Linux® emulation in FreeBSD

Roman Divacky <rdivacky@FreeBSD.org >
Revision: 43184

Adobe, Acrobat, Acrobat Reader, Flash and PostScript are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

IBM, AIX, OS/2, PowerPC, PS/2, S/390, and ThinkPad are trademarks of International Business Machines Corporation in the United States, other countries, or both.

FreeBSD is a registered trademark of the FreeBSD Foundation.

Linux is a registered trademark of Linus Torvalds.

NetBSD is a registered trademark of the NetBSD Foundation.

RealNetworks, RealPlayer, and RealAudio are the registered trademarks of RealNetworks, Inc.

Oracle is a registered trademark of Oracle Corporation.

Sun, Sun Microsystems, Java, Java Virtual Machine, JDK, JRE, JSP, JVM, Netra, OpenJDK, Solaris, StarOffice, SunOS and VirtualBox are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this document, and the FreeBSD Project was aware of the trademark claim, the designations have been followed by the "™" or the "®" symbol.

2013-11-13 by hrs.

Abstract

This masters thesis deals with updating the Linux® emulation layer (the 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 on making the Linux® 2.6 emulation the default emulation layer in FreeBSD.

Table of Contents

1. Introduction 2
2. A look inside... 2
3. Emulation 12
4. Linux® emulation layer -MD part 18
5. Linux® emulation layer -MI part 22
6. Conclusion 33
7. Literatures 35

1. Introduction

In the last few years the open source UNIX® based operating systems started to be widely deployed on server and client machines. Among these operating systems I would 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 for 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 (the so called Linuxulator) in FreeBSD to include Linux® 2.6 facilities. This thesis is written as a part of this project.

2. A look inside...

In this section we are going to describe every operating system in question. How they deal with syscalls, trapframes etc., all the low-level 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 interprocess communication primitives, copy-on-write, etc. UNIX® itself does not exist any more but its ideas have been used by many other operating systems world wide thus forming the 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. Let us summarize typical UNIX® characteristics.

2.2. 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 the user space. The kernel is where the management of the processes, hardware, and low-level details take place. The kernel provides a standard unified UNIX® API to the user space. The most important ones are covered below.

2.2.1. 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, the syscall number 85 is the swapon(2) syscall and the syscall number 132 is mkfifo(2). Some syscalls need parameters, which are passed from the user-space to the 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 sending a *signal* (division by zero).

2.2.2. 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 processes or by the kernel and received by processes.

Some signals can be ignored or handled by a user supplied routine, some result in a predefined action that cannot be altered or ignored.

2.2.3. Process management

Kernel instances are processed first in the system (so called init). Every running process can create its identical copy using the fork(2) 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 the exec(3) syscall. Some modifications of this syscall were introduced but all serve the same basic purpose. Processes end their lives by calling the exit(2) syscall. Every process is identified by a unique number called PID. Every process has a defined parent (identified by its PID).

2.2.4. Thread management

Traditional UNIX® does not define any API nor implementation for threading, while 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
- - the scheduling cannot be altered by the user (slightly mitigated by the POSIX® API)
- + no syscall wrapping necessary
- + can utilize multiple CPUs

1:N threading

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

2.3. What is FreeBSD?

The FreeBSD project is one of the oldest open source operating systems currently available for daily use. It is a direct descendant of the genuine UNIX® so it could be claimed that it is a true UNIX® although licensing issues do not permit that. 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 will focus on FreeBSD.

FreeBSD is a modern UNIX®-based operating system with all the features of UNIX®. Preemptive multitasking, multiuser facilities, TCP/IP networking, memory protection, symmetric multiprocessing support, virtual memory with merged VM and buffer cache, they are all there. One of the interesting and extremely useful features is the ability to emulate other UNIX®-like operating systems. As of December 2006 and 7-CURRENT development, the following emulation functionalities are supported:

- FreeBSD/i386 emulation on FreeBSD/amd64
- FreeBSD/i386 emulation on FreeBSD/ia64
- Linux®-emulation of Linux® operating system on FreeBSD
- NDIS-emulation of Windows networking drivers interface
- NetBSD-emulation of NetBSD operating system
- PECoff-support for PECoff FreeBSD executables
- SVR4-emulation of System V revision 4 UNIX®

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

FreeBSD development happens in a central CVS repository where only a selected team of so called committers can write. This repository possesses several branches; the most interesting are the HEAD branch, in FreeBSD nomenclature called -CURRENT, and RELENG_X branches, where X stands for a number indicating a major version of FreeBSD. As of December 2006, there are development branches for 6.X development (RELENG_6) and for the 5.X development (RELENG_5). Other branches are closed and not actively maintained or only fed with security patches by the Security Officer of the FreeBSD project.

