# Emulação de Linux no FreeBSD

Resumo

Essa tese master lida com a atualização da camada de emulação do Linux® (o chamado *Linuxulator*). A tarefa foi atualizar a camada para casar com a funcionalidade do Linux® 2.6. Como uma referencia a implementação, o kernel Linux® 2.6.16 foi escolhido. O conceito é perdidamente baseado na implementação do NetBSD. Maior parte do trabalho foi feito no verão de 2006 como parte de um programa de estudante do Google Summer of Code. O foco foi trazer o suporte do *NPTL* (nova biblioteca de threads POSIX®) pra dentro da camada de emulação, incluindo *TLS* (thread local storage), *futexes* (mutexes rapidos na camada de usuario), *PID mangling*, e algumas outras coisas menores. Muitos pequenos problemas foram identificados e corrigidos. Meu trabalho foi integrado dentro do repositório de principal do FreeBSD e vai ser ligado ao 7.0R release. Nós, o time de desenvolvimento de emulação estamos trabalhando na emulação do Linux® 2.6 a camada de emulação padr ão do FreeBSD.

# Índice



# 1. Introdução

Nos últimos anos, os sistemas operacionais baseados em código aberto UNIX® começaram a ser amplamente implantados em máquinas servidores e clientes. Entre esses sistemas operacionais eu gostaria de destacar dois: FreeBSD, por sua herança BSD, base de código comprovada pelo tempo e muitos recursos interessantes e Linux® por sua ampla base de usuários, entusiasta comunidade aberta de desenvolvedores e apoio de grandes empresas. O FreeBSD tende a ser usado em máquinas de classe servidor, tarefas de rede pesadas com menos uso em máquinas de classe desktop para usuários comuns. Embora o Linux® tenha o mesmo uso em servidores, mas é muito mais usado por usuários domésticos. Isto leva a uma situação onde existem muitos programas binários disponíveis apenas para Linux® que não suportam o FreeBSD.

Naturalmente, surge a necessidade da habilidade de executar binários Linux® em um sistema FreeBSD e é com isso que esta tese trata: a emulação do kernel do Linux® no sistema operacional FreeBSD.

Durante o verão de 2006, a Google Inc. patrocinou um projeto que se concentrava em estender a camada de emulação do Linux® (o chamado Linuxulator) no FreeBSD para incluir necessidades do Linux® 2.6. Esta tese é escrita como parte deste projeto.

# 2. Um olhar para dentro...

Nesta seção vamos descrever cada sistema operacional em questão. Como eles lidam com syscalls, trapframes etc., todo o material de baixo nível. Também descrevemos a maneira como eles entendem primitivas comuns UNIX®, como o que é um PID, o que é uma thread, etc. Na terceira subseção, falamos sobre como UNIX® em emuladores UNIX® pode ser feita em geral.

## 2.1. O que é o  UNIX ®

UNIX® é um sistema operacional com um longo histórico que influenciou quase todos os outros sistemas operacionais atualmente em uso. Começando na década de 1960, seu desenvolvimento continua até hoje (embora em projetos diferentes). O desenvolvimento de UNIX® logo se bifurcou em duas formas principais: as famílias BSDs e System III/V. Eles se influenciaram mutuamente ao desenvolver um padrão UNIX® comum. Entre as contribuições originadas no BSD, podemos nomear memória virtual, rede TCP/IP, FFS e muitas outras. A ramificação SystemV contribuiu para as primitivas de comunicação entre processos SysV, copy-on-write, etc. UNIX® em si não existe mais, mas suas idéias têm sido usadas por muitos outros sistemas operacionais amplos formando assim os chamados sistemas operacionais como UNIX®. Hoje em dia os mais influentes são Linux®, Solaris e possivelmente (até certo ponto) FreeBSD. Existem sistemas UNIX® de companhias derivados como (AIX, HP-UX etc.), mas estas foram cada vez mais migrados para os sistemas acima mencionados. Vamos resumir as características típicas do UNIX®.

## 2.2. Detalhes técnicos

Todo programa em execução constitui um processo que representa um estado da computação. O processo de execução é dividido entre o espaço do kernel e o espaço do usuário. Algumas operações podem ser feitas somente a partir do espaço do kernel (lidando com hardware, etc.), mas o processo deve passar a maior parte de sua vida útil no espaço do usuário. O kernel é onde o gerenciamento dos processos, hardware e detalhes de baixo nível acontecem. O kernel fornece uma API unificada padrão UNIX® para o espaço do usuário. Os mais importantes são abordados abaixo.

