08.430
giro 5377513

Exception Report

Snow B.V.

17 april 2008

Versiebeheer

Versie	Datum	Gewijzigd door
1.0	14 mrt 2008	e.schouten@snow.nl

Distributie

Naam	Functie/rol	E-mail
Jos Jansen	Opdrachtgever	jos@snow.nl
Joost Helberg	Toezichthouder	joost@snow.nl

Akkoord

Naam	Paraaf	Datum
Jos Jansen		
Joost Helberg		

Inhoudsopgave

1 Doel van dit document

Het doel van dit document is het melden van een overschrijding grenswaarden van het project.

2 Oorzaak en omvang afwijking

Nadat de syscons driver geport was naar de nieuwe TTY laag, bleken er vreemde panics te ontstaan tijdens het afsluiten, die veroorzaakt werden door de `revoke(2)` system call. Na onderzoek is gebleken dat deze system call een grote ontwerpfout heeft:

- De system call roept de `d_close()` routine van de device driver aan. Deze routine wordt normaal alleen aangeroepen wanneer de laatste descriptor naar het apparaat gesloten is.
- Threads kunnen zich op dat moment in de device driver begeven, waardoor het apparaat gesloten wordt terwijl het in gebruik is.
- De oude TTY laag werkt hier met een trial and error methode omheen.
- Andere drivers die hier niet van bewust zijn, kunnen met gemak het systeem omlaag brengen (BPF).

2.1 Consequenties afwijking

Het is met de huidige opzet niet mogelijk om in de nieuwe laag robuust deze functionaliteit te implementeren. Ook is het duidelijk te zien dat de stabliteit van het systeem ernstig wordt belemmerd door deze system call.

2.2 Alternatieven

Omdat het ondersteunen van `revoke(2)` niet van belang is voor iedere device driver, is de beste oplossing om een `d_revoke()` toe te voegen aan de device driver code, waarmee de driver zelf de verantwoordelijkheid krijgt om deze functionaliteit te implementeren.

3 Impact op Business Case

De business case zal ongewijzigd blijven.

4 Risico's

Deze wijziging zal het drivermodel van FreeBSD incompatibel maken met het oude model. `revoke(2)` zal voor bepaalde drivers niet meer naar behoren werken.

5 Projecttoleranties

Aangezien het project goed op schema ligt, zal het project ruim binnen de toleranties blijven.

5.1 Stagetoleranties

Ondanks dat er lichtelijk wordt afgedwaald van de oorspronkelijk gestelde opdracht, zal de wijziging toch doorgevoerd moeten worden om de stage naar behoren af te kunnen ronden.

6 Aanbeveling van de Projectleider

De projectleider gaat akkoord.

7 Referenties

- The Design and Implementation of the 4.3BSD Operating System.
- `revoke(2)` manual page.