VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

LINUXOVÁ EMULAČNÍ VRSTVA VE FREEBSD

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. ROMAN DIVÁCKÝ
AUTHOR

BRNO 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

LINUXOVÁ EMULAČNÍ VRSTVA VE FREEBSD
LINUX EMULATION LAYER IN FREEBSD

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. ROMAN DIVÁCKÝ
AUTHOR

VEDOUCÍ PRÁCE Ing. RUDOLF ČEJKA
SUPERVISOR

BRNO 2007

Abstrakt

Tato diplomová práce se zabývá aktualizací Linuxové emulační vrstvy (takzvaného Linuxulatoru). Úloha spočívala v aktualizaci emulační vrstvy tak, aby odpovídala funkcionalitě Linuxu verze 2.6. Jako referenční implementace byl zvolen Linux verze 2.6.16. Koncept je volně založen na implementaci v NetBSD. Většina práce byla dokončena v létě 2006 v rámci studentského programu Google Summer of Code. Zaměřil jsem se na implementaci podpory NTPL (nová posixová vláknová knihovna) do emulační vrstvy, včetně TLS (lokální uložiště vlákna), futexů (rychlé mutexy v user space), měnění PIDu a některé další věci. Mnoho menších problémů bylo nalezeno a opraveno během této práce. Moje práce byla integrována do vývojového stromu FreeBSD a bude distribuována v nadcházející verzi 7.0. Emulační tým, včetně mě, pracuje na tom, aby emulace Linuxu 2.6 byla implicitní emulací ve FreeBSD.

Klíčová slova

FreeBSD, Linuxová emulace, Linuxulator

Abstract

This masters thesis deals with updating the Linux emulation layer (so called Linuxulator). The task was to update the layer to match the functionality of Linux 2.6. As a reference implementation, the Linux 2.6.16 kernel was chosen. The concept is loosely based on the NetBSD implementation. Most of the work was done in the summer of 2006 as a part of the Google Summer of Code students program. The focus was on bringing the NPTL (new posix thread library) support into the emulation layer, including TLS (thread local storage), futexes (fast user space mutexes), PID mangling, and some other minor things. Many small problems were identified and fixed in the process. My work was integrated into the main FreeBSD source repository and will be shipped in the upcoming 7.0R release. We, the emulation development team, are working toward making the Linux 2.6 emulation the default emulation layer in FreeBSD.

Keywords

FreeBSD, Linux emulation, Linuxulator

Citace

Roman Divácký: Linuxová emulační vrstva ve FreeBSD, diplomová práce, Brno, FIT VUT v Brně, 2007

Linuxová emulační vrstva ve FreeBSD

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Rudolfa Čejky

....
Roman Divácký
17. května 2007

Poděkování

Thanks goes to Alexander Leidinger, Rudolf Cejka, Konstantin Belousov, David Yeske, Robert Watson, David Xu, Google etc.

Obsah

Kapitola 1

Introduction

In the last few years open source Unix based operating systems started to be widely deployed on server and client machines. Among these operating systems I'd like to point out two: FreeBSD, for its BSD heritage, time proven code base and many interesting features and Linux for its wide user base, enthusiastic open developer community and support from large companies. FreeBSD tends to be used on server class machines serving heavy duty networking tasks with less usage on desktop class machines for ordinary users. While Linux has the same usage on servers, but it is used much more by home based users. This leads to a situation where there are many binary only programs available for Linux that lack support in FreeBSD.

Naturally a need for the ability to run Linux binaries on a FreeBSD system arises and this is what this thesis deals with, the emulation of the Linux kernel in the FreeBSD operating system.

During the Summer of 2006 Google Inc. sponsored a project which focused on extending the Linux emulation layer (so called Linuxulator) in FreeBSD to include Linux 2.6 facilities. This thesis is written about part of this project.

The First chapter introduces the thesis, and explains why it was written and what it deals with.

The Second chapter describes the operating systems in questions and how emulation of Unix on Unix works.

The Third chapter provides a detailed description of how Linux emulation in FreeBSD works. It explains everything from low level methods like trapframes and syscalls convention, to overall architecture design focusing mainly on the way we do emulation of Linux 2.6.

Appendix A gives us some background about the inner processes of the Google Summer of Code.

Appendix B provides code metrics and the actual code delivered in the thesis project.

Kapitola 2

A look inside...

In this chapter we are going to describe every operating system in question. How they deal with syscalls, trapframes etc. all the lowlevel stuff. We also describe the way they understand common Unix primitives like what a PID is, what a thread is, etc. In the third subsection we talk about how Unix on Unix emulation could be done in general.

2.1 What is Unix

Unix is an operating system with a long history that has influenced almost every other operating system currently in use. Starting in the 1960s, its development continues to this day (although in different projects). Unix development soon forked into two main ways - the BSDs and System III/V families. They mutually influenced themselves by growing a common Unix standard. Among the contributions originated in BSD we can name virtual memory, TCP/IP networking, FFS, and many others. The System V branch contributed to SysV inter process communication primitives, copy-on-write, etc. Unix itself doesn't exist anymore but its ideas have been used by many other operating systems world wide thus forming so called Unix-like operating systems. These days the most influential ones are Linux, Solaris, and possibly (to some extent) FreeBSD. There are in-company Unix derivatives (AIX, HP-UX etc.), but these have been more and more migrated to the aforementioned systems. Lets summarize typical Unix characteristics...

2.1.1 Technical details

Every running program constitutes a *process* that represents a state of the computation. Running process is divided between *kernel-space* and *user-space*. Some operations can be done only from kernel space (dealing with hardware etc.), but the process should spend most of its lifetime in user space. The kernel is where the management of the processes, hardware, and lowlevel details takes place. The kernel provides a standard unified Unix API to user space. The most important are:

Communication between kernel and user space process

Common Unix API defines a *syscall* as a way to issue commands from a user space process to the kernel. The most common implementation is either by using an *interrupt* or specialized instruction (think of SYSENTER/SYSCALL instructions for ia32). Syscalls are defined by a number. For example in FreeBSD, syscall number 85 is the swapon syscall and syscall number 132 is mkfifo. Some syscalls need parameters which are passed from user-space to kernel-space in various ways (implementation dependant). Syscalls are synchronous.

Another possible way to communicate is by using a *trap*. Traps occur asynchronously after some event occurs (division by zero, page fault etc.). A trap can be transparent for a process (page fault) or can result in a reaction like signal sent (division by zero).

Communication between processes

There are other APIs (System V ipc, shared memory etc.) but the single most important API is *signal*. Signals are sent by a process or by the kernel. Signals are received by process. Some signals can be ignored or handled by a user supplied routine, some result in predefined action that cannot be altered/ignored.

Process management

Kernel instances first process in the system (so called *init*). Every running process can create its identical copy using *fork* syscall. Some slightly modified versions of this syscall were introduced but the basic semantic is the same. Every running process can morph into some other process using *exec* syscall. Some modifications of this syscall were introduced but all serve the same basic purpose. Process ends its life by calling *exit* syscall. Every process is identified by a unique number called *PID*. Every process has a defined parent (identified by its PID).

Thread management

Traditional Unix doesn't define any API nor implementation for threading. Posix defines its threading API but the implementation is undefined. Traditionally there were two ways of implementing threads. Handling them as separate processes (1:1 threading) or envelope the whole thread group in one process and managing the threading in userspace (1:N threading). Comparing main features of each approach:

1:1 threading

- - heavyweight threads
- - user cannot alter scheduling (slightly mitigated by Posix API)
- + no syscalls wrapping necessary
- + can utilize multiple CPUs

1:N threading

- + lightweight threads
- + user can easily alter scheduling
- - syscalls must be wrapped
- - cannot utilize more than one CPU

2.2 What is FreeBSD

The FreeBSD project is one of the oldest open source operating systems currently available for daily use. Its a direct descendant of the genuine Unix so it could be claimed that it's a true Unix although licensing issues doesn't permit so. The start of the project dates back to the early 1990's when a crew of fellow BSD users patched the 386BSD operating system. Based on this patchkit a new operating system arose named FreeBSD for its liberal license. Another group created the NetBSD operating system with different goals in mind. We'll focus on FreeBSD.

FreeBSD is a modern Unix based operating system with all its features. Preemptive multitasking, multiuser facilities, TCP/IP networking, memory protection, symmetric multiprocessing support, virtual memory with merged VM and buffer cache, you name it its all there. One of interesting and extremely useful features is ability to emulate other Unix operating systems. As of December 2006 and 7-CURRENT development, there is support for:

1. FreeBSD32 - emulation of 32bit FreeBSD on 64bit FreeBSD. Useful for amd64 FreeBSD.
2. IA32 - emulation of i386 FreeBSD on Itanium FreeBSD.
3. Linux - emulation of Linux operating system on FreeBSD.
4. NDIS - emulation of Windows networking drivers interface.
5. NetBSD - emulation of NetBSD operating system.
6. PECoff - support for PECoff FreeBSD executables.
7. svr4 - emulation of System V revision 4 Unix.