### 2.2.1. Comunicação entre o kernel e o processo de espaço do usuário

A API comum do UNIX® define uma syscall como uma forma de emitir comandos de um processo do espaço do usuário para o kernel. A implementação mais comum é usando uma instrução de interrupção ou especializada (pense em instruções `SYSENTER`/`SYSCALL` para ia32). Syscalls são definidos por um número. Por exemplo, no FreeBSD, a syscall número 85 é a syscall swapon(2) e a syscall número 132 é a syscall mkfifo(2). Algumas syscalls precisam de parâmetros, que são passados do espaço do usuário para o espaço do kernel de várias maneiras (dependente da implementação). Syscalls são síncronas.

Outra maneira possível de se comunicar é usando uma *trap*. As traps ocorrem de forma assíncrona após a ocorrência de algum evento (divisão por zero, falha de página, etc.). Uma trap pode ser

transparente para um processo (falha de página) ou pode resultar em uma reação como o envio de um *signal* (divisão por zero).

### 2.2.2. Comunicação entre processos

Existem outras APIs (System V IPC, memória compartilhada, etc.), mas a API mais importante é o signal. Os signals são enviados por processos ou pelo kernel e recebidos por processos. Alguns signals podem ser ignorados ou manipulados por uma rotina fornecida pelo usuário, alguns resultam em uma ação predefinida que não pode ser alterada ou ignorada.

### 2.2.3. Gerenciamento de processos

As instâncias do kernel são processadas primeiro no sistema (chamado init). Todo processo em execução pode criar sua cópia idêntica usando a syscall fork(2). Algumas versões ligeiramente modificadas desta syscall foram introduzidas, mas a semântica básica é a mesma. Todo processo em execução pode se transformar em algum outro processo usando a syscall exec(3). Algumas modificações desta syscall foram introduzidas, mas todas servem ao mesmo propósito básico. Os processos terminam suas vidas chamando a syscall exit(2). Todo processo é identificado por um número único chamado PID. Todo processo tem um processo pai definido (identificado pelo seu PID).

### 2.2.4. Gerenciamento de threads

O UNIX® tradicional não define nenhuma API nem implementação para threading, enquanto POSIX® define sua API de threading, mas a implementação é indefinida. Tradicionalmente, havia duas maneiras de implementar threads. Manipulando-as como processos separados (threading 1:1) ou envolver todo o grupo de thread em um processo e gerenciando a threading no espaço do usuário (threading 1:N). Comparando as principais características de cada abordagem:

1:1 threading

- - threads pesadas
- - o agendamento não pode ser alterado pelo usuário (ligeiramente mitigado pela API POSIX ®)
- + não necessita de envolvimento do syscall
- + pode utilizar várias CPUs

1: N threading

- + threads leves
- + agendamento pode ser facilmente alterado pelo usuário
- - syscalls devem ser acondicionadas
- - não pode utilizar mais de uma CPU

## 2.3. O que é o FreeBSD?

O projeto FreeBSD é um dos mais antigos sistemas operacionais de código aberto atualmente

disponíveis para uso diário. É um descendente direto do verdadeiro UNIX®, portanto, pode-se afirmar que ele é um verdadeiro UNIX® embora os problemas de licenciamento não permitam isso. O início do projeto remonta ao início dos anos 90, quando uma equipe de usuários BSD corrigiu o sistema operacional 386BSD. Baseado neste patchkit surgiu um novo sistema operacional, chamado FreeBSD por sua licença liberal. Outro grupo criou o sistema operacional NetBSD com diferentes objetivos em mente. Vamos nos concentrar no FreeBSD.