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

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

2.3.1. Technical details

FreeBSD is traditional flavor of UNIX® in the sense of dividing the run of processes 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 is only one way to return. In the subsequent sections we will describe the three gates to/from the kernel. The whole description applies to the i386 architecture as the Linuxulator only exists there but the concept is similar on other architectures. The information was taken from [1] and the source code.

2.3.1.1. System entries

FreeBSD has an abstraction called an execution class loader, which is a wedge into the execve(2) 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 is used later in the syscall processing code and at some other places. System entries are handled by trap handlers, where we can access both the kernel-space and the user-space at once.

2.3.1.2. 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 arguments (i.e. copies them on to the stack) for a call to a C function syscall(2) (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 32-bit or 64-bit one (changes size of the parameters), then the parameters are copied, including the syscall. Next, the actual syscall function is executed with processing of the return code (special cases for `ERESTART` and `EJUSTRETURN` errors). Finally an `userret()` is scheduled, switching the process back to the users-pace. The parameters to the actual syscall handler are passed in the form of `struct thread *td` , `struct syscall args *` arguments where the second parameter is a pointer to the copied in structure of parameters.

2.3.1.3. Traps

Handling of traps in FreeBSD is similar to the handling of syscalls. Whenever a trap occurs, an assembler handler is called. It is 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()`.

2.3.1.4. Exits

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

2.3.1.5. UNIX® primitives

FreeBSD operating system adheres to the traditional UNIX® scheme, where every process has a unique identification number, the so called *PID* (Process ID). 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(2) 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. Although those two libraries use the same kernel primitives, they are accessed through 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 kernel's 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).

2.4. 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 companies such 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®-like OS: multiprocess-

ing, multiuser support, virtual memory, networking, basically everything is there. There are also highly 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 or changed while D is a minor version for bugfixes only.

More information can be obtained from [4].

2.4.1. Technical details

Linux® follows the 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: via a trap or via a syscall. The return is handled only in one way. The further description applies to Linux® 2.6 on the i386™ architecture. This information was taken from [3].

2.4.1.1. Syscalls

Syscalls in Linux® are performed (in userspace) using `syscallX` macros where `X` substitutes a number representing the 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 values to positive `errno` values and sets `res` to `-1` in case of an error. Whenever the 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).

2.4.1.2. Traps

The trap handlers are introduced in `arch/i386/kernel/traps.c` and most of these handlers live in `arch/i386/kernel/entry.S` , where handling of the traps happens.

2.4.1.3. Exits

Return from the syscall is managed by syscall exit(3), 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 the registers are restored from the stack and the process returns to the userspace.

2.4.1.4. UNIX® primitives

In the 2.6 version, the Linux® operating system 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(2) 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 the `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:

- `CLONE_VM` - processes share their memory space
- `CLONE_FS` - share umask, cwd and namespace
- `CLONE_FILES` - share open files
- `CLONE_SIGHAND` - share signal handlers and blocked signals
- `CLONE_PARENT` - share parent
- `CLONE_THREAD` - be thread (further explanation below)
- `CLONE_NEWNS` - new namespace
- `CLONE_SYSVSEM` - share SysV undo structures
- `CLONE_SETTLS` - setup TLS at supplied address
- `CLONE_PARENT_SETTID` - set TID in the parent

- `CLONE_CHILD_CLEARTID` - clear TID in the child
- `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. The `CLONE_*_*TID` set of flags sets/clears user supplied address to TID or 0.

As you can see the `CLONE_THREAD` does most of the work and does not 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. It is also unclear what this partition is good for as glibc does not 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 created thread shares the same PID and gets assigned a 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(3) 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.

2.5. What is emulation

According to a dictionary definition, emulation 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 the 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 nor possible at all. For example someone might want to use a program developed for a different operating system than he uses. Then emulation comes in handy. Sometimes there is no other way but to use emulation - e.g. when the hardware device you try to use does 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 is 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 does not 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. It is obvious that there is no need for a complex emulation layer but simply reversing of the printed text is sufficient. Sometimes the emulating environment is very similar to the emulated one so just 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 do not affect some global kernel state.

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

No emulation is perfect and emulations tend to lack some parts but this usually does not 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(2)/write(2), fork(2) fam-

ily, signal(3) handling, exit(3), socket(2) API) hence it is easy to emulate because their semantics is shared among all UNIX®es, which exist todays.

3. Emulation

3.1. How emulation works in FreeBSD

As stated earlier, FreeBSD supports running binaries from several other UNIX®es. This works because FreeBSD has an abstraction called the execution class loader. This wedges into the execve(2) syscall, so when execve(2) 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 from type ELF. 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 the given 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(2) 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 provides a very elegant and easy way of supporting 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 is a handler called that possibly cleans up whatever the subsystem needs to be cleaned.

Those simple facilities provide basically everything that is 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 layers need some support from the operating system. I am going to describe some of the supported primitives in the FreeBSD operating system.

3.2.1. Locking primitives

Contributed by: Attilio Rao <attilio@FreeBSD.org >

The FreeBSD synchronization primitive 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 a high level point of view you can consider three kinds of synchronization primitives in the FreeBSD kernel:

- atomic operations and memory barriers
- locks
- scheduling barriers

Below there are descriptions for the 3 families. For every lock, you should really check the linked manpage (where possible) for more detailed explanations.

3.2.1.1. Atomic operations and memory barriers

Atomic operations are implemented through a set of functions performing simple arithmetics 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 directly in the end-level code, 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 based on just 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 with respect to other memory accesses. For example, if we need that an atomic operation happen 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 as well) should ideally only be used for building front-ending locks (as mutexes).

3.2.1.2. 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 does not exist currently, just check `sys/refcount.h` for an overview of the existing API.

3.2.1.3. Locks

FreeBSD kernel has huge classes 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's locking scheme presents three different behaviours for contenders:

1. spinning
2. blocking
3. sleeping

Note

numbers are not casual

3.2.1.4. Spinning locks

Spin locks let waiters to spin until they cannot acquire the lock. An important matter do deal with is when a thread contests on a spin lock if it is not descheduled. Since the 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 not to be used at all if not explicitly requested (explained later).

3.2.1.5. Blocking

Block locks let waiters to be descheduled and blocked until the lock owner does not 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.

3.2.1.6. Sleeping

Sleep locks let waiters to be descheduled and fall asleep until the lock holder does not 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 do not 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 2), it is allowed to acquire a spin lock (level 1) but not a sleep lock (level 3), 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:

- spin mutex - spinning - mutex(9)
- sleep mutex - blocking - mutex(9)
- pool mutex - blocking - mtx_pool(9)
- sleep family - sleeping - sleep(9) pause tsleep msleep msleep spin msleep rw msleep sx
- condvar - sleeping - condvar(9)
- rwlock - blocking - rwlock(9)
- sxlock - sleeping - sx(9)
- lockmgr - sleeping - lockmgr(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.

3.2.1.7. Scheduling barriers

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

- critical sections (and preemption)
- sched_bind
- 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 do not let the diagnose of simple eventual problems with locking debugging tools (as witness(4)).

3.2.1.8. 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 pre-

empted by interrupt threads (in this way, they do not 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(9) and critical_exit(9), 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.

3.2.1.9. 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()` and `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 is very effective in the particular case when we have to access at per-cpu datas and we assume other threads will not change those data. The latter condition will determine a critical section as a too strong condition for our code.

3.2.1.10. 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 executes the code, until a `sched_unbind` function call does not unbind it. This feature has a key role in situations where you cannot trust the current state of CPUs (for example, at very early stages of boot), as you want to avoid your thread to migrate on inactive CPUs. Since `sched_bind` and `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 sometimes require some additional per-process data. It can manage separate structures (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 is a pointer to the `sysentvec` described above. So by comparing this pointer to the address where the `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 per-

formance. This code should be turned into a macro because currently it is not very flexible, i.e. we do not support Linux®64 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, i.e. 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 handler 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.

3.2.3.1. namei

The namei(9) routine is a central entry point to pathname lookup and translation. It traverses the path point by point from the starting point to the end point using lookup function, which is internal to VFS. The namei(9) syscall can cope with symlinks, absolute and relative paths. When a path is looked up using namei(9) it is inputed to the name cache. This behaviour can be suppressed. This routine is used all over the kernel and its performance is very critical.

3.2.3.2. vn_fullpath

The vn_fullpath(9) 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 does not traverse the on medium structures), it does not work with hardlinks, etc. This routine is used in several places in the Linuxulator.

3.2.3.3. Vnode operations

- `fgetvp` - given a thread and a file descriptor number it returns the associated vnode
- vn_lock(9) - locks a vnode
- `vn_unlock` - unlocks a vnode
- VOP_READDIR(9) - reads a directory referenced by a vnode
- VOP_GETATTR(9) - gets attributes of a file or a directory referenced by a vnode
- VOP_LOOKUP(9) - looks up a path to a given directory
- VOP_OPEN(9) - opens a file referenced by a vnode
- VOP_CLOSE(9) - closes a file referenced by a vnode
- vput(9) - decrements the use count for a vnode and unlocks it

- vrele(9) - decrements the use count for a vnode
- vref(9) - increments the use count for a vnode

3.2.3.4. 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

4. Linux® emulation layer -MD part

This section deals with implementation of Linux® emulation layer in FreeBSD operating system. It first describes the machine dependent part talking about how and where interaction between userland and kernel is implemented. It talks about syscalls, signals, ptrace, traps, stack fixup. This part discusses i386 but it is written generally so other architectures should not differ very much. The next part is the machine independent part of the Linuxulator. This section only covers i386 and ELF handling. 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 is why it is 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 is 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 `syscalls.master` . The form of the file is:

```
...
 AUE_FORK STD  { int linux_fork(void); }
...
```

```
 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 is 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 do not prepend ABI prefix, etc. `UNIMPL` means that 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 `syscalls.master` a script generates three files: `linux_syscall.h`, `linux_proto.h` and `linux_sysent.c`. The `linux_syscall.h` contains definitions of syscall names and their numerical value, e.g.:

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

The `linux_proto.h` contains structure definitions of arguments to every syscall, e.g.:

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

And finally, `linux_sysent.c` contains structure describing the system entry table, used to actually dispatch a syscall, e.g.:

```
{ 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(2) so it has no linux arguments structure associated and in the system entry table it is not prefixed with linux as it calls the real close(2) 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` 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-dependent 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. Furthermore, 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 signal 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 could not be merged with a benefit of 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 the original process context. It also unmasks the signal in process signal mask.

4.3. Ptrace

Many UNIX® derivates implement the ptrace(2) 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(2) also lets you set various information in the traced process (registers etc.). ptrace(2) is a UNIX®-wide standard implemented in most UNIX®es around the world.

Linux® emulation in FreeBSD implements the ptrace(2) facility in `linux_ptrace.c` . The routines for converting registers between Linux® and FreeBSD and the actual ptrace(2) syscall emulation syscall. The syscall is a long switch block that implements its counterpart in FreeBSD for every ptrace(2) command. The ptrace(2) 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 a pointer to the data so after performing a (native) ptrace(2) syscall, a copyout must be done to preserve Linux® semantics.

The ptrace(2) implementation in Linuxulator has some known weaknesses. There have been panics seen when using `strace` (which is a ptrace(2) 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 works in a smart way.

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 does not work with it but this does not cause any problem, as the linux-base in ports probably do not 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.

5. Linux® emulation layer -MI part

This section 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 NPTL

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 resource 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 is usually considered a part of NPTL. 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 deal 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 sysctl(8), namely `compat.linux.osrelease` , which is set to 2.4.2 by default (as of April 2007) and with all Linux® versions up to 2.6 it just determined what uname(1) outputs. It is different with 2.6 emulation where setting this sysctl(8) 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 whether 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(8) should not be changed when running any Linux® binary as it might harm things.

5.2.2. Linux® processes 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 is 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 (and shared VM etc. of course). Everything else happens in the same way as to 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 is really not very consistent and looks error-prone.