Actively developed emulations are the Linux layer and various FreeBSD-on-FreeBSD layers. Others are not known to work nor usable these days.

FreeBSD development happens in a central CVS repository where only a selected team of so-called commiters can write. This repository posses several branches and most interesting is HEAD branch, in FreeBSD nomenclature called -CURRENT, and RELENG_X branches, where X stands for a number indicating version of FreeBSD. As of December 2006, there are development branches for the RELENG_6 branch (for 6.x development) and RELENG_5 branch (for 5.x development). Other branches are closed and not actively maintained.

Historically the active development was done in the HEAD branch so it was considered extremely unstable and „can break at any time“ branch. This is not true anymore as the Perforce (commercial version control system) repository was introduced and active development happens there. There are many branches in Perforce where development of certain parts of the system happen and these branches are from time to time merged back to the main CVS repository thus effectively putting the given feature to FreeBSD operating system. As it was with rdivacky_linuxolator branch where development of this thesis code was going on.

More info about the FreeBSD operating system can be found at [2].

2.2.1 Technical details

FreeBSD is traditional Unix in the sense of dividing the run of a process into two halves. Kernel space and user space run. There are two types of process entry to the kernel: a syscall and a trap. There's only one way to return. In the subsequent sections we will describe the 3 gates to/from the kernel. All the description applies to i386 architecture as the Linuxulator exists only there but it's similar for other archs. The information was taken from [1] and source codes.

System entries

FreeBSD has an abstraction called an „execution class loader“, which is a wedge into the execve() syscall. This employs a structure *sysentvec* which describes an executable ABI. It contains things like errno translation table, signal translation table, various functions to serve syscall needs (stack fixup, coredumping etc.). Every ABI the FreeBSD kernel wants to support must define this structure, as it's used later in the syscall processing code and in some other places. System entries are handled by trap handlers where we can access both kernel space and user space at once.

Syscalls

Syscalls on FreeBSD are issued by executing interrupt *0x80* with register *%eax* set to a desired syscall number with arguments passed on the stack.

When a process issues an interrupt *0x80*, the *int0x80_syscall* trap handler is issued (defined in sys/i386/i386/exception.s) which prepares (= copies them on to the stack) arguments for a call to a C function *syscall()* (defined in sys/i386/i386/trap.c) which processes the passed in trapframe. The processing consists of preparing the syscall (depending on the sysvec entry), determining if the syscall is 32bit or 64bit (changes size of the params), then the parameters are copied, including the syscall. Next the actual syscall function is executed with processing of the return (special cases for ERESTART and EJUSTRETURN errors). Finally an *userret()* is scheduled switching the process back to userspace. The parameters to the actual syscall handler are passed in the form of *struct thread *td, struct syscall_args *args* where the second parameter is a pointer to the copied in structure of parameters.

Traps

Handling of traps in FreeBSD is similar to the handling of syscalls. Whenever a trap occurs an assembler handler is called. It's chosen between *alltraps*, *alltraps_with_regs_pushed* or *calltrap* depending on the type of the trap. This handler prepares arguments for a call to a C function *trap()* (defined in sys/i386/i386/trap.c) which then processes the occurred trap. After the processing it might send a signal to the process and/or exit to userland using *userret()*.

Exits

Exits from kernel to userspace happens using the assembler routine *doreti* regardless of whether the kernel was entered with a trap or with a syscall. This restores the program status from the stack and returns to the userspace.

Unix primitives

FreeBSD operating system adheres to the traditional Unix scheme where every process has a unique identification number, the so called *PID*. PID numbers are allocated either linearly or randomly ranging from 0 to PID_MAX. The allocation of PID numbers is done using linear searching of PID space. Every thread in a process receives the same PID number as result of the *getpid()* call.

There are currently two ways to implement threading in FreeBSD. The first way is M:N threading followed by the 1:1 threading model. The default library used is M:N threading (libpthread) and you can switch at runtime to 1:1 threading (libthr). The plan is to switch to 1:1 library by default, soon. Those two libraries although using the same kernel primitives are accessed using different api(es). The M:N library uses the *kse_** family of syscalls while the 1:1 library uses the *thr_** family of syscalls. Because of this there is no general concept of thread ID shared between kernel and userspace. Of course both threading libraries implement the pthread thread ID API. Every kernel thread (as described by *struct thread*) has td_tid identifier but this is not directly accessible from userland and solely serves the kernels needs. It is also used for 1:1 threading library as pthread's thread ID but handling of this is internal to the library and cannot be relied on.

As stated previously there are two implementations of threading in FreeBSD. The M:N library divides the work between kernel space and userspace. Thread is an entity that gets scheduled in the kernel but it can represent various number of userspace threads. M userspace threads get mapped to N kernel threads thus saving resources while keeping the ability to exploit multiprocessor parallelism. Further information about the implementation can be obtained from the man page or [1]. The 1:1 library directly maps a userland thread to a kernel thread thus greatly simplifying the scheme. None of these designs implement a fairness mechanism (such a mechanism was implemented but it was removed recently because it caused serious slowdown and made the code more difficult to deal with).

2.3 What is Linux

Linux is a Unix-like kernel originally developed by Linus Torvalds, and now being contributed to by a massive crowd of programmers all around the world. From its mere beginnings to todays, with wide support from such companies as IBM or Google, Linux is being associated with its fast development pace, full hardware support and benevolent dictator model of organization.

Linux development started in 1991 as a hobbyist project at University of Helsinki in Finland. Since then it has obtained all the features of a modern Unix OS - multiprocessing, multiuser support, virtual memory, networking, basically everything is there. There are also very advanced features like virtualization etc.

As of 2006 Linux seems to be the most widely used open source operating system with support from independent software vendors like Oracle, RealNetworks, Adobe etc. Most of the commercial software distributed for Linux can only be obtained in a binary form so recompilation for other operating systems is impossible.

Most of the Linux development happens in a GIT version control system. Git is a distributed system so there is no central source of the Linux code, but some branches are considered prominent and official. The version number scheme implemented by Linux consists of four numbers A.B.C.D. Currently development happens in 2.6.C.D where C represents major version where new features are added/changed and D is a minor version for bugfixes only.

More information can be obtained from [4].

2.3.1 Technical details

Linux follows traditional Unix scheme of dividing the run of a process in two halves, the kernel and user space. The kernel can be entered in two ways: a trap and a syscall. The return is handled only one way. The further description applies to Linux 2.6 on i386 architecture. This information was taken from [3].

Syscalls

Syscalls in Linux are performed (in userspace) using *_syscallX* macros where X substitutes a number representing number of parameters of the given syscall. This macro translates to a code that loads *%eax* register with a number of the syscall and executes interrupt *0x80*. After this *__syscall_return* is called which translates negative return value to positive errno and sets res to -1 in the case of an error. Whenever an interrupt *0x80* is called the process enters the kernel in *system_call* trap handler. This routine saves all registers on the stack and calls the selected syscall entry. Note that the Linux calling convention expects parameters to the syscall to be passed via registers as shown here:

1. parameter —> *%ebx*
2. parameter —> *%ecx*
3. parameter —> *%edx*
4. parameter —> *%esi*
5. parameter —> *%edi*
6. parameter —> *%ebp*

There are some exceptions to this where Linux uses different calling convention (most notably the clone syscall).

Traps

In *arch/i386/kernel/traps.c* the trap handlers are made and most of these handlers live in *arch/i386/kernel/entry.S* where handling of the traps happens.

Exit

Return from the syscall is managed by *syscall_exit* which checks for the process having unfinished work, then checks whether we used user-supplied selectors. If this happens stack fixing is applied and finally restoring the registers from the stack and returning to the userspace.

Unix primitives

Linux operating system in its 2.6 version redefined some of the traditional Unix primitives, notably *PID*, *TID* and thread. *PID* is defined not to be unique for every process, so for some processes (threads) *getppid()* returns the same value. Unique identification of process is provided by *TID*. This is because NPTL (new posix thread library) defines threads to be normal processes (so called 1:1 threading). Spawning a new process in Linux 2.6 happens using *clone* syscall (fork variants are reimplemented using it). This *clone* syscall defines a set of flags that affect behaviour of the cloning process regarding thread implementation. The semantic is a bit fuzzy as there is no single flag telling the syscall to create a thread.

Implemented clone flags are:

1. CLONE_VM - processes share their memory space
2. CLONE_FS - share umask, cwd and namespace
3. CLONE_FILES - share open files
4. CLONE_SIGHAND - share signal handlers and blocked signals
5. CLONE_PARENT - share parent
6. CLONE_THREAD - be thread (further explanation below)
7. CLONE_NEWNS - new namespace
8. CLONE_SYSVSEM - share SysV undo structures
9. CLONE_SETTLS - setup TLS at supplied address
10. CLONE_PARENT_SETTID - set TID in the parent
11. CLONE_CHILD_CLEARTID - clear TID in the child
12. CLONE_CHILD_SETTID - set TID in the child

CLONE_PARENT sets the real parent to the parent of the caller. This is useful for threads because if thread A creates thread B we want thread B to be parented to the parent of the whole thread group. CLONE_THREAD does exactly the same thing as CLONE_PARENT, CLONE_VM and CLONE_SIGHAND, rewrites *PID* to be the same as *PID* of the caller, sets exit signal to be none and enters the thread group. CLONE_SETTLS sets up GDT entries for TLS handling. CLONE_*_*TID this set of flags sets/clear user supplied address to *TID* or 0.