O FreeBSD é um sistema operacional baseado no UNIX® com todos os recursos do UNIX®. Multitarefa preemptiva, necessidades de multiusuário, rede TCP/IP, proteção de memória, suporte a multiprocessamento simétrico, memória virtual com VM mesclada e cache de buffer, todos eles estão lá. Um dos recursos interessantes e extremamente úteis é a capacidade de emular outros sistemas operacionais UNIX®-like. A partir de dezembro de 2006 e do desenvolvimento do 7-CURRENT, as seguintes funcionalidades de emulação são suportadas:

- Emulação FreeBSD/i386 no FreeBSD/amd64
- Emulação de FreeBSD/i386 no FreeBSD/ia64
- Emulação-Linux® do sistema operacional Linux ® no FreeBSD
- Emulação de NDIS da interface de drivers de rede do Windows
- Emulação de NetBSD do sistema operacional NetBSD
- Suporte PECoff para executáveis PECoff do FreeBSD
- Emulação SVR4 do UNIX® System V revisão 4

Emulações ativamente desenvolvidas são a camada Linux® e várias camadas FreeBSD-on-FreeBSD. Outros não devem funcionar corretamente nem ser utilizáveis nos dias de hoje.

## 2.3.1. Detalhes técnicos

O FreeBSD é o gostinho tradicional de UNIX® no sentido de dividir a execução dos processos em duas metades: espaço do kernel e execução do espaço do usuário. Existem dois tipos de entrada de processo no kernel: uma syscall e uma trap. Há apenas uma maneira de retornar. Nas seções subseqüentes, descreveremos as três portas de/para o kernel. Toda a descrição se aplica à arquitetura i386, pois o Linuxulator só existe lá, mas o conceito é semelhante em outras arquiteturas. A informação foi retirada de [1] e do código fonte.

### 2.3.1.1. Entradas do sistema

O FreeBSD tem uma abstração chamada loader de classes de execução, que é uma entrada na syscall execve(2). Isto emprega uma estrutura `sysentvec`, que descreve uma ABI executável. Ele contém coisas como tabela de tradução de errno, tabela de tradução de sinais, várias funções para atender às necessidades da syscall (correção de pilha, coredumping, etc.). Toda ABI que o kernel do FreeBSD deseja suportar deve definir essa estrutura, como é usado posteriormente no código de processamento da syscall e em alguns outros lugares. As entradas do sistema são tratadas pelos manipuladores de traps, onde podemos acessar o espaço do kernel e o espaço do usuário de uma só vez.

#### 2.3.1.2. Syscalls

Syscalls no FreeBSD são emitidos executando a interrupção `0x80` com o registrador `%eax` definido para um número de syscall desejado com argumentos passados na pilha.

Quando um processo emite uma interrupção `0x80`, a syscall manipuladora de trap `int0x80` é proclamada (definida em sys/i386/i386/exception.s), que prepara argumentos (ou seja, copia-os para a pilha) para uma chamada para uma função C syscall(2) (definida em sys/i386/i386/trap.c), que processa o trapframe passado. O processamento consiste em preparar a syscall (dependendo da entrada `sysvec`), determinando se a syscall é de 32 ou 64 bits (muda o tamanho dos parâmetros), então os parâmetros são copiados, incluindo a syscall. Em seguida, a função syscall real é executada com o processamento do código de retorno (casos especiais para erros `ERESTART` e `EJUSTRETURN`). Finalmente, um `userret()` é agendado, trocando o processo de volta ao ritmo do usuário. Os parâmetros para a syscall manipuladora atual são passados na forma de argumentos `struct thread *td`, `struct syscall args*` onde o segundo parâmetro é um ponteiro para o copiado na estrutura de parâmetros.

#### 2.3.1.3. Armadilhas (Traps)

O manuseio de traps no FreeBSD é similar ao manuseio de syscalls. Sempre que ocorre uma trap, um manipulador de assembler é chamado. É escolhido entre alltraps, alltraps com regs push ou calltrap, dependendo do tipo de trap. Este manipulador prepara argumentos para uma chamada para uma função C `trap()` (definida em sys/i386/i386/trap.c), que então processa a trap ocorrida. Após o processamento, ele pode enviar um sinal para o processo e/ou sair para o espaço do usuário usando `userret()`.

#### 2.3.1.4. Saídas

As saídas do kernel para o userspace acontecem usando a rotina assembler `doreti`, independentemente de o kernel ter sido acessado por meio de uma trap ou via syscall. Isso restaura o status do programa da pilha e retorna ao espaço do usuário.

#### 2.3.1.5. primitivas UNIX®

O sistema operacional FreeBSD adere ao esquema tradicional UNIX®, onde cada processo possui um número de identificação único, o chamado *PID* (ID do processo). Números PID são alocados de forma linear ou aleatória variando de `0` para `PID_MAX`. A alocação de números PID é feita usando pesquisa linear de espaço PID. