# Emulación Linux® en FreeBSD

Resumen

Esta tesis doctoral trata sobre cómo actualizar la capa de emulación de Linux® (también llamada *Linuxulator*). La tarea consistía en actualizar dicha capa para alcanzar en funcionalidad a Linux® 2.6. Como implementación de referencia se escogió el kernel Linux® 2.6.16. El concepto se basa ligeramente en la implementación de NetBSD. La mayoría del trabajo se realizó en el verano de 2006 como parte del programa de estudiantes Google Summer of Code. El foco se situó en añadir soporte para *NPTL* (la nueva librería de hilos POSIX®) a la capa de emulación, incluyento *TLS* (almacenamiento local para hilos), *futexes* (mutex rápidos en espacio de usuario), *PID mangling* y otras cosas menores. En el proceso se identificaron y arreglaron muchos problemas menores. Mi trabajo se integró en el repositorio fuente principal de FreeBSD y estará disponible en la próxima versión 7.0R. Los miembros del equipo de desarrollo de emulación estamos trabajando para que la emulación de Linux® 2.6 sea la capa de emulación por defecto en FreeBSD.

# Tabla de contenidos

1. Introducción . . . 1
2. Una mirada al interior ... . . . 2
3. Emulación . . . 10
4. Parte MD de la capa de emulación de Linux® . . . 16
5. Parte MI de la capa de emulación Linux® . . . 20
6. Conclusión . . . 31
7. Bibliografía . . . 32

# 1. Introducción

En los últimos años, los sistemas operativos basados en el código abierto de UNIX® han empezado a ser desplegados ampliamente tanto en máquinas cliente como servidores. Entre estos sistemas operativos me gustaría resaltar dos: FreeBSD, por su herencia BSD, base de código que resiste el paso del tiempo y por tener muchas características interesantes y Linux® por su amplio número de usuarios, comunidad de desarrolladores entusiasta y abierta y el apoyo de grandes corporaciones. FreeBSD se suele utilizar en máquinas de tipo servidor que realizan duras tareas intensivas de red con menos uso en máquinas de tipo escritorio para usuarios ordinarios. Aunque Linux® tiene el mismo uso en servidores, es mucho más usado por usuarios en sus casas. Esto lleva a una situación en la que hay muchos programas sólo disponibles en forma binaria para Linux® y que no tienen soporte para FreeBSD.

Naturalmente, surge la necesidad de ejecutar binarios de Linux® en un sistema FreeBSD y eso es de lo que trata esta tesis: la emulación del kernel Linux® en el sistema operativo FreeBSD.

En el verano de 2006 Google Inc. patrocinó un proyecto enfocado en extender la capa de emulación Linux® (el llamado Linuxulator) en FreeBSD para incluir las capacidades de Linux® 2.6. Esta tesis se escribió como parte de este proyecto.

# 2. Una mirada al interior ...

En esta sección vamos a describir cada sistema operativo en cuestión. Cómo manejan las llamadas al sistema, trapframes, etc., todo lo que sea bajo nivel. También describimos la manera en la que entienden primitivas comunes de UNIX® como qué es un PID, qué es un hilo, etc. En la tercera subsección hablamos acerca de cómo se podría hacer emulación UNIX® sobre UNIX® de forma general.

## 2.1. Qué es UNIX®

UNIX® es un sistema operativo con una larga historia que ha influenciado a casi todos los sistemas operativos que se utilizan actualmente. Comenzando en 1960, su desarrollo continúa en la actualidad (aunque en diferentes proyectos). El desarrollo de UNIX® pronto se dividió en dos ramas principales: las familias BSD y System III/V. Ambas se influenciaron mutuamente haciendo crecer el estándar UNIX®. Entre las contribuciones que se originaron en BSD podemos nombrar la memoria virtual, las redes TCP/IP, FFS, y muchos otros. La rama System V aportó las primitivas SysV de comunicación entre procesos, el copy-on-write, etc. UNIX® en sí mismo ya no existe pero sus ideas se han usado en muchos otros sistemas operativos por todo el mundo formando así los llamados sistemas operativos tipo UNIX®. Actualmente los más influyentes son Linux®, Solaris, y posiblemente (hasta cierto punto) FreeBSD. Hay derivados de UNIX® internos en algunas compañías (AIX, HP-UX etc.) pero estos han sido migrados cada vez más a los sistemas mencionados anteriormente. Resumamos las características típicas de UNIX®.

## 2.2. Detalles técnicos

Cada programa en ejecución constituye un proceso que representa el estado de la computación. Un proceso en ejecución se divide entre espacio del kernel y espacio de usuario. Algunas operaciones sólo se pueden hacer en espacio de kernel (tratar con hardware etc.), pero el proceso debería pasar la mayoría de su vida en espacio de usuario. El kernel es donde tienen lugar la gestión de los procesos, hardware y los detalles de bajo nivel. El kernel proporciona al espacio de usuario un API UNIX® estándar y unificado. Las más importantes se tratan abajo.

### 2.2.1. Comunicación entre el kernel y el proceso de espacio de usuario

El API común de UNIX® define llamadas al sistema como forma de ejecutar comandos en el kernel desde espacio de usuario. La implementación más habitual es utilizar una interrupción o una instrucción especializada (como las instrucciones `SYSENTER`/`SYSCALL` en ia32). Las llamadas al sistema se definen mediante un número. Por ejemplo en FreeBSD, la llamada al sistema número 85 es la llamada al sistema de swapon(2) y la llamada al sistema número 132 es mkfifo(2). Algunas llamadas al sistema necesitan parámetros, que son pasados desde espacio de usuario a espacio de kernel de varias formas (dependiendo de la implementación). Las llamadas al sistema son síncronas.

Otra forma posible de comunicarse es mediante un *trap*. Los traps (trampas) ocurren de forma

asíncrona después de que ocurra algún evento (división por cero, fallo de página, etc.). Un trap puede ser transparente para un proceso (error de página) o puede resultar en una reacción como enviar una *señal* (división por cero).

### 2.2.2. Comunicación entre procesos

Hay otras API (System V IPC, memoria compartida, etc.) pero la API más importante es la señal. Las señales son enviadas por procesos o por el kernel y recibidas por procesos. Algunas señales pueden ser ignoradas o manejadas por una rutina proporcionada por el usuario, otras dan como resultado una acción predefinida que no se puede alterar ni ignorar.

### 2.2.3. Gestión de procesos

Las instancias del kernel se procesan las primeras en el sistema (llamado init). Cada proceso en ejecución puede crear una copia idéntica a sí mismo utilizando la llamada al sistema fork(2). Se han introducido algunas versiones ligeramente modificadas de esta llamada pero la semántica es básicamente la misma. Cada proceso en ejecución se puede convertir en otro proceso utilizando la llamada al sistema exec(3). Se han introducido algunas modificaciones a esta llamada pero todas tienen básicamente el mismo propósito. Los procesos terminan sus vidas invocando la llamada al sistema exit(2). Cada proceso se identifica por un número único llamado PID. Cada proceso tiene definido un padre (identificado por su PID).

### 2.2.4. Gestión de hilos

Los UNIX® tradicionales no definen ni un API ni una implementación para hilos, mientras que POSIX® define un API para hilos pero la implementación no está definida. Tradicionalmente había dos formas de implementar hilos. Manejarlos como procesos separados (modelo 1:1) o envolver todo el grupo de hilos en un proceso y manejar los hilos en espacio de usuario (modelo 1:N). Comparando las características principales de cada aproximación:

Hilos 1: 1

- hilos pesados
- el usuario no puede alterar la planificación (ligeramente mitigado por el API de POSIX®) + no es necesario un recubrimiento para la llamada al sistema + puede utilizar varias CPU

Hilos 1: N

+ hilos ligeros + el usuario puede modificar fácilmente la planificación - las llamadas al sistema necesitan estar recubiertas - no puede utilizar más de una CPU

## 2.3. ¿Qué es FreeBSD?

El proyecto FreeBSD es uno de los sistemas operativos open source más antiguos que están actualmente disponibles para uso diario. Es un descendiente directo del UNIX® genuino así que se podría afirmar que es un UNIX® verdadero aunque asuntos con las licencias no permiten hacerlo. El inicio del proyecto data de principios de los 90 cuando un grupo de usuarios de BSD parchearon el sistema operativo 386BSD. Basado en este conjunto de parches surgió un nuevo sistema

operativo llamado FreeBSD debido a su licencia liberal. Otro grupo creó el sistema operativo NetBSD pensando en diferentes objetivos . Nos centraremos en FreeBSD.

FreeBSD es un sistema operativo moderno basado en UNIX® con todas las características de UNIX®. Multitarea preemptiva, capacidades multiusuario, redes TCP/IP, protección de memoria, soporte para multiprocesamiento simétrico, memoria virtual con cache de buffer y VM fusionadas, todo está ahí. Una de las características interesantes y extremadamente útiles es la habilidad de emular otros sistemas operativos tipo UNIX®. A fecha de Diciembre de 2006 en el desarrollo de 7-CURRENT, se soportan las siguientes características de emulación:

- Emulación FreeBSD/i386 en FreeBSD/amd64
- Emulación FreeBSD/i386 en FreeBSD/ia64
- Emulación del sistema operativo Linux® en FreeBSD
- Emulación NDIS de la interfaz de controladores de red de Windows
- Emulación NetBSD del sistema operativo NetBSD
- Soporte PECoff para ejecutables PECoff FreeBSD
- Emulación del UNIX® System V revision 4

Las emulaciones activamente en desarrollo son la capa de Linux® y las capas de FreeBSD sobre FreeBSD. Otras no están soportadas para funcionar correctamente o no son utilizables actualmente.

### 2.3.1. Detalles técnicos

FreeBSD es una versión tradicional de UNIX® en el sentido en el que divide la ejecución de los procesos en dos mitades: espacio de kernel y ejecución en espacio de usuario. Hay dos tipos de entradas al kernel para los procesos: una llamada al sistema y un trap. Sólo hay una forma de volver. En las siguientes secciones se describirán las tres puertas desde/hacia el kernel. Toda la descripción aplica a la arquitectura i386 ya que el Linuxulator sólo existe ahí pero el concepto es similar para otras arquitecturas. La información se ha tomado de [1] y del código fuente.