As you can see the CLONE_THREAD does most of the work and doesn't seem to fit the scheme very well. The original intention is unclear (even for authors, according to comments in the code) but I think originally there was one threading flag which was then parcelled among many other flags but this separation was never fully finished. Its also unclear what this partition is good for as glibc doesn't use that so only hand-written use of the clone permits a programmer to access this features.

For non-threaded programs the *PID* and *TID* are the same. For threaded programs the first thread *PID* and *TID* are the same and every thread created shares the same *PID* and gets assigned an unique *TID* (because CLONE_THREAD is passed in) also parent is shared for all processes forming this threaded program.

The code that implements *pthread_create* in NPTL defines the clone flags like this:

```
  int clone_flags = (CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGNAL
                     | CLONE_SETTLS | CLONE_PARENT_SETTID
                     | CLONE_CHILD_CLEARTID | CLONE_SYSVSEM
#if __ASSUME_NO_CLONE_DETACHED == 0
                     | CLONE_DETACHED
#endif
                     | 0);
```

the CLONE_SIGNAL is defined like

```
#define CLONE_SIGNAL            (CLONE_SIGHAND | CLONE_THREAD)
```

the last 0 means no signal is sent when any of the threads exits.

2.4 What is emulation

Emulation according to a dictionary definition is the ability of a program or device to imitate another program or device. This is achieved by providing the same reaction to a given stimulus as emulated object. In practice, the software world mostly sees three types of emulation - a program used to emulate a machine (QEMU, various game console emulators etc.), software emulation of a hardware facility (OpenGL emulators, floating point units emulation etc.) and operating system emulation (either in kernel of the operating system or as a userspace program).

Emulation is usually used in a place where using the original component is not feasible or possible at all. For example someone might want to use a program developed for a different operating system than he's using. Then emulation comes in handy. Sometimes there is no other way but to use emulation - the hardware device you are trying to use doesn't not exist (yet/anymore) then there is no other way but emulation. This happens often when porting an operating system to a new (non-existent) platform. Sometimes it's just cheaper to emulate.

Looking from an implementation point of view, there are two main approaches to the implementation of emulation. You can either emulate the whole thing - accepting possible inputs of the original object, maintaining inner state and emitting correct output based on the state and/or input. This kind of emulation doesn't require any special conditions and basically can be implemented anywhere for any device/program. The drawback is that implementing such emulation is quite difficult, time-consuming and error-prone. In some cases we can use a simpler approach. Imagine you want to emulate a printer that prints from left to right on a printer that prints from right to left. Its obvious that there is no need for a complex emulation layer but simple reversing of the printed text is sufficient. Sometimes the emulating environment is very similar to the emulated one so only a thin layer of some translation is necessary to provide fully working emulation! As you can see this is much less demanding to implement, so less time-consuming and error-prone than the previous approach. But the necessary condition is that the two environments must be similar enough. The third approach combines the two previous. Most of the time the objects do not provide the same capabilities so in a case of emulating the more powerful one on the less powerful we have to emulate the missing features with full emulation described above.

This master thesis deals with emulation of Unix on Unix which is exactly the case where only a thin layer of translation is sufficient to provide full emulation. The Unix API consists of a set of syscalls which are usually self contained and don't affect some global kernel state.

There are a few syscalls that affect inner state but this can be dealt with by providing some structure maintaining the extra state.

No emulation is perfect and emulations tend to lack some parts but this usually doesn't cause any serious drawbacks. Imagine a game console emulator that emulates everything but music output. No doubt that the games are playable and one can use the emulator. It might not be that comfortable as the original game console but its an acceptable compromise between price and comfort.

The same goes with the Unix API. Most programs can live with a very limited set of syscalls working. Those syscalls tend to be the oldest ones (read/write, fork family, signal handling, exit, socket API) hence easy to emulate because their semantics is shared among all Unixes that exist todays.

Kapitola 3

Emulation

3.1 How emulation works in FreeBSD

As stated earlier FreeBSD supports running binaries from several other Unixes. This works because FreeBSD has an abstraction called the „execution class loader“. This wedges into the execve() syscall, so when execve() is about to execute a binary it examines its „type“. There are basically two types of binaries in FreeBSD. Shell-like text scripts which are identified by „#!“ as their first two characters and normal (typically ELF) binaries which are a representation of a compiled executable object. The vast majority (one could say all of them) of binaries in FreeBSD are ELF type. ELF files contain a header which specifies the OS ABI for this ELF file. By reading this information, the operating system can accurately determine what type of binary this file is.

Every OS ABI must be registered in the FreeBSD kernel. This applies to the FreeBSD native OS ABI as well. So when execve() executes a binary it iterates through the list of registered APIs and when it finds the right one it starts to use the information contained in the OS ABI description (its syscall table, errno translation table etc.). So every time the process calls a syscall, it uses its own set of syscalls instead of some global one. This effectively enables very elegant and easy support for execution of various binary formats.

The nature of emulation of different OSes (and also some other subsystems) led developers to invite a handler event mechanism. There are various places in the kernel where a list of event handlers are called. Every subsystem can register an event handler and they are called accordingly. For example when a process exits there's a handler called that possibly cleans up whatever the subsystem needs to be cleaned.

Those simple facilities provide basically all that's needed for the emulation infrastructure and in fact these are basically the only things necessary to implement the Linux emulation layer.

3.2 Common primitives in the FreeBSD kernel

Emulation layer(s) need some support from the operating system. I am going to describe some of the supporting primitives in the FreeBSD operating system.

3.2.1 Locking primitives

Contributed by Attilio Rao

The FreeBSD synchronization primitives set is based on the idea to supply a rather huge number of different primitives in a way that the better one can be used for every particular, appropriate situation.

To an high level point of view you can consider three kinds of synchronization primitives in the FreeBSD kernel:

1. atomic operations / memory barriers
2. locks
3. scheduling barriers

Below there are descriptions for the 4 families. For every lock, you should really check the linked manpage (where possible) for more detailed explanations.

Atomic operations / memory barriers

Atomic operations are implemented through a set of functions performing simple aritmetics on memory operands in an atomic way with respect to external events (interrupts, preemption, etc.). Atomic operations can guarantee atomicity just on small data types (in the magnitude order of the 'long' architecture C data type), so should be rarely used in end-level code directly, if not only for very simple operations (like flag setting in a bitmap, for example). In fact, it is rather simple and common to write down a wrong semantic just based on atomic operations (usually referred as lock-less). The FreeBSD kernel offers a way to perform atomic operations in conjunction with a memory barrier. The memory barriers will guarantee that an atomic operation will happen following some specified ordering in respect to other memory accesses. For example, if we need that an atomic operation happens just after all other pending writes (in terms of instructions reordering buffers activities) are completed, we need to explicitly use a memory barrier in conjunction to this atomic operation. So it is simple to understand why memory barriers play a key role for higher-level locks building (just as refcounts, mutexes, etc.). For a detailed explanatory on atomic operations, please refer to atomic(9). It is far, however, noting that atomic operations (and memory barriers too) should, ideally, only be used for building front-ending locks (as mutexes).

Refcounts

Refcounts are interfaces for handling reference counters. They are implemented through atomic operations and are intended to be used just for cases where the reference counter is the only one thing to be protected, so even something like a spin-mutex is deprecated. Using the refcount interface for structures where a mutex is already used is often wrong since we should probably close the reference counter in some already protected paths. A manpage discussing refcount doesn't exist currently, just check sys/refcount.h for an overview of the existing API.

Locks

FreeBSD kernel has huge class of locks. Every lock is defined by some peculiar properties, but probably the most important is the event linked to contesting holders (or, in other terms, the behaviour of threads unable to acquire the lock). FreeBSD locking scheme presents 3 different behaviours for contenders:

1. 0. spinning
2. 1. blocking
3. 2. sleeping

(NOTE: numbers are not casual)

Spinning locks

Spin locks let waiters to spin until they can't acquire the lock. An important matter do deal with is that when a thread contests on a spin lock it is not descheduled. Since FreeBSD kernel is preemptive, this exposes spin lock at the risk of deadlocks that can be solved just disabling interrupts while they are acquired. For this and other reasons (like lack of priority propagation support, poorness in load balancing schemes between CPUs, etc.), spin locks are intended to protect very small paths of code, or ideally to not be used at all if not explicitly requested (explained later).