Every NPTL 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 mimicked in the 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_se_tid` and `child_clear_tid` are used for TID address copyout when a process exits and is created. The `shared` pointer points to a structure shared among threads. The `pdeath_signal` variable identifies the parent death signal and the `threads` pointer is used to link this structure to the list of threads. The `linux_emuldata_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). 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`. The prototype of the function is:

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

Here, `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 knowing 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_tid` and `child_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 processes sharing various things independently, for example two processes can share file descriptors but not VM, etc. Last byte of the `flags` parameter is the 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 (i.e. do what normal fork(2) routine does). The `parent_tidptr` parameter is used as an address for copying out process PID (i.e. thread id) once the process is sufficiently instantiated but is not runnable yet. The `dummy` parameter is here because of the very strange calling convention of this syscall on i386. It uses the registers directly and does not let the compiler do it what 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` and `CLONE_FILES` because FreeBSD does not 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 does not cause any problems, because those flags are always set together. After setting the flags the process is forked using the internal `fork1` routine, the process is instrumented not to be put on a run queue, i.e. not to be set runnable. After the forking is done we possibly reparent the newly created process to emulate `CLONE_PARENT` semantics. Next part is creating the emulation data. Threads in Linux® does not signal their parents so we set exit signal to be `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` . 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(2) semantics 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 is used for emulating classic fork(2) and vfork(2) syscalls. Newer glibc in a case of 2.6 kernel uses `clone` to implement fork(2) and vfork(2) syscalls.

5.2.5. Locking

The locking is implemented to be per-subsystem because we do not 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 `linux_emuldata_shared` . The `emul_lock` is a nonsleepable blocking mutex while `emul_shared_lock` is a sleepable blocking `sx_lock` . Because of the per-subsystem locking we can coalesce some locks and that is why the em find offers the non-locking access.

5.3. TLS

This section deals with TLS also known as 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. The threads in the process share process wide data (file descrip-

tors, etc.) but also have their 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(3), pthread_setspecific(3) and pthread_getspecific(3) where a thread can create a key to the thread local data and using pthread_getspecific(3) or pthread_getspecific(3) to 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 of 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 the whole 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 price for context switches.

5.3.2. Segments on i386

The i386 architecture implements the 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 let us 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 a special use, for example `%cs` is used for code segment and `%ss` is used 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 up to 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 ma-

chine 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 the system itself should choose the first free slot. In practice, the vast majority of programs use only one TLS entry and does not 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

5.3.4.1. 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 the Linux® form segment descriptor from the 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 did not work) as the userspace process typically passes in 1. Then we convert the descriptor from the Linux® form to a machine dependant form (i.e. 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 a critical section so that 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 does not add any overhead over plain FreeBSD.

5.3.4.2. amd64

The amd64 implementation is similar to the 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). Apart from this the TLS loading is exactly the same just the segment numbers are different and the descriptor format and the loading differs slightly.

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, i.e. not fast) and simple. In 1:1 model, the situation is also quite clear - the threads must be synchronized using kernel facilities (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, the most well written threaded programs show little locks contention.

5.4.2. Futexes introduction

Linux® implements 1:1 threading, i.e. it has to use in-kernel synchronization primitives. As stated earlier, well written threaded programs have little lock contention. So a typical sequence could be performed as two atomic increase/decrease mutex reference counter, which is very fast, as presented by the following example:

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

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 takes place 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);
```

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

Futexes implement the following operations:

- FUTEX_WAIT
- FUTEX_WAKE
- FUTEX_FD
- FUTEX_REQUEUE
- FUTEX_CMP_REQUEUE
- FUTEX_WAKE_OP

5.4.3.1. 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 gets suspended. If the argument timeout is non-zero it specifies the maximum time for the sleeping, otherwise the sleeping is infinite.

5.4.3.2. FUTEX_WAKE

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

5.4.3.3. FUTEX_FD

This operations associates a file descriptor with a given futex.

5.4.3.4. FUTEX_REQUEUE

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

5.4.3.5. FUTEX_CMP_REQUEUE

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

5.4.3.6. 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 a positive number it wakes up val2 threads on futex at uaddr2.

The operations implemented in FUTEX_WAKE_OP :

- FUTEX_OP_SET
- FUTEX_OP_ADD
- FUTEX_OP_OR

- FUTEX_OP_AND
- FUTEX_OP_XOR

Note

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_paroc, waiting_proc) f_waiting_proc;
};
```