#### Entradas del sistema

FreeBSD tiene una abstracción denominada cargador de clases de ejecución que es un enganche a la llamada al sistema execve(2). Esta emplea una estructura `sysentvec` que describe el ABI de un ejecutable. Contiene cosas como la tabla de traducción de errno, la tabla de traducción de señales, varias funciones para satisfacer las necesidades de las llamadas al sistema (fixups de la pila, volcado de cores, etc). Cada ABI que el kernel de FreeBSD quiera soportar debe definir esta estructura puesto que es utilizada después el código de procesamiento de la llamada al sistema y en algunos otros sitios. Las entradas al sistema se manejan mediante manejadores de traps donde podemos acceder al espacio del kernel y de usuario al mismo tiempo.

#### Llamadas al sistema

Las llamadas al sistema en FreeBSD se llevan a cabo ejecutando la interrupción `0x80` con el registro `%eax` establecido al número de la llamada deseado y con los argumentos pasados en la pila.

Cuando un proceso realiza una interrupción `0x80`, se invoca el manejador de trap de llamada al

sistema `int0x80` (definido en sys/i386/i386/exception.s), el cual prepara los argumentos (es decir, los copia a la pila) para llamar a una función C syscall(2) (definida en sys/i386/i386/trap.c) que procesa el marco de trap pasado. El procesamiento consiste en preparar la llamada al sistema (dependiendo de la entrada de `sysvec`), determinar si la llamada es de 32 o 64 bit (cambia el tamaño de los parámetros), luego copiar los parámetros incluyendo la llamada al sistema. Después, se ejecuta la llamada al sistema real procesando el código de retorno (casos especiales para los errores `ERESTART` y `EJUSTRETURN`). Por último se planifica un `userret()`, cambiando el proceso de nuevo a espacio de usuario. Los parámetros para la llamada al sistema real se pasan con la forma de los argumentos `struct thread *td`, `struct syscall args *` donde el segundo parámetro es un puntero a la estructura de parámetros copiada.

#### Trampas

El manejo de traps en FreeBSD es similar al manejo de llamadas al sistema. Siempre que ocurre un trap, se llama a un manejador en ensamblador. Se elige entre todos los traps, aquellas con registros empujados a la pila o traps de llamadas dependiendo del tipo de trap. Este controlador prepara argumentos para una llamada a una función C `trap()` (definida en sys/i386/i386/trap.c), que luego procesa el trap ocurrido. Después del procesamiento, puede enviar una señal al proceso y / o salir al espacio de usuario usando `userret()`.

#### Salida

Las salidas del kernel al espacio de usuario ocurren usando la rutina en ensamblador `doreti` independientemente de si se ingresó al kernel mediante un trap o mediante una llamada al sistema. Esto restaura el estado del programa desde la pila y vuelve al espacio de usuario.

#### Primitivas UNIX®

El sistema operativo FreeBSD sigue el esquema tradicional UNIX®, donde cada proceso tiene un número único de identificación, el llamado *PID* (Process ID). Los números PID se generan o linealmente o de forma aleatoria en el rango `0` a `PID_MAX`. La generación de números PID se hace usando una búsqueda lineal en el espacio PID. Cada hilo en un proceso recibe el mismo número PID como resultado de llamar a getpid(2).

Actualmente hay dos formas de implementar multihilo en FreeBSD. La primera es M:N seguido del modelo 1:1. La librería usada por defecto es multihilo M:N (`libpthread`) y puedes cambiar en tiempo de ejecución a multihilo 1:1 (`libthr`). El plan es cambiar pronto a la librería 1:1 por defecto. Aunque estas dos librerías utilizan las mismas primitivas del kernel, se acceden mediante APIs diferentes. La librería M:N utiliza la familia `kse_*` de llamadas al sistema mientras que la librería 1:1 utiliza la familia `thr_*` de llamadas al sistema. Debido a esto, no hay un concepto general de ID de hilo compartido entre el kernel y el espacio de usuario. Por supuesto, ambas librerías implementan el API de ID de hilo de pthread. Cada hilo del kernel (descrito por `struct thread`) tiene el identificador tid pero no es directamente accesible desde espacio de usuario y sólo sirve para cubrir necesidades del kernel. También se usa para la librería 1:1 como el ID de hilo de pthread pero este manejo es interno a la librería y no se puede depender de él.

Como se indicó anteriormente, hay dos implementaciones de multihilo en FreeBSD. La biblioteca M:N divide el trabajo entre el espacio del kernel y el espacio de usuario. El hilo es una entidad que se planifica en el kernel, pero puede representar varios hilos en espacio de usuario. M hilos en

espacio de usuario se asignan a N hilos del núcleo, lo que ahorra recursos y mantiene la capacidad de explotar el paralelismo de multiprocesador. Se puede obtener más información sobre la implementación en la página del manual o en [1]. La biblioteca 1:1 mapea directamente un hilo de espacio de usuario a un hilo del kernel, lo que simplifica enormemente el esquema. Ninguno de estos diseños implementa un mecanismo de equidad (se implementó un mecanismo de este tipo, pero se eliminó recientemente porque causaba una grave lentitud y hacía que el código fuera más difícil de tratar).

## 2.4. Qué es Linux®

Linux® es un kernel de tipo UNIX® desarrollado originalmente por Linus Torvalds, y al que ahora contribuye un enorme número de programadores en todo el mundo. Desde sus primeros comienzos hasta ahora, con amplio apoyo de compañías como IBM o Google, Linux® se ha asociado con su rápido ritmo de desarrollo, amplio soporte hardware y su modelo de organización de tipo dictador benevolente.

El desarrollo de Linux® comenzó como un hobby en 1991 en la Universidad de Helsinki en Finlandia. Desde entonces ha adquirido todas las características de un sistema operativo moderno tipo UNIX®: multiprocesamiento, soporte multiusuario, memoria virtual, redes, básicamente lo tiene todo. También hay características altamente avanzadas como virtualización, etc.

A fecha de 2006 Linux® parece ser el sistema operativo open source más ampliamente usado con soporte de empresas de software independientes como Oracle, RealNetworks, Adobe, etc. La mayoría del software comercial que se distribuye para Linux® sólo se puede obtener en forma binaria de forma que recompilar para otros sistemas operativos es imposible.

La mayoría del desarrollo de Linux® tiene lugar en el sistema de control de versiones Git. Git es un sistema distribuido de forma que no hay una fuente de código central de Linux®, pero algunas ramas se consideran prominentes y oficiales. El esquema de numeración de versiones implementado por Linux® consiste en cuatro números A.B.C.D. El desarrollo actual tiene lugar en 2.6.C.C, donde C representa la versión mayor, donde se cambian o añaden nuevas características mientras que D es la versión menor sólo para arreglos de bugs.

Se puede obtener más información en [3].

### 2.4.1. Detalles técnicos

Linux® sigue el esquema tradicional UNIX® de dividir la ejecución de un proceso en dos partes: espacio de kernel y espacio de usuario. Al kernel se puede entrar de dos formas: vía trap o vía llamada al sistema. La vuelta se maneja de una sola forma. La descripción que sigue aplica a Linux® 2.6 en la arquitectura i386™. La información se ha obtenido de [2].

#### Llamadas al sistema

Las llamadas al sistema en Linux® se realizan (en espacio de usuario) utilizando las macros `syscallX` donde X se sustituye por el número que representa el número de parámetros de la llamada al sistema. Esta macro traduce a un código que carga el registro `%eax` con un número de llamada al sistema y ejecuta la interrupción `0x80`. Después de la llamada al sistema se llama a return, que traslada valores de retorno negativos a valores positivos `errno` y establece `res` a `-1` en

caso de error. Cada vez que se llama a la interrupción `0x80` el proceso entra en el kernel en un manejador de llamada al sistema. Esta rutina salva todos los registros en la pila y llama a la entrada de llamada al sistema seleccionada. Nótese que la convención de llamadas de Linux® espera que los parámetros de la llamada al sistema se pasen vía registros como se muestra aquí:

1. parámetro → `%ebx`
2. parámetro → `%ecx`
3. parámetro → `%edx`
4. parámetro → `%esi`
5. parámetro → `%edi`
6. parámetro → `%ebp`

Hay algunas excepciones a esta regla, donde Linux® utiliza una convención de llamadas diferente (principalmente en la llamada al sistema `clone`).

#### Trampas

Los manejadores de traps se encuentran en arch/i386/kernel/traps.c y la mayoría de estos manejadores viven en arch/i386/kernel/entry.S, donde ocurre el manejo de los traps.

#### Salida

La vuelta de la llamada al sistema se gestiona mediante la llamada al sistema exit(3) que comprueba que el proceso no tenga trabajo sin finalizar, luego comprueba si hemos utilizado los selectores proporcionados por el usuario. Si esto sucede se aplica un fix a la pila y finalmente se restauran los registros desde la pila y el proceso vuelve a espacio de usuario.

#### Primitivas UNIX®

En la versión 2.6, el sistema operativo Linux® redefinió algunas de las primitivas tradicionales de UNIX®, en particular PID, TID e hilo. PID no se define como único para cada proceso, así que para algunos procesos (hilos) getpid(2) devuelve el mismo valor. La identificación única de proceso se proporciona mediante TID. Esto es así porque *NPTL* (New POSIX® Thread Library) define los hilos como procesos normales (el llamado multihilo 1:1). Crear un nuevo proceso en Linux® 2.6 se hace utilizando la llamada al sistema `clone` (las variantes de fork se reimplementan usando esta). Esta llamada al sistema clone define una serie de flags que afecta el comportamiento de los procesos clonados respecto a la implementación del multihilo. La semántica es un poco difusa ya que no hay un único flag para decirle a la llamada al sistema que cree un hilo.

Las banderas de clonado implementadas son:

- `CLONE_VM` - los procesos comparten su espacio de memoria
- `CLONE_FS` - comparte umask, cwd y namespace (espacio de nombres)
- `CLONE_FILES` - comparte ficheros abiertos
- `CLONE_SIGHAND` - comparte manejadores de señales y señales bloqueadas
- `CLONE_PARENT` - comparte padre

- CLONE_THREAD - sé un hilo (más explicación abajo)
- CLONE_NEWNS - nuevo espacio de nombres
- CLONE_SYSVSEM - comparte estructuras para deshacer operaciones en semáforos de SysV
- CLONE_SETTLS - establece TLS en la dirección proporcionada
- CLONE_PARENT_SETTID - establece TID en el padre
- CLONE_CHILD_CLEARTID - borra TID en el hijo
- CLONE_CHILD_SETTID - establece TID en el hijo

CLONE_PARENT establece el padre real al padre del llamante. Esto es útil para los hilos porque si el hilo A crea el hilo B queremos que el padre del hilo B sea todo el grupo de hilos. CLONE_THREAD hace exactamente lo mismo que CLONE_PARENT, CLONE_VM y CLONE_SIGHAND, reescribe el PID para que sea igual al PID del llamante, blanquea la señal exit y se une al grupo de hilos. CLONE_SETTLS establece las entradas GDT para el manjeo de TLS. El conjunto de flags CLONE_*_*TID establece/borra la dirección proporcionada por el usuario a TID o 0.

Como puedes ver CLONE_THREAD hace la mayor parte del trabajo y no parece encajar muy bien en el esquema. La intención original no está clara (incluso para los autores, según los comentarios en el código), pero creo que originalmente había un flag de hilo, que luego se dividió entre muchos otros flags pero esta separación nunca se terminó por completo. Tampoco está claro para qué sirve esta partición, ya que glibc no la usa, por lo que solo el uso a mano de clone permite al programador acceder a estas funciones.

Para programas no multihilo el PID y el TID son iguales. Para programas multihilo el PID y el TID del primer hilo son el mismo y cada hilo que se crea comparte el mismo PID y se le asigna un TID único (porque se pasa CLONE_THREAD) también se comparte el padre en todos los procesos que forman este programa multihilo.

El código que implementa pthread_create(3) en NPTL define los flags de clone así:

```
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);
```

CLONE_SIGNAL se define como

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

el último 0 significa que no se envía ninguna señal cuando alguno de los hilos sale.

## 2.5. Que es la emulación

Según una definición de diccionario, la emulación es la capacidad de un programa o dispositivo para imitar otro programa o dispositivo. Esto se logra proporcionando la misma reacción a un estímulo dado que la que produce el objeto emulado. En la práctica, en el mundo del software hay principalmente tres tipos de emulación: un programa utilizado para emular una máquina (QEMU, varios emuladores de consola de juegos, etc.), emulación de software de una instalación de hardware (emuladores OpenGL, emulación de unidades de punto flotante, etc.) y emulación del sistema operativo (ya sea en el núcleo del sistema operativo o como un programa de espacio de usuario).

La emulación se usa generalmente en un lugar donde usar el componente original no es factible ni posible en absoluto. Por ejemplo, alguien podría querer usar un programa desarrollado para un sistema operativo diferente al que usa. Entonces la emulación es útil. A veces no hay otra forma que usar la emulación, por ejemplo, cuando el dispositivo de hardware que intentas utilizar no existe (todavía/más), no hay otra forma que la emulación. Esto sucede a menudo cuando se traslada un sistema operativo a una plataforma nueva (inexistente). A veces es más barato emular.

Desde el punto de vista de la implementación, hay dos enfoques principales para la implementación de la emulación. Puedes emular todo, aceptando posibles entradas del objeto original, manteniendo el estado interno y emitiendo la salida correcta según el estado y/o la entrada. Este tipo de emulación no requiere condiciones especiales y básicamente se puede implementar en cualquier lugar para cualquier dispositivo/programa. El inconveniente es que implementar tal emulación es bastante difícil, requiere mucho tiempo y es propenso a errores. En algunos casos, podemos utilizar un enfoque más simple. Imagina que quieres emular una impresora que imprime de izquierda a derecha en una impresora que imprime de derecha a izquierda. Es obvio que no hay necesidad de una capa de emulación compleja, pero basta con invertir el texto impreso. A veces, el entorno de emulación es muy similar al emulado, por lo que solo se necesita una capa fina de traducción para proporcionar una emulación completamente funcional. Como puedes ver, esto es mucho menos exigente de implementar, por lo que consume menos tiempo y es menos propenso a errores que el enfoque anterior. Pero la condición necesaria es que los dos entornos sean lo suficientemente similares. El tercer enfoque combina los dos anteriores. La mayoría de las veces los objetos no brindan las mismas capacidades, por lo que en el caso de emular el más potente en el menos potente, tenemos que emular las características faltantes con la emulación completa descrita anteriormente.

Esta tesis trata de la emulación de UNIX® en UNIX®, que es exactamente el caso, donde una fina capa de traducción es suficiente para proporcionar emulación completa. El API UNIX® consiste en un conjunto de llamadas al sistema, las cuales están normalmente autocontenidas y no afectan al estado global del kernel.

Hay algunas llamadas al sistema que afectan el estado interno, pero esto se puede solucionar proporcionando algunas estructuras que mantienen el estado adicional.

Ninguna emulación es perfecta y las emulaciones tienden a carecer de algunas partes, pero esto no suele causar inconvenientes graves. Imagina un emulador de consola de juegos que emula todo

menos la salida de música. No hay duda de que los juegos se pueden jugar y se puede usar el emulador. Puede que no sea tan cómodo como la consola de juegos original, pero es un compromiso aceptable entre precio y comodidad.

Lo mismo aplica al API de UNIX®. La mayoría de los programas pueden vivir con un conjunto muy limitado de llamadas al sistema funcionales. Esas llamadas al sistemas suelen ser las más antiguas (read(2)/write(2), la familia fork(2), manejo de signal(3), exit(3), socket(2) API) y por lo tanto es fácil de emular porque sus semánticas se comparten entre todos los UNIX® que existen a día de hoy.

# 3. Emulación

## 3.1. Cómo funciona la emulación en FreeBSD

Como se ha mencionado antes, FreeBSD suporta ejecutar binarios de otros UNIX®. Esto funciona porque FreeBSD tiene una capa de abstracción llamada el cargador de clases de ejecución. Este se inserta en la llamada al sistema execve(2) de forma que cuando execve(2) está a punto de ejecutar un binario examina su tipo.

Básicamente, existen dos tipos de binarios en FreeBSD. Scripts de texto tipo shell que se identifican por `#!` como sus dos primeros caracteres y binarios (typically *ELF*) normales, que son una representación de un objeto compilado ejecutable. La gran mayoría (se podría decir que todos) de los binarios en FreeBSD son del tipo ELF. Los archivos ELF contienen un encabezado, que especifica la ABI del sistema operativo para este archivo ELF. Al leer esta información, el sistema operativo puede determinar con precisión de qué tipo de binario es el archivo dado.

Cada ABI de sistema operativo tiene que estar registrada en el kernel de FreeBSD. Esto aplica también al ABI nativo de FreeBSD. Cuando execve(2) ejecuta un binario itera por la lista de APIs registradas y cuando encuentra la correcta usa la información contenida en la descripción del ABI (su tabla de llamadas al sistema, tabla de traducción de `errno`, etc.). Así que cada vez que un proceso realiza una llamada al sistema, utiliza su propio conjunto de llamadas al sistema en lugar de uno global. Esto de forma efectiva proporciona una forma muy elegante de soportar la ejecución de varios formatos binarios.

La naturaleza de la emulación de diferentes sistemas operativos (y también algunos otros subsistemas) llevó a los desarrolladores a introducir un mecanismo de manejadores de eventos. Hay varios lugares en el kernel, donde se llama a una lista de manejadores de eventos. Cada subsistema puede registrar un manejador de eventos y se los llama en consecuencia. Por ejemplo, cuando un proceso termina, se llama a un manejador que posiblemente limpia lo que sea que el subsistema necesite que se limpie.

Esos simples servicios básicamente proporcionan todo lo que se necesita para la infraestructura de emulación y de hecho esto es básicamente lo único necesario para implementar la capa de emulación Linux®.

## 3.2. Primitivas comunes en el kernel de FreeBSD

Las capas de emulación necesitan soporte del sistema operativo. Voy a describir algunas de las

primitivas soportadas en el sistema operativo FreeBSD.

### 3.2.1. Primitivas de bloqueo

Aportado por: Attilio Rao <attilio@FreeBSD.org>

El conjunto de primitivas de sincronización de FreeBSD se basa en la idea de suministrar un número bastante grande de primitivas diferentes de manera que se pueda utilizar la mejor para cada situación particular y apropiada.

Desde un punto de vista de alto nivel, puede considerar tres tipos de primitivas de sincronización en el kernel de FreeBSD:

- operaciones atómicas y barreras de memoria
- Locks
- barreras de planificación

A continuación hay descripciones de las 3 familias. Para cada bloqueo, deberías consultar la página de manual vinculada (cuando sea posible) para obtener explicaciones más detalladas.

#### Operaciones atómicas y barreras de memoria

Las operaciones atómicas se implementan a través de un conjunto de funciones que realizan aritmética simple sobre operandos de memoria de forma atómica con respecto a eventos externos (interrupciones, apropiación, etc.). Las operaciones atómicas pueden garantizar la atomicidad solo en tipos de datos pequeños (en el orden de magnitud del tipo de datos `.long.` de arquitectura C), por lo que rara vez se debe usar directamente en el código de nivel final, sino solo para operaciones muy simples (como la configuración de flags en un mapa de bits, por ejemplo). De hecho, es bastante simple y común escribir una semántica incorrecta basada solo en operaciones atómicas (generalmente llamadas "sin bloqueo"). El kernel de FreeBSD ofrece una forma de realizar operaciones atómicas junto con una barrera de memoria. Las barreras de memoria garantizarán que ocurra una operación atómica siguiendo un orden específico con respecto a otros accesos a la memoria. Por ejemplo, si necesitamos que ocurra una operación atómica justo después de que se completen todas las demás escrituras pendientes (en términos de instrucciones que reordenan las actividades de buffer), necesitamos usar explícitamente una barrera de memoria junto con esta operación atómica. Por lo tanto, es sencillo entender por qué las barreras de memoria juegan un papel clave para la construcción de bloqueos de alto nivel (como refcounts, mutexes, etc.). Para obtener una explicación detallada sobre las operaciones atómicas, consulte atomic(9). Sin embargo, se está lejos de señalar que las operaciones atómicas (y las barreras de memoria también) deberían idealmente usarse solo para construir bloqueos frontales (como mutex).

#### Contadores de referencias

Los refcounts son interfaces para manejar contadores de referencia. Se implementan a través de operaciones atómicas y están destinadas a usarse solo en casos donde el contador de referencia es lo único que debe protegerse, por lo que incluso algo como un spin-mutex está en desuso. El uso de la interfaz refcount para estructuras, donde ya se usa un mutex, a menudo es incorrecto, ya que probablemente deberíamos cerrar el contador de referencia en algunas rutas ya protegidas. Actualmente no existe una página de manual que discuta refcount, solo lee sys/refcount.h para

obtener una descripción general de la API existente.