Blocking

Block locks let waiters to be descheduled and blocked until the lock owner doesn't drop it and wakes up one or more contenders. In order to avoid starvation issues, blocking locks do priority propagation from the waiters to the owner. Block locks must be implemented through the turnstile interface and are intended to be the most used kind of locks in the kernel, if no particular conditions are met.

Sleeping

Sleep locks let waiters to be descheduled and fall asleep until the lock holder doesn't drop it and wakes up one or more waiters. Since sleep locks are intended to protect large paths of code and to cater asynchronous events, they don't do any form of priority propagation. They must be implemented through the sleepqueue(9) interface.

The order used to acquire locks is very important, not only for the possibility to deadlock due at lock order reversals, but even because lock acquisition should follow specific rules linked to locks natures. If you give a look at the table above, the practical rule is that if a thread holds a lock of level n (where the level is the number listed close to the kind of lock) it is not allowed to acquire a lock of superior levels, since this would break the specified semantic for a path. For example, if a thread holds a block lock (level 1), it is allowed to acquire a spin lock (level 0) but not a sleep lock (level 2), since block locks are intended to protect smaller paths than sleep lock (these rules are not about atomic operations or scheduling barriers, however).

This is a list of lock with their respective behaviours:

1. spin mutex - spinning - mutex(9)
2. sleep mutex - blocking - mutex(9)
3. pool mutex - blocking - mtx_pool(9)
4. sleep family - sleeping - sleep(9) pause tsleep msleep msleep_spin msleep_rw msleep_sx
5. condvar - sleeping - condvar(9)

6. rwlock - blocking - rwlock(9)
7. sxlock - sleeping - sx(9)
8. lockmgr - sleeping - lockmgr(9)
9. semaphores - sleeping - sema(9)

Among these locks only mutexes, sxlocks, rwlocks and lockmgrs are intended to handle recursion, but currently recursion is only supported by mutexes and lockmgrs.

Scheduling barriers

Scheduling barriers are intended to be used in order to drive scheduling of threading using it. They consist mainly of 3 different stubs:

1. critical sections (and preemption)
2. sched_bind
3. sched_pin

Generally, these should be used only in a particular context and even if they can often replace locks, they should be avoided because they don't let the diagnose of simple eventual problems with locking debugging tools (as WITNESS).

Critical sections

The FreeBSD kernel has been made preemptive basically to deal with interrupt threads. In fact, in order to avoid high interrupt latency, time-sharing priority threads can be preempted by interrupt threads (in this way, they don't need to wait to be scheduled as the normal path previews). Preemption, however, introduces new racing points that need to be handled as well. Often, in order to deal with preemption, the simplest thing to do is to completely disable it. A critical section defines a piece of code (borderlined by the pair of functions critical_enter()/critical_exit()) where preemption is guaranteed to not happen (until the protected code is fully executed). This can often replace a lock effectively but should be used carefully in order to not lose the whole advantage that preemption brings.

sched_pin/sched_unpin

Another way to deal with preemption is the sched_pin() interface. If a piece of code is closed in the sched_pin()/sched_unpin() pair of functions it is guaranteed that the respective thread, even if it can be preempted, it will always be executed on the same CPU. Pinning results very effective in the particular case we have to access at per-cpu datas and we assume other threads won't change those data. The not-changing condition, will determine a critical section as a too strong condition for our code.

sched_bind/sched_unbind

sched_bind is an API used in order to bind a thread to a particular CPU for all the time it is executed the code, until a sched_unbind() function call doesn't unbind it. This feature has a key role in situations where you can't trust about the current state of CPUs (as, for example, very early stages of boot), as you want to avoid your thread to migrate on inactive CPUs. Since sched_bind/sched_unbind manipulate internal scheduler structures, they need to be enclosed in sched_lock acquisition/releasing when used.

3.2.2 Proc structure

Various emulation layers sometime require some additional per-process data. It can manage separate structure (a list, a tree etc.) containing these data for every process but this tends to be slow and memory consuming. To solve this problem the FreeBSD proc structure contains *p_emuldata* which is a void pointer to some emulation layer specific data. This proc entry is protected by the proc mutex.

The FreeBSD proc structure contains a *p_sysent* entry that identifies which ABI this process is running. In fact it's a pointer to the *sysentvec* described above. So by comparing this pointer to the address where *sysentvec* structure for the given ABI is stored we can effectively determine whether the process belongs to our emulation layer. The code typically looks like:

```
if (__predict_true(p->p_sysent != &elf_Linux_sysvec))
        return;
```

As you can see we effectively use the *__predict_true* modifier to collapse the most common case (FreeBSD process) to a simple return operation thus preserving high performance. This code should be turn into a macro because currently its not very flexible, ie. we don't support Linux64 emulation nor $A.OUT$ Linux processes on i386.

3.2.3 VFS

The FreeBSD VFS subsystem is very complex but the Linux emulation layer uses just a small subset via a well defined API. It can either operate on vnodes or file handlers. Vnode represents a virtual vnode, ie. representation of a node in VFS. Another representation is a file handler which represents an opened file from the perspective of a process. A file handle can represent a socket or an ordinary file. A file handler contains a pointer to its vnode. More then one file handler can point to the same vnode.

namei

The *namei* routine is a central entry point to pathname lookup and translation. It traverses the path point by point from starting point to the end using *lookup* function which is internal to VFS. The namei can cope with symlinks, absolute and relative paths. When a path is looked up using namei it is inputed to the name cache. This behaviour can be supressed. This routine is used all over the kernel and its performance is very critical.

vn_fullpath

The vn_fullpath function takes the „best effort“ to traverse VFS name cache and returns a path for a given (locked) vnode. This process is unreliable but works just fine for the most common cases. The unreliability is because it relies on VFS cache (it doesn't traverse the on medium structures), it doesn't work with hardlinks etc. This routine is used in several places in the Linuxulator.

Vnode operations

fgetvp - given a thread and a file descriptor number it returns associated vnode
vn_lock - locks a vnode
vn_unlock - unlocks a vnode
VOP_READDIR - reads a directory referenced by a vnode
VOP_GETATTR - gets attributes of a file or a directory referenced by a vnode
VOP_LOOKUP - looks up a path to a given directory
VOP_OPEN - opens a file referenced by a vnode
VOP_CLOSE - closes a file referenced by a vnode
vput - decrement the use count for a vnode and unlock it
vrele - decrement the use count for a vnode
vref - increment the use count for a vnode

File handler operations

fget - given a thread and a file descriptor number it returns associated file handler and references it
fdrop - drops a reference to a file handler
fhold - references a file handler

Kapitola 4

Linux emulation layer - MD part

This chapter deals with implementation of Linux emulation layer in FreeBSD operating system. It first describes the machine dependant part talking about how and where interaction between userland and kernelland is implemented. It talks about syscalls, signals, ptrace, traps, stack fixup. This part covers only i386 but is written generally so no other architecture should differ much. The next part is the machine independant part of the Linuxulator. This part covers i386 and ELF handling only. A.out is obsolete and untested.

4.1 Syscall handling

Syscall handling is mostly written in *linux_sysvec.c* which covers most of the routines pointed out in the *sysentvec* structure. When a Linux process running on FreeBSD issues a syscall, the general syscall routine calls *linux_prepsyscall* routine for the Linux ABI.

4.1.1 Linux_prepsyscall

Linux passes arguments to syscalls via registers (that's why it's limited to 6 parameters on i386) while FreeBSD uses the stack. The *Linux_prepsyscall* routine must copy parameters from registers to the stack. The order of the registers is: %ebx, %ecx, %edx, %esi, %edi, %ebp. The catch is that this is true for only MOST of the syscalls. Some (most notably clone) uses a different order but it's luckily easy to fix by inserting a dummy parameter in the linux_clone prototype.

4.1.2 Syscall writing

Every syscall implemented in the Linuxulator must have its prototype with various flags in the *syscalls.master* file. The form of the file is:

```
...
2       AUE_FORK        STD     { int linux_fork(void); }
...
6       AUE_CLOSE       NOPROTO { int close(int fd); }
...
```

The first column represents the syscall number. The second column is for auditing support. The third column represents the syscall type. It's either STD, OBSOL, NOPROTO and UNIMPL. STD is a standard syscall with full prototype and implementation. OBSOL is obsolete and defines just the prototype. NOPROTO means that the syscall is implemented elsewhere so dont prepend ABI prefix etc. UNIMPL means the syscall will be substituted with the *nosys* syscall (a syscall just printing out a message about the syscall not being implemented and returning *ENOSYS*).

From the *syscalls.master* file a script generates three files: *linux_syscall.h*, *linux_proto.h* and *linux_sysent.c*. The *linux_syscall.h* contains defines with syscall names and their numerical value, eg.:

```
...
#define LINUX_SYS_linux_fork    2
...
#define LINUX_SYS_close 6
...
```

The *linux_proto.h* contains structure definitions of arguments to every syscall, eg.:

```
struct linux_fork_args {
        register_t dummy;
};
```

And finally the *linux_sysent.c* contains structure describing the system entry table, used in actual dispatching a syscall, eg:

```
{ 0, (sy_call_t *)linux_fork, AUE_FORK, NULL, 0, 0 },           /* 2 = linux_fork */
{ AS(close_args), (sy_call_t *)close, AUE_CLOSE, NULL, 0, 0 },  /* 6 = close */
```

As you can see *linux_fork* is implemented in Linuxulator itself so the definition is of STD type and has no argument which is exhibited by the dummy argument structure. On the other hand *close* is just an alias for real FreeBSD close so it has no linux arguments structure associated and in the system entry table it's not prefixed with linux as it calls the real close function in the kernel.

4.1.3 Dummy syscalls

The Linux emulation layer is not complete, as some syscalls are not implemented properly and some are not implemented at all. The emulation layer employs a facility to mark unimplemented syscalls with the *DUMMY* macro. These dummy definitions reside in *linux_dummy.c* file in a form of *DUMMY(syscall);* which is then translated to various syscall auxiliary files and the implementation consists of printing a message saying that this syscall is not implemented. The UNIMPL prototype is not used because we want to be able to identify the name of the syscall that was called in order to know what syscalls are more important to implement.

4.2 Signal handling

Signal handling is done generally in the FreeBSD kernel for all binary compatibilities with a call to a compat-dependant layer. Linux compatibility layer defines *linux_sendsig* routine for this purpose.

4.2.1 Linux_sendsig

This routine first checks whether the signal has been installed with a *SA_SIGINFO* in which case it calls *linux_rt_sendsig* routine instead. Further it allocates (or reuses an already existing) signal handle context, then it builds a list of arguments for the signal handler. It translates the signal number based on the signals translation table, assigns a handler, translates sigset. Then it saves context for the *sigreturn* routine (various registers, translated trap number and signal mask). Finally it copies out the signal context to the userspace and prepares context for the actual signal handler to run.

4.2.2 Linux_rt_sendsig

This routine is similar to *Linux_sendsig* just the signal context preparation is different. It adds *siginfo*, *ucontext*, and some POSIX parts. It might be worth considering whether those two functions couldn't be merged benefiting in less code duplication and possibly even faster execution.

4.2.3 Linux_sigreturn

This syscall is used for return from the signal handler. It does some security checks and restores original process context. It also unmasks the signal in process signal mask.

4.3 Ptrace

Many Unix derivates implement the *ptrace* syscall in order to allow various tracking and debugging features. This facility enables the tracing process to obtain various information about the traced process, like register dumps, any memory from the process address space etc. and also to trace the process like in stepping an instruction or between system entries (syscalls and traps). Ptrace also lets you set various information in the traced process (registers etc.). Ptrace is a Unix wide standard implemented in most Unixes around the world.

Linux emulation in FreeBSD implements ptrace facility in *linux_ptrace.c.* The routines for converting registers between Linux and FreeBSD and the actual *ptrace* syscall emulation syscall. The syscall is a long switch block that for every ptrace command implements its counterpart in FreeBSD. Ptrace commands are mostly equal between Linux and FreeBSD so usually just a small modification is needed. For example *PT_GETREGS* in Linux operates on direct data while FreeBSD uses pointer to the data so after a (native) ptrace syscall was performed a copyout must be done to preserve Linux semantics.

Ptrace implementation in Linuxulator has some known weaknesses. There have been panics seen when using strace (which is a ptrace consumer) in the linuxulator environment. Also *PT_SYSCALL* is not implemented.

4.4 Traps

Whenever a Linux process running in the emulation layer traps the trap itself is handled transparently with the only exception of the trap translation. Linux and FreeBSD differs in opinion on what a trap is so this is dealt with here. The code is actually very short:

```
static int
translate_traps(int signal, int trap_code)
{
        if (signal != SIGBUS)
                return signal;
        switch (trap_code) {
        case T_PROTFLT:
        case T_TSSFLT:
        case T_DOUBLEFLT:
        case T_PAGEFLT:
                return SIGSEGV;
        default:
                return signal;
        }
}
```

4.5 Stack fixup

The RTLD run-time link-editor expects so called AUX tags on stack during an *execve* so a fixup must be done to ensure this. Of course every RTLD system is different so the emulation layer must provide its own stack fixup routine to do this. So does Linuxulator. The *elf_linux_fixup()* simply copies out AUX tags to the stack and adjusts the stack of the user space process to point right after those tags. So RTLD happily works.

4.6 A.OUT support

The Linux emulation layer on i386 also supports Linux A.OUT binaries. Pretty much everything described in the previous sections must be implemented for A.OUT support (beside traps translation and signals sending). The support for A.OUT binaries is no longer maintained, especially the 2.6 emulation doesn't work with it but this doesn't cause any problem, as the Linux base in ports probably don't support A.OUT binaries at all. This support will probably be removed in future. Most of the stuff necessary for loading Linux A.OUT binaries is in *imgact_linux.c* file.

Kapitola 5

Linux emulation layer - MI part

This chapter talks about machine independent part of the Linuxulator. It covers the emulation infrastructure needed for Linux 2.6 emulation, the thread local storage (TLS) implementation (on i386) and futexes. Then we talk briefly about some syscalls.

5.1 Description of NTPL

One of the major areas of progress in development of Linux 2.6 was threading. Prior to 2.6, the Linux threading support was implemented in the *linuxthreads* library. The library was a partial implementation of POSIX threading. The threading was implemented using separate processes for each thread using the *clone* syscall to let them share the address space (and other things). The main weaknesses of this approach was that every thread had a different *PID*, signal handling was broken (from the pthreads perspective) etc. Also the performance was not very good (use of SIGUSR signals for threads synchronization, kernel resources consumption etc.) so to overcome these problems a new threading system was developed and named NPTL.

The NPTL library focused on two things but a third thing came along so it's usually considered a part of the NTPL. Those two things were embedding of threads into a process structure and futexes. The additional third thing was TLS which is not directly required by NPTL but the whole NPTL userland library depends on it. Those improvements yielded in much improved performance and standards conformance. NPTL is a standard threading library in Linux systems these days.

The FreeBSD Linuxulator implementation approaches the NPTL in three main areas. The TLS, futexes and PID mangling which is meant to simulate the Linux threads. Further sections describe each of these areas.

5.2 Linux 2.6 emulation infrastructure

These sections deals with the way Linux threads are managed and how we simulate that in FreeBSD.

5.2.1 Runtime determining of 2.6 emulation

The Linux emulation layer in FreeBSD supports runtime setting of the emulated version. This is done via a sysctl, namely *compat.linux.osrelease* which is set to 2.4.2 by default (as of April 2007) and with all Linux versions upto 2.6 it just determined what *uname* outputs. It is different with 2.6 emulation where setting this sysctl affects runtime behaviour of the emulation layer. When set to 2.6.x it sets the value of *linux_use_linux26* while setting to something else keeps it unset. This variable (plus per-prison variables of the very same kind) determines wheter 2.6 infrastructure (mainly PID mangling) is used in the code or not. The version setting is done system-wide and this affects all Linux processes. The sysctl should not be changed when running any Linux binary as it might harm things.

5.2.2 Linux process and thread identifiers

The semantics of Linux threading are a little confusing and uses entirely different nomenclature to FreeBSD. A process in Linux consists of a *struct task* embedding two identifier fields - PID and TGID. PID is NOT a process ID but it's a thread ID. The tgid identifies a thread group in other words a process. For single-threaded process the PID equals the TGID.

The thread in NPTL is just an ordinary process that happens to have tgid not equal to PID and have a group_leader not equal to itself (+shared VM etc. of course). Everything else is just like an ordinary process! There is no separation of a shared status to some external structure like in FreeBSD. This creates some duplication of information and possible data inconsistency. The Linux kernel seems to use task->group information in some places and task information elsewhere and it's really not very consistent and looks error-prone.

Every NTPL thread is created by a call to the *clone* syscall with a specific set of flags (more in the next subsection). The NPTL implements strict 1:1 threading.

In FreeBSD we emulate NPTL threads with ordinary FreeBSD processes that share VM space etc. and the PID gymnastic is just mimiced in the linux-emulation specific structure attached to the process. The structure attached to the process looks like:

```
struct linux_emuldata {
        pid_t   pid;

        int    *child_set_tid;  /* in clone(): Child's TID to set on clone */
        int    *child_clear_tid;/* in clone(): Child's TID to clear on exit */

        struct linux_emuldata_shared *shared;

        int     pdeath_signal;          /* parent death signal */

        LIST_ENTRY(linux_emuldata) threads;     /* list of linux threads */
};
```

The PID is used to identify the FreeBSD process that attaches this structure. The child_set—clear_tid is used for TID addresses copyout at process exit and creation. The shared pointer points to a structure shared among threads. The pdeath_signal identifies parent death signal. And the threads pointer is used to link this structure to the list of threads. The shared structure looks like:

```
struct linux_emuldata_shared {
        int     refs;
        pid_t   group_pid;
        LIST_HEAD(, linux_emuldata) threads; /* head of list of linux threads */
};
```

The refs is a reference counter being used to determine when we can free the structure to avoid memory leaks. The group_pid is to identify PID (=TGID) of the whole process (=thread group). And the threads pointer is the head of the list of threads in the process.

The linux_emuldata structure can be obtained from the process using *em_find* function. The prototype of the function is:

```
struct linux_emuldata *em_find(struct proc *, int locked);
```

where the proc is the process we want the emuldata structure from and the locked parameter determines whether we want to lock or not. The accepted values are *EMUL_DOLOCK* and *EMUL_DOUNLOCK*. More about locking later.

5.2.3 PID mangling

Because of the described different view know what a process ID and thread ID is between FreeBSD and Linux we have to translate the view somehow. We do it by *PID mangling*. This means that we fake what a PID (=TGID) and TID (=PID) is between kernel and userland. The rule of thumb is that in kernel (in Linuxulator) PID = PID and TGID = shared->group_pid and to userland we present PID = shared-¿group_pid and TID = proc->p_pid. The PID member of linux_emuldata structure is a FreeBSD PID.

The above affects mainly *getpid, getppid, gettid* syscalls. Where we use PID/TGID respectively. In copyout of TIDs in child_clear—set_tid we copy out FreeBSD PID.

5.2.4 Clone syscall

The clone syscall is the way threads are created in Linux. The syscall prototype looks like this:

```
int linux_clone(l_int flags, void *stack, void *parent_tidptr, int dummy,
                   void * child_tidptr);
```

The flags parameter tells the syscall how exactly the processes should be cloned. As described above Linux can create process sharing various things independently, for example two processes can share file descriptors but not VM etc. Last byte of the flags parameter is exit signal of the newly created process. The stack parameter if non-NULL tells where the thread stack is and if it is NULL we are supposed to copy-on-write the calling process stack (ie. do what normal fork() routine does). The parent_tidptr parameter is used as an address for copying out process PID (ie. thread id) once the process is sufficiently instantiated but is not runnable yet. The dummy parameter is here because of very strange calling convention of this syscall on i386. It uses directly registers and not letting the compiler do it which results in the need of a dummy syscall. The child_tidptr parameter is used as an address for copying out PID once the process has finished forking and when the process exits.

The syscall itself proceeds by setting corresponding flags depending on the flags passed in. For example CLONE_VM maps to RFMEM (sharing of VM) etc. The only nit here is CLONE_FS/CLONE_FILES because FreeBSD doesn't allow setting this separately so we fake it by not setting RFFDG (copying of fd table and other fs information) if either of these is defined. This doesn't cause any problems, because those flags are always set together. After setting flags the process is forked using internal *fork1()* routine, the process is instrumented to not be put on a run queue, ie. not be set runnable. After the forking is done we possibly reparent the newly created process to emulate CLONE_PARENT semantic. Next part is creating emulation data. Threads in Linux does not signal its parent so we set exit_signal to 0 to disable this. After that setting of child_set_tid and child_clear_tid is performed enabling the functionality later in the code. At this point we copy out the PID to the address specified by parent_tidptr pointer. The setting of process stack is done by simply rewriting thread frame *%esp* register (%rsp on amd64). Next part is setting up TLS for the newly created process. After this vfork semantic might be emulated and finally the newly created process is put on a run queue and copying out its PID to the parent process via clone() return value is done.

The clone syscall is able and in fact used for emulating classic *fork()* and *vfork()* syscalls. Newer glibc in a case of 2.6 kernel uses clone() to implement fork and vfork syscalls.

5.2.5 Locking

The locking is implemented to be per-subsystem because we don't expect a lot of contention on these. There are two locks: emul_lock used to protect manipulating of linux_emuldata and emul_shared_lock used to manipulate the linux_emuldata_shared. The emul_lock is a nonsleepable blocking mutex while the emul_shared_lock is a sleepable blocking sx lock. Because of the per-subsystem locking we can coalesce some locks and that's why the em_find offers the non-locking access.

5.3 TLS

This section deals with TLS aka thread local storage.

5.3.1 Introduction to threading

Threads in computer science are entities within a process that can be scheduled independently from each other. Threads in the process share process wide data (file descriptors etc.) but also have its own stack for their own data. Sometimes there is a need for process wide data specific to a given thread. Imagine a name of the thread in execution or something like that. The traditional Unix threading API, *PTHREADS* provides a way to do it via *pthread_key_create*, *pthread_setspecific* and *pthread_getspecific* where a thread can create key to the thread local data and using setspecific/getspecific manipulate those data. You can easily see that this is not the most comfortable way this could be accomplished. So various producers of C/C++ compilers introduced a better way. They defined a new modifier keyword *__thread* that specifies that a variable is thread specific. A new method of accessing such variables was developed as well (at least on i386). The PTHREADS method tends to be implemented in userspace as a trivial lookup table. The performance of such a solution is not very good. So the new method uses (on i386) segment registers to address a segment where TLS area is stored so the actual accessing a __thread variable is just appending the segment register to the address thus addressing via it. The segment registers are usually *gs* and *fs* acting like segment selectors. Every thread has its own area where the thread local data are stored and the segment must be loaded on every context switch. This method is very fast and used almost exclusively in all i386 Unix world. Both FreeBSD and Linux implement this approach and it yields very good results. The only drawback is the need to reload the segment on every context switch which can slowdown context switches. FreeBSD tries to avoid this overhead by using only 1 segment descriptor for this while Linux uses 3. Interesting thing is that almost nothing uses more than 1 descriptor (only WINE seems to use 2) so Linux pays this unnecessary prize for context switches.

5.3.2 Segments on i386

The i386 architecture implements so called segments. A segment is a description of an area of memory. The base address (bottom) of the memory area, the end of it (ceiling), type, protection etc. The memory described by a segment can be accessed using segment selector registers (CS, DS, SS, ES, FS, GS). For example lets suppose we have a segment which base address is 0x1234 and length and this code:

```
mov     %edx,%gs:0x10
```

This will load the content of the *edx* register into memory location 0x1244. Some segment registers have special use like CS being used for code segment and SS for stack segment but FS and GS are generally unused. Segments are either stored in a global GDT table or in a local LDT table. LDT is accessed via an entry in the GDT. The LDT can store more types of segments. LDT can be per process. Both tables define upto 8191 entries.

5.3.3 Implementation on Linux i386

There are two main ways of setting up TLS in Linux. It can be set when cloning a process using the clone syscall or it can call *set_thread_area*. When a process passes *CLONE_SETTLS* flag to clone(), the kernel expects the memory pointed to by the esi register a linux user space representation of a segment which gets translated to the machine representation of a segment and loaded into a GDT slot. The GDT slot can be specified with a number or -1 can be used meaning that system itself should chose the first free slot. In practice, the vast majority of programs use only 1 TLS entry and doesn't care about the number of the entry. We exploit this in the emulation and in fact depend on it.

5.3.4 Emulation of Linux TLS

i386

Loading of TLS for the current thread happens by calling *set_thread_area* while loading TLS for a second process in clone is done in the separate block in clone. Those two functions are very similar. The only difference being the actual loading of the GDT segment which happens on the next context switch for the newly created process while *set_thread_area* must load this directly. The code basically does this. It copies linux form segment descriptor from userland. The code checks for the number of the descriptor but because this differs between FreeBSD and Linux we fake it a little. We only support indexes of 6, 3 and -1. The 6 is genuine Linux number, 3 is genuine FreeBSD one and -1 means autoselection. Then we set the descriptor number to constant 3 and copy out this to the userspace. We rely on the userspace process using the number from the descriptor but this works most of the time (have never seen a case where this didn't work) as the userspace process typically passes in -1. Then we convert the descriptor from linux form to machine dependant form (ie. operating system independent form) and copy this to the FreeBSD defined segment descriptor. Finally we can load it. We assign the descriptor to threads PCB (process control block) and load the gs segment using *load_gs*. This loading must be done in critical section so nothing can interrupt us. The *CLONE_SETTLS* case works exactly like this just the loading using *load_gs* is not performed. The segment used for this (segment number 3) is shared for this use between FreeBSD processes and Linux processes so the Linux emulation layer doesn't add any overhead over plain FreeBSD.

amd64

The amd64 implementation is similar to i386 one but there was initially no 32bit segment descriptor used for this purpose (hence not even native 32bit TLS users worked) so we had to add such a segment and implement its loading on every context switch (when a flag signaling use of 32bit is set). Otherwise the TLS loading is exactly the same just the segment numbers are different and the descriptor format and loading slightly differs.

5.4 Futexes

5.4.1 Introduction to synchronization

Threads need some kind of synchronization and POSIX provides some of them. Mutexes for mutual exclusion, read-write locks for mutual exclusion with biased ratio of reads and writes and condition variables for signaling a status change. It is interesting to note that POSIX threading API lacks support for semaphores. Those synchronization routines implementations are heavily dependant on the type threading support we have. In pure 1:M (userspace) model the implementation can be solely done in userspace and thus be very fast (the condition variables will probably end up being implemented using signals, ie. not fast) and simple. In 1:1 model, the situation is also quite clear - the threads must be synchronized using kernel facilites (which is very slow because a syscall must be performed). The mixed M:N scenario just combines the first and second approach or rely solely on kernel.