And the structure waiting_proc is:

```
struct waiting_proc {

  struct thread *wp_t;

  struct futex *wp_new_futex;

  TAILQ_ENTRY(waiting_proc) wp_list;
};
```

5.4.4.1. futex_get / futex_put

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

5.4.4.2. 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(9) to sus-

pend the thread. The sleep can be timed out. After tsleep(9) returns (the thread was woken up or it timed out) the `working_proc` structure is removed from the list and is destroyed. All this is done in the `futex_sleep` function. If we got woken up from `futex_wake` we have `wp_new_futex` set so we sleep on it. This way the actual requeueing is done in this function.

5.4.4.3. futex_wake

Waking up a thread sleeping on a futex is performed in the `futex_wake` function. First in this function we 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 does not 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` .

5.4.4.4. futex_wake_op

The `FUTEX_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` (i.e. `timeout`) waiter on the second futex.

5.4.4.5. 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 this one:

```
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 `oldval` is compared to `cmparg` argument with cmp comparator.

5.4.4.6. Futex locking

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

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 the `filename` parameter is absolute `dirfd` is ignored but when the path to the file is relative, it comes to the play. The `dirfd` parameter is a directory relative to which the relative pathname is checked. The `dirfd` parameter 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 the working directory. Imagine that a process consists of two threads, thread A and thread B. Thread A issues `open(./tmp/foo/bah., flags, mode)` and before returning it gets preempted and thread B runs. Thread B does not care about the needs of thread A and renames or removes `/tmp/foo/`. We got a race. To avoid this we can open `/tmp/foo` and use it as `dirfd` for `openat` syscall. This also enables user to implement per-thread working directories.

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(9) routine and simple wrapping layer.

5.5.1.1. Implementation

The implementation is done by altering the namei(9) routine (described above) to take additional parameter `dirfd` in its `nameidata` structure, which specifies the starting point of the pathname lookup instead of using the current working directory every time. The resolution of `dirfd` 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 is vnode attached to it then we get a vnode. Then we check this vnode for being a directory. In the actual namei(9) routine we simply substitute the `dvp` vnode for `dp` variable in the namei(9) function, which determines the starting point. The namei(9) 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()
```

For this reason `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 syscalls. This is being discussed right now.

5.5.2. Ioctl

The ioctl interface is quite fragile due to 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, the file descriptor, command and an argument. The command is a 16-bit number, which in theory is divided into high 8 bits determining class of the ioctl command and low 8 bits, which are the actual command within the given set. The emulation takes advantage of this division. We implement handlers for each set, like `sound_handler` or `disk_handler` . Each handler has a maximum command and a 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 the easy porting of applications.

5.5.3. Debugging

Every syscall should be debuggable. For this purpose we introduce a small infrastructure. We have the 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 debugging messages.

6. Conclusion

6.1. Results

As of April 2007 the Linux® emulation layer is capable of emulating the Linux® 2.6.16 kernel quite well. The remaining problems concern futexes, unfinished *at family of syscalls, problematic signals delivery, missing `epoll` and `inotify` and probably some bugs we have not 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 committed 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 www/linux-firefox, www/linux-opera, net-im/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 the 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 for 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 signal delivery and possibly implement the `epoll` and `inotify` facilities.

We hope to be able to run the most important programs flawlessly soon, so we will be able to switch to the 2.6 emulation by default and make the Fedora Core 6 the default linux_base because our currently used Fedora Core 4 is not supported any more.

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. Some already implemented syscalls are also 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 <`jhb@FreeBSD.org` >
- Konstantin Belousov <`kib@FreeBSD.org` >
- Emmanuel Dreyfus
- Scot Hetzel
- Jung-uk Kim <`jkim@FreeBSD.org` >
- Alexander Leidinger <`netchild@FreeBSD.org` >
- Suleiman Souhlal <`ssouhlal@FreeBSD.org` >

- Li Xiao
- David Xu <davidxu@FreeBSD.org >

I would like to thank all those people for their advice, code reviews and general support.

7. Literatures

1. Marshall Kirk McKusick - George V. Nevile-Neil. Design and Implementation of the FreeBSD operating system. Addison-Wesley, 2005.
2. http://www.FreeBSD.org
3. http://tldp.org
4. http://www.linux.org