**Locks**

El kernel de FreeBSD tiene muchas clases de bloqueos. Cada bloqueo está definido por algunas propiedades peculiares, pero probablemente la más importante es el evento vinculado a los elementos que compiten (o en otros términos, el comportamiento de los hilos que no pueden adquirir el bloqueo). El esquema de bloqueo de FreeBSD presenta tres comportamientos diferentes para los contendientes:

1. iterando
2. bloqueo
3. dormir



los números no son casuales

**Spin locks**

Los spinlocks permiten a los que esperan iterar indefinidamente hasta que no pueden adquirir el lock. Un asunto importante que tratar es cuando un hilo compite en un spinlock si no se desplanifica su ejecución. Dado que el kernel de FreeBSD es preventivo, esto expone el spinlock al riesgo de interbloqueos que pueden resolverse simplemente deshabilitando las interrupciones mientras se adquieren. Por esta y otras razones (como la falta de soporte de propagación de prioridad, deficiencias en los esquemas de equilibrio de carga entre las CPU, etc.), los spinlocks están destinados a proteger rutas de código muy pequeñas o, idealmente, no deben usarse en absoluto si no se solicitan explícitamente (explicado más adelante).

**Bloqueo**

Los locks de bloques permiten que los que esperan sean desprogramados y bloqueados hasta que el propietario del lock no lo libere y despierte a uno o más contendientes. Para evitar problemas de inanición, los locks de bloque propagan la prioridad de los que esperan al propietario. Los locks de bloque deben implementarse a través de la interfaz turnstile y están destinados a ser el tipo de bloqueo más utilizado en el núcleo, si no se cumplen condiciones particulares.

**Dormir**

Los sleep lock permiten a los que esperan ser desplanificados y ponerse a dormir hasta que el elemento que tiene el lock no lo libere y despierte a uno o más de los elementos dormidos. Puesto que los sleep locks están pensados para proteger grandes rutas de código y de abastecer eventos asíncronos, no hacen ningún tipo de propagación de prioridad. Se deben implementar mediante la interfaz sleepqueue(9).

El orden utilizado para adquirir locks es muy importante, no solo por la posibilidad de interbloqueo debido a las inversiones de orden de lock, sino incluso porque la adquisición de locks debe seguir reglas específicas vinculadas a la naturaleza de los locks. Si echas un vistazo a la tabla de arriba, la regla práctica es que si un hilo tiene un lock de nivel n (donde el nivel es el número listado cerca del tipo de lock) no está permitido adquirir un lock de niveles superiores , ya que esto

rompería la semántica especificada para una ruta. Por ejemplo, si un hilo tiene un lock de bloque (nivel 2), se le permite adquirir un spin lock (nivel 1) pero no un sleep lock (nivel 3), ya que los locks de bloque están destinados a proteger rutas más pequeñas que el bloqueo de suspensión (sin embargo, estas reglas no se refieren a operaciones atómicas o barreras de programación).

Esta es una lista de bloqueo con sus respectivos comportamientos:

- spin mutex - iterativo - mutex(9)
- sleep mutex - bloqueante - mutex(9)
- pool mutex - bloqueante - mtx(pool)
- sleep family - suspendido - sleep(9) pause tsleep msleep msleep spin msleep rw msleep sx
- condvar - suspendido - condvar(9)
- rwlock - bloqueante - rwlock(9)
- sxlock - suspendido - sx(9)
- lockmgr - bloqueante - lockmgr(9)
- semaphores - bloqueante - sema(9)

Entre estos bloqueos, solo los mutex, sxlocks, rwlocks y lockmgrs están pensados para manejar recursividad, pero actualmente la recursividad solo es compatible con mutexes y lockmgrs.

#### Barreras de programación

Las barreras de programación están destinadas a utilizarse para impulsar la programación multihilo. Consisten principalmente en tres elementos diferentes:

- secciones críticas (y preemptividad)
- sched_bind
- sched_pin

Normalmente, estos sólo se deberían utilizar en un contexto particular e incluso aunque muchas veces pueden reemplazar a los locks, se deberían evitar porque no permiten el diagnóstico de problemas simples con las herramientas de depuración de locking (como witness(4)).

#### Secciones críticas

El kernel de FreeBSD se ha hecho preemptivo básicamente para tratar con hilos de interrupción. De hecho, para evitar una latencia de interrupción alta, los hilos de tiempo compartido con prioridad pueden ser reemplazados por hilos de interrupción (de esta manera, no necesitan esperar para ser programados como vistas previas de la ruta normal). Un kernel preemptivo, sin embargo, introduce nuevos puntos de carrera que también deben manejarse. A menudo, para hacer frente a la preemptividad, lo más sencillo es desactivarla por completo. Una sección crítica define un fragmento de código (delimitado por el par de funciones critical_enter(9) y critical_exit(9), donde se garantiza que la preemptividad no ocurrirá hasta que el código protegido se ejecute por completo). Esto a menudo puede reemplazar un lock de manera efectiva, pero debe usarse con cuidado para no perder toda la ventaja que brinda la preemptividad.

#### sched_pin/sched_unpin

Otra forma de lidiar con la preemptividad es la interfaz `sched_pin()`. Si un fragmento de código está cerrado en el par de funciones `sched_pin()` y `sched_unpin()`, se garantiza que el hilo respectivo, incluso si puede ser reemplazado, siempre se ejecutará en la misma CPU. La fijación (pinning) es muy eficaz en el caso particular en que tenemos que acceder a datos por CPU y asumimos que otros hilos no cambiarán esos datos. La última condición determinará una sección crítica como una condición demasiado fuerte para nuestro código.

#### sched_bind/sched_unbind

`sched_bind` es una API que se utiliza para vincular un hilo a una CPU en particular durante todo el tiempo que ejecuta el código, hasta que no lo desvincula la llamada a la función `sched_unbind`. Esta función tiene un papel clave en situaciones en las que no puedes confiar en el estado actual de las CPU (por ejemplo, en las primeras etapas del arranque), ya que deseas evitar que tu hilo migre a CPUs inactivas. Como `sched_bin` y `sched_unbind` manipulan las estructuras internas del planificador, es necesario que estén dentro de la adquisición/liberación `sched_lock` cuando se usan.

### 3.2.2. Estructura de proceso

Varias capas de emulación a veces requieren algunos datos adicionales por proceso. Puede administrar estructuras separadas (una lista, un árbol, etc.) que contienen estos datos para cada proceso, pero esto tiende a ser lento y consume memoria. Para solucionar este problema la estructura `proc` de FreeBSD contiene `p_emuldata`, que es un puntero vacío a algunos datos específicos de la capa de emulación. La entrada a este `proc` está protegida por el mutex proc.

La estructura `proc` de FreeBSD contiene una entrada `p_sysent` que identifica qué ABI está ejecutando este proceso. De hecho, es un puntero al `sysentvec` descrito arriba. Entonces, comparando este punto con la dirección donde se almacena la estructura `sysentvec` para la ABI dada podemos determinar si el proceso corresponde a nuestra capa de emulación. El código típicamente se parece a esto:

```
if (__predict_true(p->p_sysent != &elf_Linux(R)_sysvec))
      return;
```

Como puedes ver, utilizamos el modificador `__predict_true` para colapsar el caso más común (proceso de FreeBSD) a una simple operación de retorno preservando así un alto rendimiento. Este código debería convertirse en una macro porque actualmente no es muy flexible, es decir no soportamos emulación Linux®64 o procesos Linux® A.OUT en i386.

### 3.2.3. VFS

El subsistema VFS de FreeBSD es muy complejo pero la capa de emulación de Linux® sólo usa una pequeña parte mediante una API bien definida. Puede operar con vnodes o con manejadores de ficheros. Vnode representa un nodo virtual, es decir es la representación de un nodo en VFS. Otra representación es un manejador de fichero que representa un fichero abierto desde la perspectiva de un proceso. Un manejador de fichero puede representar un socket o un fichero ordinario. Un manejador de fichero contiene un puntero a su vnode. Varios manejadores de fichero pueden

apuntar al mismo vnode.

#### namei

La rutina namei(9) es el punto central de entrada para la búsqueda de rutas y su traducción. Recorre la ruta punto por punto desde el comienzo hasta el fin utilizando una función de búsqueda que es interna a VFS. La llamada al sistema namei(9) puede manejar enlaces simbólicos y rutas absolutas y relativas. Cuando se busca una ruta con namei(9) se introduce en la caché de nombres. Este comportamiento se puede eliminar. Esta rutina se usa en todo el kernel y su rendimiento es altamente crítico.

#### vn_fullpath

La función vn_fullpath(9) hace todo lo posible por recorrerse la caché de nombres de VFS y devolver la ruta para un vnode (bloqueado) dado. Este proceso no es fiable pero funciona bien para los casos más comunes. Esta falta de fiabilidad se produce porque depende de la caché de VFS (no recorre las estructuras del medio en cuestión), no funciona con enlaces duros, etc. Esta rutina se usa en varios sitios en el Linuxulator.

#### Operaciones de vnode

- `fgetvp` - dado un hilo y un número de descriptor de fichero devuelve el vnode asociado
- vn_lock(9) - bloquea un vnode
- `vn_unlock` - desbloquea un vnode
- VOP_READDIR(9) - lee un directorio referenciado por un vnode
- VOP_GETATTR(9) - obtiene los atributos de un fichero o directorio referenciados por un vnode
- VOP_LOOKUP(9) - busca una ruta a un directorio dado
- VOP_OPEN(9) - abre un fichero referenciado por un vnode
- VOP_CLOSE(9) - cierra un fichero referenciado por un vnode
- vput(9) - decrementa al contador de uso de un vnode y lo desbloquea
- vrele(9) - decrementa el contador de uso de un vnode
- vref(9) - incrementa el contador de uso de un vnode

#### Operaciones del controlador de archivos

- `fget` - dado un hilo y un número de descriptor de fichero devuelve el manejador de fichero asociado y lo referencia
- `fdrop` - elimina una referencia al menejador de fichero
- `fhold` - referencia un manejador de fichero

# 4. Parte MD de la capa de emulación de Linux®

Esta sección trata de la implementación de la capa de emulación Linux® en el sistema operativo FreeBSD. Primero describe la parte que depende de la arquitectura hablando sobre cómo y dónde se implementa la interacción entre el kernel y el espacio de usuario. Habla acerca de llamadas al sistema, señales, ptrace, traps, arreglos de la pila. Esta parte trata sobre i386 pero está escrita de forma general de forma que otras arquitecturas no deberían ser muy diferentes. La siguiente parte es la parte del Linuxulator independiente de la arquitectura. Esta sección sólo cubre el manejo de i386 y ELF. A.OUT está obsoleto y sin probar.

## 4.1. Manejo de llamadas al sistema

El manejo de llamadas al sistema está escrito principalmente en linux_sysvec.c, el cual cubre la mayoría de las rutinas indicadas en la estructura `sysentvec`. Cuando un proceso Linux® que se ejecuta en FreeBSD realiza una llamada al sistema, la rutina general de llamadas al sistema llama a la rutina linux prepsyscall para el ABI de Linux®.

### 4.1.1. Linux® prepsyscall

Linux® pasa los argumentos a las llamadas al sistema mediante registros (por eso está limitado a 6 parámetros en i386) mientras que FreeBSD utiliza la pila. La rutina prepsyscall de Linux® debe copiar los parámetros desde los registros a la pila. El orden de los registros es: `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, `%ebp`. El truco es que esto es verdad sólo para la *mayoría* de las llamadas al sistema. Algunas (principalmente `clone`) utiliza un orden distinto pero se puede arreglar fácilmente introduciendo un parámetro dummy en el prototipo de `linux_clone`.

### 4.1.2. Escritura de syscall

Cada llamada al sistema implementada en el Linuxulator debe tener su prototipo con varios flags en syscalls.master. La forma del archivo es:

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

La primera columna representa el número de llamada al sistema. La segunda columna es para proporcionar auditoría. La tercera columna representa el tipo de llamada al sistema. Es una de `STD`, `OBSOL`, `NOPROTO` o `UNIMPL`. `STD` es una llamada al sistema estándar con un prototipo e implementación completas. `OBSOL` es una llamada obsoleta que define sólo el prototipo. `NOPROTO` significa que la llamada al sistema está implementada en otro sitio así que no hay que añadir el prefijo del ABI, etc. `UNIMPL` significa que la llamada al sistema será sustituida por la llamada `nosys` (una llamada al sistema que tan sólo muestra un mensaje diciendo que la llamada no está implementada y que

devuelve ENOSYS).

A partir de syscalls.master un script genera tres ficheros: linux_syscall.h, linux_proto.h y linux_sysent.c. linux_syscall.h contiene las definiciones de los nombres de las llamadas al sistema y sus valores numéricos, ejemplo:

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

linux_proto.h contiene definiciones de estructuras de argumentos de todas las llamadas al sistema, ejemplo:

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

Y finalmente, linux_sysent.c contiene una estructura que describe la tabla de entrada del sistema, utilizada para enviar una llamada al sistema, por ejemplo:

```
{ 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 */
```

Como puedes ver `linux_fork` se implementa en el propio Linuxulator de modo que la definición de su tipo es `STD` y no tiene argumentos lo que se ve por la estructura de argumentos dummy. Por otro lado `close` es sólo un alias para la llamada close(2) real de FreeBSD de forma que no tiene una estructura de argumentos de linux asociada y en la tabla de entrada al sistema no tiene un prefijo "linux" ya que llama a la función close(2) real del kernel.

### 4.1.3. Llamadas al sistema ficticias

La capa de emulación de Linux® no es completa ya que algunas llamadas al sistema no están implementadas de forma adecuada y otras no están implementadas en absoluto. La capa de emulación utiliza un método para marcar las llamadas al sistema no implementadas con la macro `DUMMY`. Estas definiciones dummy se encuentran en linux_dummy.c en la forma `DUMMY(syscall)`, que luego se traduce a varios ficheros auxiliares de llamadas al sistema y cuya implementación consiste en imprimir un mensaje diciendo que la llamada no está implementada. El prototipo `UNIMPL` no se utiliza porque queremos ser capaces de identificar el nombre de la llamada al sistema que fue invocada con el fin de saber qué llamadas al sistema son importantes de implementar.

## 4.2. Manejo de señales

El manejo de señales se hace normalmente en el kernel de FreeBSD para todas las compatibilidades binarias con una llamada a la capa compat-dependiente. La capa de compatibilidad Linux® define

la rutina `linux_sendsig` con este propósito.

### 4.2.1. Linux® sendsig

Esta rutina comprueba primero si la señal se ha instalado con un `SA_SIGINFO` en cuyo caso llama en su lugar a la rutina `linux_rt_sendsig`. Además, asigna (o reutiliza uno existente) un contexto de manejador de señal ya existente, luego crea una lista de argumentos para el manejador de señal. Traduce el número de señal basado en la tabla de traducción de señales, asigna un manejador, traduce sigset. Luego guarda contexto para la rutina `sigreturn` (varios registros, número de trap traducido y máscara de señal). Finalmente, copia el contexto de la señal al espacio de usuario y prepara el contexto para que se ejecute el manejador de señal real.

### 4.2.2. linux_rt_sendsig

Esta rutina es similar a `linux_sendsig`, sólo es diferente la preparación del contexto de la señal. Añade `siginfo`, `ucontext` y algunas partes POSIX®. Podría ser interesante considerar si esas dos funciones podrían fusionarse en una sola con el beneficio de una menor duplicación de código y una posible ejecución de código más rápida.

### 4.2.3. linux_sigreturn

Esta llamada al sistema se utiliza para la devolución desde controlador de señales. Realiza algunas comprobaciones de seguridad y restaura el contexto del proceso original. También desenmascara la señal en la máscara de señal de proceso.

## 4.3. Ptrace

Muchos derivados de UNIX® implementan la llamada al sistema ptrace(2) para proporcionar diversas características de depuración y traza. Estas características permiten la traza de un proceso para obtener información valiosa acerca del proceso que es trazado, como volcado de registros, cualquier posición de memoria del espacio de direcciones del proceso, etc. y también para trazar procesos para saltar una instrucción o entre entradas al sistema (llamadas al sistema y traps). ptrace(2) también te permite establecer información en los procesos trazados (registros, etc). ptrace(2) es un estándar ampliamente disponible e implementado en la mayoría de UNIX® en todo el mundo.

La emulación de Linux® en FreeBSD implementa las características de ptrace(2) en linux_ptrace.c. Las rutinas para convertir registros entre Linux® y FreeBSD y la llamada al systema real de la emulación de ptrace(2). La llamada al sistema es un gran bloque switch que implementa su parte contraria en FreeBSD para cada comando de ptrace(2). Los comandos de ptrace(2) son mayoritariamente iguales entre Linux® y FreeBSD de forma que normalmente sólo se necesita una pequeña modificación. Por ejemplo `PT_GETREGS` en Linux® opera sobre datos directamente mientras que en FreeBSD utiliza un puntero a los datos de forma que después de ejecutar una llamada a ptrace(2) nativo, se debe hacer un copyout para preservar la semántica de Linux®.

La implementación de ptrace(2) en el Linuxulator tiene algunas debilidades. Ha habido algunos "panics" cuando se ha usado `strace` (que consume ptrace(2)) en el entorno del Linuxulator. Tampoco se ha implementado `PT_SYSCALL`.

## 4.4. Trampas

En cualquier momento en el que un proceso Linux® está ejecutándose en un trap de la capa de emulación la propia trap en sí misma es manejada de forma transparente con excepción de la traducción del trap. Linux® y FreeBSD tienen opiniones diferentes sobre lo que es un trap y cómo manejarlas. El código es normalmente muy corto:

```
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. Reparación de pila

El editor de enlaces en tiempo de ejecución de (RTLD) espera las llamadas etiquetas AUX en la pila durante una llamada a `execve` por lo que se debe realizar una reparación para garantizar esto. Por supuesto, cada sistema RTLD es diferente, por lo que la capa de emulación debe proporcionar su propia rutina de reparación de la pila para hacer esto. Linuxulator también. `elf_linux_fixup` simplemente copia las etiquetas AUX a la pila y ajusta la pila del proceso de espacio de usuario para que apunte justo después de esas etiquetas. Entonces RTLD funciona de manera inteligente.

## 4.6. soporte A.OUT

La capa de emulación Linux® en i386 también soporta binarios A.OUT de Linux®. Básicamente todo lo descrito en las secciones anteriores se tiene que implementar para el soporte de A.OUT (además de traducción de traps y envío de señales). El soporte de binarios A.OUT ya no se mantiene, en concreto la emulación de 2.6 ya no trabaja con ello pero esto no causa ningún problema ya que linux-base en ports probablemente no soporta en absoluto los binarios A.OUT. Es probable que se quite el soporte en el futuro. La mayoría de lo necesario para cargar binarios A.OUT de Linux® está en el fichero imgact_linux.c.

# 5. Parte MI de la capa de emulación Linux®

Esta sección trata acerca de la parte del Linuxulator que es independiente de la arquitectura. Cubre la infraestructura de emulación necesaria para Linux® 2.6, la implementación en i386 del almacenamiento local para hilos (TLS) y futexes. Después hablamos brevemente acerca de algunas llamadas al sistema.

## 5.1. Descripción de NPTL

Una de las áreas de mayor progreso en el desarrollo de Linux® 2.6 fue el multihilo. Antes de 2.6, el soporte de multihilo de Linux® estaba implementado en la librería linuxthreads. La librería era una implementación parcial de hilos POSIX®. El sistema de hilos se implementó utilizando procesos separados para cada hilo utilizando la llamada al sistema `clone` para dejarles compartir el espacio de direcciones (y otras cosas). La principal debilidad de esta aproximación era que cada hilo tenía un PID diferente, el envío de señales estaba roto (desde la perspectiva de pthreads), etc. Tampoco el rendimiento era muy bueno (uso de señales `SIGUSR` para sincronización de hilos, consumo de recursos del kernel, etc.) de forma que para solucionar estos problemas se desarrolló un nuevo sistema de hilos que se llamó NPTL.

La librería NPTL se centraba en dos cosas pero una tercera surgió de forma que se considera parte de NPTL. Esas dos cosas eran introducir hilos en la estructura de un proceso y los futexes. La tercera cosa adicional fue TLS, que no es necesaria directamente para NPTL pero toda la librería NPTL en espacio de usuario depende de ello. Todas estas mejoras resultaron en mucho mejor rendimiento y adhesión a los estándares. NPTL es a día de hoy una librería de hilos estándar en los sistemas Linux®.

La implementación del Linuxulator de FreeBSD se aproxima a la NTPL en tres áreas principales. TLS, futexes y renombrado de PID que se utiliza para simular hilos de Linux®. Secciones posteriores describen cada una de estas áreas.

## 5.2. Infraestructura de emulación de Linux® 2.6

Estas secciones tratan con la forma en la que se gestionan los hilos de Linux® y cómo lo simulamos en FreeBSD.

### 5.2.1. Determinación del entorno de ejecución de la emulación 2.6

La capa de emulación de Linux® en FreeBSD soporta la configuración del entorno de ejecución de la versión emulada. Esto se hace vía sysctl(8), en concreto `compat.linux.osrelease`. Establecer esta sysctl(8) afecta al comportamiento del entorno de ejecución de la capa de emulación. Cuando se establece a 2.6.x se establece el valor de `linux_use_linux26` mientras que si se establece a otra cosa no se pone nada. Esta variable (más las variables correspondientes del mismo tipo por cada jail) determinan qué infraestructura 2.6 (principalmente PID mangling) se usa o no en el código. El establecimiento de la versión se realiza en todo el sistema y afecta a todos los procesos Linux®. sysctl(8) no se debería cambiar cuando un binario Linux® se está ejecutando ya que podría romper algo.

### 5.2.2. Procesos Linux® e identificadores de hilos

Las semánticas de los hilos en Linux® son un poco confusas y utilizan una nomenclatura completamente diferente a la utilizada en FreeBSD. Un proceso en Linux® consiste en una `struct task` que contiene dos campos identificadores PID y TGID. PID *no* es el ID del proceso sino el ID del hilo. El TGID identifica a un grupo de hilos o en otras palabras, a un proceso. Para procesos monohilo el PID es igual al TGID.

El hilo en NPTL es tan sólo un proceso ordinario que resulta que tiene un TGID que no es igual al PID y que tiene un líder de grupo que no es él mismo (y VM compartida etc. por supuesto). Todo lo demás sucede de la misma forma que en un proceso ordinario. No hay separación entre un estado compartido y una estructura externa como en FreeBSD. Esto crea algo de información duplicada y una posible inconsistencia de datos. El kernel de Linux® aparentemente utiliza la información de task→group en algunos sitios y la información de la tarea en otros sitios y no es muy consistente y es propensa a errores.

Cada hilo NPTL se crea mediante una llamada a la llamada al sistema `clone` con un conjunto específico de flags (más en la siguiente subsección). La librería NPTL implementa un mecanismo de hilos estricto 1:1.

En FreeBSD emulamos hilos NPTL con procesos FreeBSD ordinarios que comparten espacio VM, etc. y la gimnasia que se hace con el PID simplemente se imita en la estructura específica de emulación adjunta al proceso. La estructura adjunta al proceso se ve así:

```
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 */
};
```

El PID se utiliza para identificar el proceso de FreeBSD que contiene esta estructura. Los campos `child_se_tid` y `child_clear_tid` se usan para hacer un copyout de la dirección del TID cuando un proceso sale y es creado. El puntero `shared` apunta a una estructura compartida entre los hilos. La variable `pdeath_signal` identifica la señal de morir del padre y el punto `threads` se utiliza para enlazar esta estructura a la lista de hilos. La estructura `linux_emuldata_shared` tiene este aspecto:

```
struct linux_emuldata_shared {

  int refs;

  pid_t group_pid;
```

```
  LIST_HEAD(, linux_emuldata) threads; /* head of list of linux threads */
};
```

`refs` es un contador de referencias que se usa para determinar cuándo liberar la estructura para evitar pérdidas de memoria. `group_id` se usa para identificar el PID (=TGID) de todo el proceso (=grupo de hilos). El puntero `threads` es la cabecera de la lista de hilos en el proceso.

La estructura `linux_emuldata` se puede obtener a partir del proceso utilizando `em_find`. El prototipo de la función es:

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

Aquí, `proc` es el proceso del cual queremos la estructura emuldata y el parámetro locked determina si queremos o no bloquear. Los valores aceptados son `EMUL_DOLOCK` y `EMUL_DOUNLOCK`. Más acerca de esto después.

### 5.2.3. Ajuste de PID

Puesto que hay una diferencia en la visión en cuanto a la idea de ID de proceso e ID de hilo entre FreeBSD y Linux® tenemos que traducir esa visión de algún modo. Lo hacemos modificando el PID. Esto significa que falseamos lo que son el PID (=TGID) y el TID (=PID) entre el kernel y el espacio de usuario. La regla básica es que en el kernel (en el Linuxulator) PID = PID y TGID = shared → group pid y que en espacio de usuario presentamos `PID = shared → group_pid` y `TID = proc → p_pid`. El miembro PID de la estructura `linux_emuldata` es un PID de FreeBSD.

Lo descrito arriba afecta principalmente a las llamadas al sistema getpid, getppid y gettid. Donde utilizamos PID/TGID respectivamente. Al hacer el copyout de los TID en `child_clear_tid` y `child_set_tid` copiamos hacia afuera el PID de FreeBSD.

### 5.2.4. Llamada al sistema clone

La llamada al sistema `clone` es la forma en la que se crean hilos en Linux®. El prototipo de la llamada es como este:

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

El parámetro `flags` le dice a la llamada al sistema cómo se tiene que clonar el proceso exactamente. Como se ha descrito arriba, Linux® puede crear procesos compartiendo varias cosas de forma independiente, por ejemplo dos procesos pueden compartir descriptores de ficheros pero no VM, etc. El último byte del parámetro `flags` es la señal de salida del proceso recién creado. El parámetro `stack` si no es `NULL` indica dónde está la pila del hilo y si es `NULL` se supone que debemos hacer un copy-on-write de la pila del proceso que llama (es decir hacer lo que hace la rutina fork(2) normal). El parámetro `parent_tidptr` se usa como dirección para copiar hacia afuera el PID del proceso (es decir, el id del hilo) una vez que el proceso está suficientemente instanciado pero todavía no es

ejecutable. El parámetro `dummy` está aquí por la convención de llamada tan extraña que tiene esta llamada al sistema en i386. Usa los registros directamente y deja que lo haga el compilador por lo que se necesita una llamada al sistema dummy. El parámetro `child_tidptr` se usa como dirección para copiar hacia afuera el PID una vez que el proceso ha terminado de crearse y cuando el proceso sale.

La llamada al sistema en sí procede estableciendo los flags correspondientes dependiendo de los flags que se le hayan pasado. Por ejemplo, `CLONE_VM` se corresponde con RFMEM (compartir VM), etc. El único detalle aquí son `CLONE_FS` y `CLONE_FILES` porque FreeBSD no permite establecerlos por separado por lo que lo falseamos al no establecer RFFDG (la copia de la tabla de descriptores de fichero y otra información de sistemas de ficheros) si alguno de los dos está definido. Esto no causa problemas porque esos dos flags siempre se establecen juntos. Después de establecer los flags el proceso se bifurca utilizando la rutina interna `fork1`, se insta a que el proceso no sea puesto en una cola de ejecución, es decir no se establece como ejecutable. Después de terminar el bifurcado posiblemente establezcamos el padre al nuevo proceso creado para emular la semántica de `CLONE_PARENT`. La siguiente parte es crear los datos de emulación. Los hilos en Linux® no señalizan a sus padres de forma que establecemos la señal exit a 0 para desabilitar esto. Después se establecen `child_set_tid` y `child_clear_tid` activando esta funcionalidad posteriormente en el código. En este punto copiamos el PID hacia afuera en la dirección especificada por `parent_tidptr`. La configuración de la pila del proceso se realiza simplemente reescribiendo el registro de marco de hilo `%esp` (`%rsp` en amd64). La siguiente parte es configurar TLS para el proceso recién creado. Después de esto ya se pueden emular las semánticas de vfork(2) y finalmente el proceso creado se pone en una cola de ejecución y se copia su PID en el proceso padre mediante el valor de retorno de `clone`.

La llamada al sistema `clone` es capaz y de hecho se usa para emular las llamadas al sistema clásicas fork(2) y vfork(2). Versiones nuevas de glibc funcionando con kernels 2.6 usan `clone` para implementar las llamadas a fork(2) y vfork(2).

### 5.2.5. Bloqueos

El mecanismo de bloqueo se implementa por cada subsistema porque no esperamos en ellos mucha contención. Hay dos locks: `emul_lock` se usa para manipular de forma segura `linux_emuldata` y `emul_shared_lock` se usa para manipular `linux_emuldata_shared`. `emul_lcok` es un mutex con el que no se puede dormir mientras que `emul_shared_lock` es un `sx_lock` con el que se puede dormir. Debido al mecanismo de bloqueo por subsistema podemos juntar algunos locks y por eso em_find proporciona acceso sin necesidad de bloqueos.

## 5.3. TLS

Esta sección trata sobre TLS, también conocido como almacenamiento local de hilos.

### 5.3.1. Introducción al manejo de hilos

Los hilos en ciencias de la computación son entidades en un proceso que pueden ser planificadas de forma independiente al resto de hilos. Los hilos de un proceso comparten muchos datos del proceso (descriptores de fichero, etc) pero también tienen su propia pila para sus propios datos. Algunas veces hay necesidad para tener datos de nivel de proceso pero específicos para un

determinado hilo. Imagina el nombre de un hilo en ejecución o algo así. El API de hilos tradicional de UNIX®, pthreads proporciona un método para hacerlo mediante pthread_key_create(3), pthread_setspecific(3) y pthread_getspecific(3) donde un hilo puede crear una clave para el dato local del hilo y manipular ese dato mediante pthread_getspecific(3) o pthread_getspecific(3). Se definió una nueva palabra clave que especifica que una variable es específica de un hilo. Puedes ver que esta no es la forma más cómoda de conseguir este objetivo. De forma que varios creadores de compiladores de C/C++ introdujeron un mecanismo mejor. También se desarrolló un nuevo método para acceder a dichas variables (al menos en i386). El método de pthreads se suele implementar en espacio de usuario como una tabla de búsqueda trivial. El rendimiento de esta solución no es muy bueno. El nuevo método utiliza registros de segmento (en i386) para direccionar un segmento donde se almacena el área TLS de forma que el acceso real a la variable del hilo consisten en añadir el registro del segmento a la dirección y acceder mediante ella. Los registros de segmento son normalmente `%gs` y `%fs` y actúan como selectores de segmentos. Cada hilo tiene su propia área donde se almacenan lo datos locales al hilo y el segmento se tiene que cargar en cada cambio de contexto. Este método es muy rápido y se utiliza casi en exclusiva en el mundo i386 de UNIX®. Tanto FreeBSD como Linux® implementan esta aproximación y se obtienen muy buenos resultados. El único problema es la necesidad de recargar el segmento en cada cambio de contexto lo que puede hacer los cambios de contexto más lentos. FreeBSD intenta evitar esta sobrecargar utilizando sólo 1 descriptor de segmento para esto mientras que Linux® utiliza 3. Algo interesante es que prácticamente nada utiliza más de 1 descriptor (sólo Wine parece utilizar 2) de forma que Linux® para un precio innecesario por los cambios de contexto.

### 5.3.2. Segmentos en i386

La arquitectura i386 implementa los llamados segmentos. Un segmento es una descripción de un área de memoria. La dirección base (abajo) del área de memoria, el final (techo), tipo, protección, etc. Se puede acceder a la memoria descrita por un segmento utilizando un registro de selección de segmento (`%cs`, `%ds`, `%ss`, `%es`, `%fs`, `%gs`). Por ejemplo supongamos que tenemos un segmento cuya dirección base es 0x1234 y también tenemos su longitud y este código:

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

Esto cargará el contenido del registro `%edx` en la ubicación de memoria 0x1244. Algunos registros de segmento tienen un uso especial, por ejemplo `%cs` se utiliza para el segmento de código y `%ss` se utiliza para el segmento de pila pero `%fs` y `%gs` generalmente no se utilizan. Los segmentos se almacenan en una tabla GDT global o en una tabla LDT local. Se accede a LDT a través de una entrada en el GDT. El LDT puede almacenar más tipos de segmentos. LDT puede ser por proceso. Ambas tablas definen hasta 8191 entradas.

### 5.3.3. Implementación en Linux® i386

Hay dos formas principales de establecer TLS en Linux®. Se puede establecer cuando se clona un proceso con la llamada al sistema `clone` o se puede llamar a `set_thread_area`. Cuando un proceso para el flag `CLONE_SETTLS` a `clone`, el kernel espera que la memoria apuntada por el registro `%esi` sea una representación en espacio de usuario de un segmento Linux® que se traduce a la representación máquina de un segmento y se carga en una entrada de la GDT. La entrada de la GDT se puede especificar con un número o se puede usar -1 que significa que el sistema puede escoger la

primera entrada que encuentre libre. En la práctica, la gran mayoría de programas utilizan sólo una entrada TLS y no se preocupan acerca del número de la misma. Aprovechamos esto en la emulación y de hecho dependemos de ello.

### 5.3.4. Emulación del TLS de Linux®

#### i386

La carga del TLS del hilo actual se realiza llamando a `set_thread_area` mientras que la carga del TLS para un segundo proceso en `clone` se realiza en el bloque separado en `clone`. Estas dos funciones son muy parecidas. La única diferencia es la carga del segmento GDT que sucede en el siguiente cambio de contexto para el nuevo proceso creado mientras que `set_thread_area` tiene que cargarlos directamente. El código básicamente hace esto. Copia la forma Linux® del descriptor de segmento desde el espacio de usuario. El código comprueba el número del descriptor pero como difieren entre FreeBSD y Linux® lo falseamos un poco. Sólo soportamos los índices 6, 3 y -1. El 6 es un número genuino de Linux®, el tres es genuino de FreeBSD y el -1 significa autoselección. Después establecemos el número del descriptor de forma constante a 3 y lo copiamos de vuelva a espacio de usuario. Dependemos de que el proceso en espacio de usuario use el número del descriptor pero esto funciona casi siempre (no he visto nunca un caso donde no funciones) ya que el proceso de espacio de usuario normalmente pasa -1. Después convertimos el descriptor de la forma Linux® a una forma dependiente de la máquina (es decir forma independiente del sistema operativo) y lo copiamos al descriptor de segmento definido en FreeBSD. Finalmente podemos cargarlo. Asignamos el descriptor en los PCB (bloque de control de proceso) de los hilos y cargamos el segmento `%gs` utilizando `load_gs`. Esta carga se tiene que hacer dentro de una sección crítica de forma que nada la interrumpa. El caso `CLONE_SETTLS` funciona exactamente así salvo que no se realiza la carga utilizando `load_gs`. El segmento que se usa para esto (número de segmento 3) se comparte para este uso entre los procesos de FreeBSD y de Linux® de forma que la capa de emulación Linux® no añade nada de sobrecarga respecto al funcionamiento normal de FreeBSD.

#### amd64

La implementación de amd64 es similar a la de i386, pero inicialmente no se utilizó un descriptor de segmento de 32 bits para este propósito (por lo tanto, ni siquiera los usuarios nativos de TLS de 32 bits funcionaban), por lo que tuvimos que agregar dicho segmento e implementar su carga en cada cambio de contexto (cuando se establece el flag de uso de 32 bits). Aparte de esto, la carga de TLS es exactamente la misma, solo que los números de segmento son diferentes y el formato del descriptor y la carga difieren ligeramente.

## 5.4. Futexes

### 5.4.1. Introducción a la sincronización

Los hilos necesitan algún tipo de sincronización y POSIX® proporciona algunos de ellos: mutex para exclusión mutua, locks de lectura y escritura para exclusión mutua con una proporción sesgada de lecturas y escrituras y variables de condición para señalar un cambio de estado. Es interesante notar que la API de hilos de POSIX® carece de soporte para semáforos. Esas implementaciones de rutinas de sincronización dependen en gran medida del tipo de soporte de hilos que tenemos. En el modelo puro 1:M (espacio de usuario), la implementación se puede

realizar únicamente en el espacio de usuario y, por lo tanto, es muy rápida (las variables de condición probablemente terminarán implementándose mediante señales, es decir, no tan rápido) y simple. En el modelo 1:1, la situación también es bastante clara: los hilos deben sincronizarse utilizando las primitivas del kernel (lo cual es muy lento porque se debe realizar una llamada al sistema). El escenario mixto M:N simplemente combina el primer y segundo enfoque o se basa únicamente en el kernel. La sincronización de hilos es una parte vital de la programación habilitada para hilos y su rendimiento puede afectar mucho al programa resultante. Pruebas de rendimiento recientes en el sistema operativo FreeBSD mostraron que una implementación mejorada de sx_lock producía un 40% de aceleración en *ZFS* (un usuario intensivo de sx), esto es algo dentro del kernel pero muestra claramente cuán importante es el rendimiento de las primitivas de sincronización.

Los programas multihilo se deberían escribir con la menor contención posible. De otro modo en lugar de hacer trabajo útil el hilo simplemente espera en un bloqueo. Como resultado los programas mejores escritos muestran poca contención en bloqueos.

### 5.4.2. Introducción a los futexes

Linux® implementa multihilo 1:1, es decir tiene que utilizar primitivas de sincronización dentro del kernel. Como se ha dicho antes, un programa bien escrito tiene poca contención. Así que una secuencia típica se podría realizar como dos incrementos/decrementos de contadores de referencia mutex atómicos, lo que es muy rápido, como se muestra en el siguiente ejemplo:

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

El modelo 1:1 nos obliga a realizar dos llamadas al sistema para esas llamadas mutex, lo cual es muy lento.

La solución que implementa Linux® 2.6 se llama futexes. Los futexes implementan la comprobación de la contención en espacio de usuario y llaman al kernel sólo en caso de contención. Por lo tanto el caso típico tiene lugar sin intervención del kernel. Esto ofrece una implementación de primitivas de sincronización razonablemente rápidas y flexibles.

### 5.4.3. Futex API

La llamada al sistema futex se ve así:

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

En este ejemplo `uaddr` es una dirección del mutex en espacio de usuario, `op` es una operación que estamos a punto de realizar y los otros parámetros tienen significados por operación.

Los Futexes implementan las siguientes operaciones:

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

**FUTEX_WAIT**

Esta operación verifica que se ha escrito el valor `val` en la dirección `uaddr`. Si no, se devuelve `EWOULDBLOCK`, de otro modo el hilo se encola en el futex y se suspende. Si el argumento `timeout` no es cero entonces especifica el tiempo máximo para estar durmiendo, de lo contrario se duerme indefinidamente.

**FUTEX_WAKE**

Esta operación toma un futex en la dirección `uaddr` y despierta los primeros `val` futexes encolados en el futex.

**FUTEX_FD**

Esta operación asocia un descriptor de archivo con un futex dado.

**FUTEX_REQUEUE**

Esta operación toma `val` hilos encolados en el futex que está en la dirección `uaddr`, los despierta y toma los siguientes `val2` hilos y los reencola en el futex en la dirección `uaddr2`.

**FUTEX_CMP_REQUEUE**

Esta operación hace lo mismo que `FUTEX_REQUEUE` pero primero comprueba que `val3` sea igual que `val`.

**FUTEX_WAKE_OP**

Esta operación realiza una operación atómica en `val3` (que contiene otro valor codificado) y `uaddr`. Después despierta `val` hilos en el futex de la dirección `uaddr` y si la operación atómica devolvió un número positivo despierta `val2` hilos en el futex de la dirección `uaddr2`.

Las operaciones implementadas en `FUTEX_WAKE_OP`:

- FUTEX_OP_SET
- FUTEX_OP_ADD
- FUTEX_OP_OR
- FUTEX_OP_AND
- FUTEX_OP_XOR



No hay parámetro `val2` en el prototipo de futex. `val2` se toma del parámetro `struct timespec *timeout` para las operaciones `FUTEX_REQUEUE`, `FUTEX_CMP_REQUEUE` y `FUTEX_WAKE_OP`.

### 5.4.4. Emulación Futex en FreeBSD

La emulación de futex en FreeBSD ha sido importada de NetBSD y después extendida por nosotros. Se encuentra en los ficheros `linux_futex.c` y linux_futex.h. La estructura `futex` tiene este aspecto:

```
struct futex {
  void *f_uaddr;
  int f_refcount;

  LIST_ENTRY(futex) f_list;

  TAILQ_HEAD(lf_waiting_paroc, waiting_proc) f_waiting_proc;
};
```

Y la estructura `waiting_proc` es:

```
struct waiting_proc {

  struct thread *wp_t;

  struct futex *wp_new_futex;