Threads synchronization is a vital part of thread-enabled programming and its performance can affect resulting program a lot. Recent benchmarks on FreeBSD operating system showed that an improved sx lock implementation yielded 40% speedup in ZFS (a heavy sx user), this is in-kernel stuff but it shows clearly how important the performance of synchronization primitives is.

Threaded programs should be written with as little contention on locks as possible. Otherwise instead of doing useful work the thread just waits on a lock. Because of this, most well written threaded programs show little locks contention.

5.4.2 Futexes introduction

Linux implements 1:1 threading, ie. it has to use in-kernel synchronization primitives. As stated earlier, well written threaded programs have little lock contention. So a typical sequence:

```
pthread_mutex_lock(&mutex);
....
pthread_mutex_unlock(&mutex);
```

could be performed as two atomic increase/decrease mutex reference counter. Which is very fast. Instead 1:1 threading forces us to perform two syscalls for those mutex calls. Which is very slow.

The solution Linux 2.6 implements is called *Futexes*. Futexes implement the check for contention in userspace and call kernel primitives only in a case of contention. Thus the typical case is without any kernel intervention. This yields reasonably fast and flexible synchronization primitives implementation.

5.4.3 Futex API

The futex syscall looks like this:

```
int futex(void *uaddr, int op, int val, struct timespec *timeout, void *uaddr2,
int val3);
```

where uaddr is an address of the mutex in userspace, the op is an operation we are about to perform and the other parameters have per-operation meaning.

The futexes implement those operations:

1. FUTEX_WAIT
2. FUTEX_WAKE
3. FUTEX_FD
4. FUTEX_REQUEUE
5. FUTEX_CMP_REQUEUE
6. FUTEX_WAKE_OP

FUTEX_WAIT

This operation verifies that on address *uaddr* the value *val* is written. If not *EWOULDBLOCK* is returned, otherwise the thread is queued on the futex and it is suspended. If the argument *timeout* is non-zero it specifies the maximum time for the sleeping, otherwise the sleeping is infinite.

FUTEX_WAKE

This operation takes a futex at *uaddr* and wakes up *val* first futexes queued on this futex.

FUTEX_FD

This operations associates a file descriptor with a given futex.

FUTEX_REQUEUE

This operation takes *val* threads queued on futex at *uaddr* and wakes them up, and takes *val2* next threads and requeues them on futex at *uaddr2*

FUTEX_CMP_REQUEUE

This operation does the same as *FUTEX_REQUEUE* but it checks that *val3* equals to *val* first.

FUTEX_WAKE_OP

This operation performs an atomic operation on *val3* (which contains coded some other value) and *uaddr*. Then it wakes up *val* threads on futex at *uaddr* and if the atomic operation returned positive number it wakes up *val2* threads on futex at *uaddr2*

The operations implemented in the FUTEX_WAKE_OP

1. FUTEX_OP_SET
2. FUTEX_OP_ADD
3. FUTEX_OP_OR
4. FUTEX_OP_ANDN
5. FUTEX_OP_XOR

Note that there is no *val2* parameter in the futex prototype. The *val2* is taken from the *struct timespec *timeout* parameter for operations FUTEX_REQUEUE, FUTEX_CMP_REQUEUE and FUTEX_WAKE_OP.

5.4.4 Futex emulation in FreeBSD

The futex emulation in FreeBSD is taken from NetBSD and further extended by us. It is placed in *linux_futex.c* and *linux_futex.h* files. The *futex* structure looks like:

```
struct futex {
        void   *f_uaddr;
        int     f_refcount;
        LIST_ENTRY(futex) f_list;
        TAILQ_HEAD(lf_waiting_proc, waiting_proc) f_waiting_proc;
};
```

and the structure *waiting_proc* is:

```
truct waiting_proc {
        struct thread *wp_t;
        struct futex *wp_new_futex;
        TAILQ_ENTRY(waiting_proc) wp_list;
};
```

futex get/put

A futex is obtained using the *futex_get* function that searches a linear list of futexes and returns the found one or creates a new futex. When releasing a futex from use we call the *futex_put* function which decreases a reference counter of the futex and if the refcount reaches zero it's released.

futex sleep

When a futex queues a thread for sleeping it creates a *working_proc* structure and puts this structure to the list inside the futex structure then it just performs a *tsleep* to suspend the thread. The sleep can be timed out. After the tsleep returns (the thread was woken up or it timed out) the *working_proc* structure is removed from the list and destroyed. All this is done in the *futex_sleep* function. If we got woken up from *futex_wake* we have wp-¿wp_new_futex set so we sleep on it. This way the actual requeueing is done in this function.

futex wake

Waking up a thread sleeping on a futex is performed in the *futex_wake* function. In this function we first mimic the strange Linux behaviour where it wakes up N threads for all operations, the only exception is that the REQUEUE operations are performed on N+1 threads. But this usually doesn't make any difference as we are waking up all threads. Next in the function in the loop we wake up *n* threads, after this we check if there is a new futex for requeueing. If so we requeue up to *n2* threads on the new futex. This cooperates with *futex_sleep*.

futex wake op

The WAKE_OP operation is quite complicated. First we obtain two futexes at addresses *uaddr* and *uaddr2* then we perform the atomic operation using *val3* and *uaddr2*. Then *val* waiters on the first futex is woken up and if the atomic operation condition holds we wake up *val2* (ie. *timeout*) waiter on the second futex.

futex atomic operation

The atomic operation takes two parameters *encoded_op* and *uaddr*. The encoded operation encodes the operation itself, comparing value, operation argument, and comparing argument. The pseudocode for the operation is like

```
oldval = *uaddr2
*uaddr2 = oldval OP oparg
```

And this is done atomically. First a copying in of the number at *uaddr* is performed and the operation is done. The code handles page faults and if no page fault occurs the *oldval* is compared to *cmparg* argument with *cmp* comparator.

Futex locking

Futex implementation uses two locks list protecting sx lock and global lock (either Giant or another sx lock). Every operation is performed locked from the start to the very end.

Futex problems

Currently the futex implementation in FreeBSD suffers some problems. First it is locked by Giant and second we don't pass futex testing program. It looks like the locking under contention is wrong. Work is ongoing to fix this.

5.5 Various syscalls implementation

In this section I am going to describe some smaller syscalls that are worth mentioning because their implementation is not obvious or those syscalls are interesting from other point of view.

5.5.1 *at family of syscalls

During development of Linux 2.6.16 kernel, the *at syscalls were added. Those syscalls (openat for example) work exactly like their at-less counterparts with the slight exception of the *dirfd* parameter. This parameter changes where the given file, on which the syscall is to be performed, is. When filename parameter is absolute the *dirfd* is ignored but when the path to the file is relative, the dirfd comes to play. The *dirfd* is a directory relative to which the relative pathname is checked. The dirfd is a file descriptor of some directory or *AT_FDCWD*. So for example the openat() syscall can be like this:

```
file descriptor 123 = /tmp/foo/, current working directory = /tmp/

openat(123, „/tmp/bah\, flags, mode)    ---> opens /tmp/bah
openat(123, „bah\, flags, mode)         ---> opens /tmp/foo/bah
openat(AT_FDWCWD, „bah\, flags, mode)   ---> opens /tmp/bah
openat(stdio, „bah\, flags, mode)       ---> returns error because stdio is
                                               not a directory
```

This infrastructure is necessary to avoid races when opening files outside CWD. Imagine that a process consists of two threads, thread A and thread B. Thread A issues „open("/tmp/foo/bah„, flags, mode)" and before return it gets preempted and thread B runs. Thread B doesn't care about threads A needs and renames or removes „/tmp/foo/". We got a race. To avoid this we can open „/tmp/foo" and use it as a *dirfd* for openat syscall. This also enables user to implement per-thread CWD.

Linux family of *at syscalls contains: *linux_openat, linux_mkdirat, linux_mknodat, linux_fchownat, linux_futimesat, linux_fstatat64, linux_unlinkat, linux_renameat, linux_linkat, linux_symlinkat, linux_readlinkat, linux_fchmodat* and *linux_faccessat.* All these are implemented using the modified namei routine and simple wrapping layer.

Implementation

The implementation is done by altering the namei routine (described above) to take additional parameter *dirfd* in its nameidata structure which specifies the starting point of the pathname lookup instead of using CWD every time. The dirfd resolution from file descriptor number to a vnode is done in native *at syscalls. When dirfd is AT_FDCWD the dvp entry in nameidata structure is null but when dirfd is a different number we obtain a file for this file descriptor, check whether this file is valid and if there's vnode attached to it then we get a vnode. Then we check this vnode for being a directory. In the actual namei routine we simply substitute the dvp vnode for dp variable in the namei function which determines the starting point. The namei is not used directly but via a trace of different functions on various levels. For example the openat() goes like this:

```
openat() --> kern_openat() --> vn_open() -> namei()
```

So the *kern_open* and *vn_open* must be altered to incorporate the additional dirfd parameter. No compat layer is created for those because there are not many users of this and the users can be easily converted. This general implementation enables FreeBSD to implement their own *at syscall. This is being discussed right now.

5.5.2 Ioctl

The ioctl interface is quite fragile because of its generality. We have to bear in mind that devices differ between Linux and FreeBSD so some care must be applied to do ioctl emulation work right. The ioctl handling is implemented in linux_ioctl.c where linux_ioctl() function is defined. This function simply iterates over sets of ioctl handlers to find a handler that implements a given command. The ioctl syscall has three parameters - file descriptor, command and an argument. The command is a 16bit number which in theory is divided into high 8bits determining class of the ioctl command and low 8bits which are the actual command within the given set. The emulation takes advantage of this division. We implement handlers for every set, like sound_handler or disk_handler. Every handler has maximum command and minimum command defined which is used for determining what handler is used. There are slight problems with this approach because Linux does not use the set division consistently so sometimes ioctls for a different set are inside a set they should not belong to (scsi generic ioctls inside cdrom set etc.). FreeBSD currently does not implement many Linux ioctls (compared to NetBSD for example) but the plan is to port those from NetBSD. The trend is to use Linux ioctls even in the native FreeBSD drivers because of eased porting of applications.

5.5.3 Debugging

Every syscall should be debuggable. For this purpose we introduce a small infrastructure. We have *ldebug* facility which tells whether a given syscall should be debugged (settable via a sysctl). For printing we have *LMSG* and *ARGS* macros. Those are used for altering a printable string for uniform debuging messages.

Kapitola 6

Conclusion

6.1 Results

As of April 2007 the Linux emulation layer is capable of emulating Linux 2.6.16 kernel quite well. The remaining problems concerns futexes, unfinished *at family of syscalls, problematic signals delivery, missing epoll/inotify and probably some bugs we haven't discovered yet. Despite this we are capable of running basically all the Linux programs included in FreeBSD ports collection with Fedora Core 4 at 2.6.16 and there are some rudimentary reports of success with Fedora Core 6 at 2.6.16. The Fedora Core 6 Linux base was recently commited enabling some further testing of the emulation layer and giving us some more hints where we should put our effort in implementing missing stuff.

We are able to run the most used applications like linux-firefox, linux-opera, skype and some games from the ports collection. Some of the programs exhibit bad behaviour under 2.6 emulation but this is currently under investigation and hopefully will be fixed soon. The only big application that is known not to work is the linux Java development kit and this is because of the requirement of epoll facility which is not directly related to Linux kernel 2.6.

We hope to enable 2.6.16 emulation by default some time after FreeBSD 7.0 is released at least to expose the 2.6 emulation parts to some wider testing. Once this is done we can switch to Fedora Core 6 linux base which is the ultimate plan.

6.2 Future work

Future work should focus on fixing the remaining issues with futexes, implement the rest of the *at family of syscalls, fix the signals delivery and possibly implement the epoll/inotify facilities. I hope to be able to work on this during the summer of 2007 in another run of Google Summer of Code.

We hope to be able to run the most important programs flawlessly soon so we will be able to switch to 2.6 emulation on default and make the Fedora Core 6 the default linux base because our currently used Fedora Core 4 is not supported anymore.

The other possible goal is to share our code with NetBSD and DragonflyBSD. NetBSD has some support for 2.6 emulation but its far from finished and not really tested. DragonflyBSD has expressed some interest in porting the 2.6 improvements.

Generally as Linux develops we would like to keep up with their development, implementing newly added syscalls. Splice comes to mind first. Also some already implemented syscalls are heavily crippled, for example mremap and others.

Some performance improvements can also be made, finer grained locking and others.

6.3 Team

I cooperated on this project with (in alphabetical order): John Baldwin, Konstantin Belousov, Emmanuel Dreyfus, Scot Hetzel, Jung-uk Kim, Alexander Leidinger, Boris Samorodov, Suleiman Souhlal, Li Xiao and David Xu. I would like to thank all those people for their advices, code reviews and general support.

Dodatek A

Google Summer of Code

Google Inc., organizes a students program offering some organizations sponsoring students to work on the organizations projects. The program lasts for 3 months in the summer hence it's called Summer of Code. In summer 2006 I applied for this program and was selected.

Application

When a student decides he/she wants to apply for this program and the project is selected one has to write a proposal which will be judged at the organization and Google. The proposal consists partly of a CV and partly technical description of the project possibly including a roadmap. My proposal is here:

```
Google's summer of code proposal

Personal
Name: Roman Divacky
Address:                Boreticka 9, Brno - Vinohrady, Czech republic
Birth:                  May 25th, 1983
phone:                  +420 736 130 392

email: xdivac02@stud.fit.vutbr.cz

Project

Title: Update the Linuxolator
Synopsis: Update the FreeBSD Linux emulation layer to the current Linux kernel as of
2.6.x
Benefits: Ability to run modern Linux programs under this emulation

Details: Currently FreeBSD Linuxolator implements kernel 2.4.2 api (with some syscalls
missing) but Linux world has moved forward and current Linux kernel has many
more syscalls. FreeBSD lacks this so it's unable to run some modern Linux
compiled programs which use those syscalls or doesn't run optimally.

Technical plan details:
```

1) examine the current Linuxolator code, review it fix possible bugs and do some cleanups
2) look at the code and make detail plan on what syscalls are missing, what syscalls have different semantics (sendfile comes to mind)
3) split the syscalls into groups and make plan on what to implement first and what later - this should be coordinated with mentor
4) work on the syscalls by given priorities, all syscalls should be added at least as unimplemented

Preliminary plan on syscalls, checked on i386 (which is the only platform Linuxolator supports): unimplemented syscalls, I list only those which are not obsoleted/notimplemented in Linux:
readahead
sendfile64
futex
sched_[sg]etaffinity
get_thread_area
io_*
lookup_cookie
epoll_*
remap_file_pages
set_tid_address
timer_*
clock_*
**** those syscalls are included in the current Linuxolator but not implemented ****
syscalls added up to 2.6.16:
the last syscall implemented by Linuxolator in FreeBSD is 267, Linux kernel 2.6.16 lists syscall 310 as the last syscall. This gives us 69 unimplemented syscalls.

I would put more priority on the first group of syscalls as they are older so presumably used by more applications. Especially mapping epoll to kqueue would be useful. From the latter group it must be carefully chosen what to implement (given time constraints). Some syscalls cannot be implemented (or implemented properly) due to divergence between the kernels (get_mempolicy etc.).

Deliverables:
o updated Linuxolator with new syscalls implemented
o review of the Linuxolator code
o documentation of the code, comments in the code and man page update/extension
o set of regression tests

Project schedule

Start: early June (immediately after my semester ends)
Estimated time: 3 months for the most intensive work and I am willing and able to work on this after SoC ends, I don't have other duties during summer so I can work full-time on it

```
Bio

I am a student at the Brno University of Technology, studying 4th year, aged 23. I have
earned a BS degree here. I am a contributor to the FreeBSD project, my main area of int
is various optimization tweaks. I removed COMPAT_43 related code from Linuxolator
which removed obsolete and unmaintained code and improved performance. I personally
have access to i386/amd64 machines so I can test all the work I will do. During the
work on COMPAT_43 I got some skills with regard to the Linuxolator. Hence I think
I am suitable person for the task.
```

In every organization each project is assigned a possible mentor and then the project evaluates the proposals sent in. The resulting priority list is sent to Google which decides how many sponsorships every organization will get. Last minute changes are allowed. After the final assignment is done, students are informed whether they were accepted or rejected. What happens next depends on every project.

In the case of FreeBSD I obtained a Perforce account, made my Summer of Code branch there and started working. Students are supposed to give formal status reports to their mentors, but I communicated daily with mine so this was not necessary. Two evaluations of students work are done. Midterm and final. To get positive final evaluation the work done in the Summer of Code project doesn't have to be officially integrated into the originating project.

The final payment can be done either via wire transfer or paper check. I chose the paper check because we were told its faster. Its not. Its much slower and expensive.

Overall I found the Summer of Code a great experience. We were even offered a chance for an internship at Google.

Dodatek B

Code metrics

The main patch implemented during the Summer of Code program and then committed to the FreeBSD source repository was 4688 lines long and of size 221774 bytes. Since then many other patches have been committed some of them submitted by me and some by others. The ratio is roughly that I made 50% of the patch. I spent 3 months studying and implementing the Linux 2.6 emulation layer.

Literatura

[1] Marshall Kirk McKusick. George V. Nevile-Neil. *Design and Implementation of the FreeBSD operating system.* Addison-Wesley, 2005.

[2] WWW pages. *FreeBSD operating system.*

[3] WWW pages. *The Linux documentation project.*

[4] WWW pages. *Linux operating system.*