  TAILQ_ENTRY(waiting_proc) wp_list;
};
```

#### futex_get / futex_put

Un futex se obtiene utilizando la función `futex_get`, que busca en una lista lineal de futexes y devuelve el encontrado o crea un nuevo futex. Cuando liberamos un futex llamamos a la función `futex_put`, que disminuye un contador de referencia del futex y si el refcount llega a cero lo libera.

#### futex_sleep

Cuando un futex encola un hilo para que duerma crea una estructura `working_proc` y la pone en la lista dentro de la estructura del futext, después simplemente llama a tsleep(9) para suspender el hilo. El tiempo de suspensión puede finalizar por timeout. Después de volver the tsleep(9) (el hilo ha sido despertado o ha ocurrido un timeout) se quita la estructura `working_proc` de la lista y se destruye. Todo esto se hace en la función `futex_sleep`. Si se nos despertó con `futex_wak` tenemos `wp_new_futex` establecido de forma que lo utilizamos para dormir. De este modo el reencolado en realidad se hace en esta función.

#### futex_wake

Despertar a un hilo que está durmiendo en un futex se hace con la función `futex_wake`. En esta función lo primero que hacemos es imitar el extraño comportamiento de Linux®, donde despierta N hilos para todas las operaciones, la únca excepción es que las operaciones REQUEUE se hacen en N+1 hilos. Pero normalmente esto no supone ninguna diferencia ya que estamos despertando todos los hilos. Lo siguiente en la función es el bucle en el que despertamos n hilos, después comprobamos si hay algún futex nuevo para reencolar. Si es así, reencolamos un máximo de n2 hilos en el nuevo futex. Esto coopera con `futex_sleep`.

#### futex_wake_op

La operación `FUTEX_WAKE_OP` is bastante complicada. Primero obtenemos dos futex en las direcciones `uaddr` y `uaddr2` después realizamos una operación atómica usando `val3` y `uaddr2`. Después se despierta a `val` hilos que estuvieran durmiendo y si se cumple la condición de la operación atómica despertamos `val2` (es decir `timeout`) hilos durmientes en el segundo futex.

#### operación atómica futex

La operación atómica toma dos parámetros `encoded_op` y `uaddr`. La operación codificada codifica la operación en sí, comparando valor, argumento de operación y argumento de comparación. El pseudocódigo para la operación es como este:

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

Y esto se hace automáticamente. Primero se realiza la copia del número en `uaddr` y la operación ha terminado. El código maneja fallos de página y si no ocurre ningún se compara `oldval` con `cmparg` con el comparador cmp.

#### Bloqueo futex

La implementación de futex utiliza dos listas de bloqueo que protegen `sx_lock` y locks globales (ya sea Giant u otro `sx_lock`). Cada operación se realiza estando bloqueada desde el principio hasta el final.

## 5.5. Implementación de varias llamadas al sistema

En esta sección voy a describir algunas llamadas al sistema más pequeñas que vale la pena mencionar porque su implementación no es obvia o esas llamadas al sistema son interesantes desde otro punto de vista.

### 5.5.1. Familia de llamadas al sistema *at

Durante el desarrollo del kernel 2.6.16 de Linux® se añadieron las llamadas al sistema *at. Esas llamadas (`openat` por ejemplo) funcionan igual que sus pares sin `at` con la pequeña diferencia del parámetro `dirfd`. Este parámetro cambia con el fichero dado sobre el que se va a realizar la llamada al sistema. Cuando el parámetro `filename` es absoluto `dirfd` es ignorado pero cuando la ruta al

fichero es relativa, entra en juego. El parámetro `dirfd` es un directorio relativo al cual se comprueba la ruta relativa. El parámetro `dirfd` es un descriptor de fichero de algún directorio o `AT_FDCWD`. Por ejemplo la llamada al sistema `openat` podría ser así:

```
descriptor de fichero 123 = /tmp/foo/, directorio de trabajo actual = /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
*/
```

Esta infraestructura es necesaria para evitar condiciones de carrera cuando se abren ficheros fuera del directorio de trabajo actual. Imagina un proceso que consiste en dos hilos, hilo A e hilo B. El hilo A realiza `open(./tmp/foo/bah., flags, mode)` y antes de volver es desalojado y se ejecuta el hilo B. El hilo B no se preocupa por las necesidades del hilo A y renombra o elimina /tmp/foo/. Tenemos una condición de carrera. Para evitar esto podemos abrir /tmp/foo y utilizarlo como `dirfd` en la llamada al sistema `openat`. Esto permite al usuario implementar directorios de trabajo por hilo.

La familia *at de llamadas al sistema de Linux® contiene: `linux_openat`, `linux_mkdirat`, `linux_mknodat`, `linux_fchownat`, `linux_futimesat`, `linux_fstatat64`, `linux_unlinkat`, `linux_renameat`, `linux_linkat`, `linux_symlinkat`, `linux_readlinkat`, `linux_fchmodat` y `linux_faccessat`. Todas se implementan utilizando la rutina modificada nami(9) y una sencilla capa de envoltorio.

**Implementación**

La implementación se hace modificando la rutina namei(9) (descrita arriba) para que tenga un parámetro adicional `dirfd` en su estructura `nameidata`, que especifica el punto de comienzo de la búsqueda de la ruta en lugar de utilizar el directorio de trabajo cada vez. La resolución de `dirfd` a vnode a partir del número de descriptor de fichero se hace en las llamadas al sistema *at nativas. Cuando `dirfd` es `AT_FDCWD` la entrada `dvp` en la estructura `nameidata` es `NULL` pero cuando `dirfd` otro número obtenemos el fichero para este descriptor de fichero, comprobamos si el fichero es válido y si tiene un vnode asociado lo obtenemos. Después comprobamos que el vnode sea un directorio. En la rutina namei(9) real simplemente sustituimos el vnode `dvp` por la variable `dp` en la función namei(9) que determina el punto de comienzo. namei(9) no se usa directamente sino mediante una traza de diferentes funciones a diferentes niveles. Por ejemplo `openat` hace esto:

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

Por esta razón `kern_open` y `vn_open` deben modificarse para incorporar el parámetro adicional `dirfd`. No se crea una capa de compatibilidad para aquellos porque no hay muchos usuarios de esta y los usuarios se pueden convertir fácilmente. Esta implementación general permite a FreeBSD implementar su propio *at llamadas al sistema. Esto está siendo discutido ahora mismo.

### 5.5.2. Ioctl

La interfaz ioctl es bastante frágil debido a su genericidad. Tenemos que tener en cuenta que los

dispositivos difieren entre Linux® y FreeBSD, por lo que se debe tener cuidado para que la emulación de ioctl funcione correctamente. El manejo de ioctl se implementa en `linux_ioctl.c`, donde se define la función `linux_ioctl`. Esta función simplemente itera sobre conjuntos de manejadores ioctl para encontrar un manejador que implemente un comando dado. La llamada al sistema ioctl tiene tres parámetros, el descriptor de archivo, el comando y un argumento. El comando es un número de 16 bits, que en teoría se divide en 8 bits altos que determinan la clase del comando ioctl y 8 bits bajos, que son el comando real dentro del conjunto dado. La emulación aprovecha esta división. Implementamos controladores para cada conjunto, como `sound_handler` o `disk_handler`.Cada controlador tiene un comando máximo y un comando mínimo definido, que se utiliza para determinar qué controlador se utiliza. Hay leves problemas con este enfoque porque Linux® no usa la división de conjuntos de manera consistente, por lo que a veces los ioctls de un conjunto diferente están dentro de un conjunto al que no deberían pertenecer (ioctls genéricos SCSI dentro del conjunto cdrom, etc.). FreeBSD actualmente no implementa muchos ioctls de Linux® (en comparación con NetBSD, por ejemplo) pero el plan es portarlos de NetBSD. La tendencia es usar ioctls de Linux® incluso en los controladores nativos de FreeBSD debido a la fácil migración de las aplicaciones.

### 5.5.3. Depuración

Cada llamada al sistema debería ser depurable. Para ello introducimos una pequeña infraestructura. Tenemos la función ldebug, que indica si una llamada al sistema determinada debe depurarse (configurable mediante un sysctl). Para imprimir tenemos macros LMSG y ARGS. Se utilizan para alterar una cadena imprimible para mensajes de depuración uniformes.

# 6. Conclusión

## 6.1. Resultados

A fecha de abril de 2007 la capa de emulación de Linux® es capaz de emular el kernel Linux® 2.6.16 bastante bien. Los problemas que quedan son sobre futexes, la familia de llamadas al sistema *at sin terminar, problemas con el envío de señales, la ausencia de `epoll` y `inotify` y probablemente algunos bugs que no se han descubierto todavía. A pesar de esto somos capaces de ejecutar básicamente todos los programas Linux® incluidos en la colección de ports con Fedora Core 4 en 2.6.16 y hay algunos informes rudimentarios de éxito con Fedora Core 6 en 2.6.16. El linux_base de Fedora Core 6 se añadió al repositorio recientemente permitiendo más pruebas de la capa de emulación y dándonos más pistas sobre dónde debemos poner el esfuerzo para implementar las cosas que faltan.

Somos capaces de ejecutar las aplicaciones más usadas como www/linux-firefox, net-im/skype y algunos juegos de la colección de ports. Algunos programas tienen un mal comportamiento bajo la emulación de 2.6 pero se está investigando y con suerte se solucionará pronto. La única aplicación grande que se sabe que no funciona es el Java™ Development Kit de Linux®. Esto es porque requiere `epoll` el cual no está directamente relacionado con el kernel Linux® 2.6.

Esperamos habilitar la emulación 2.6.16 por defecto algún tiempo después del lanzamiento de FreeBSD 7.0 al menos para exponer las partes de la emulación 2.6 para pruebas más amplias. Una vez hecho esto, podemos cambiar a Fedora Core 6 linux_base, que es el plan definitivo.

## 6.2. Trabajo futuro

El trabajo futuro debe centrarse en solucionar los problemas restantes con futexes, implementar el resto de la familia de llamadas al sistema *at, arreglar el envío de señales y posiblemente implementar `epoll` y `inotify`.

Esperamos poder ejecutar pronto los programas más importantes sin problemas, por lo que podremos cambiar a la emulación 2.6 por defecto y hacer que Fedora Core 6 sea la linux_base predeterminada porque nuestro Fedora Core 4 que usamos actualmente ya no es compatible.

El otro objetivo posible es compartir nuestro código con NetBSD y DragonflyBSD. NetBSD tiene algo de soporte para la emulación 2.6 pero está lejos de estar terminado y no se ha probado realmente. DragonflyBSD ha expresado cierto interés en portar las mejoras 2.6.

En general, conforme se desarrolla Linux® nos gustaría seguir actualizados con su desarrollo, implementando las nuevas llamadas al sistema. Splice se me viene a la cabeza. Algunas de las llamadas al sistema ya implementadas son subóbtimas, por ejemplo `mremap` y otras. Se pueden hacer algunas mejoras de rendimiento, bloqueos más finos y otras cosas.

## 6.3. Equipo

Colaboré en este proyecto con (en orden alfabético):

- 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>

Me gustaría agradecer a todas esas personas por sus consejos, revisiones de código y apoyo general.

# 7. Bibliografía

1. Marshall Kirk McKusick - George V. Neville-Neil. Diseño e implementación del sistema operativo FreeBSD. Addison-Wesley, 2005.
2. https://tldp.org
3. https://www.kernel.org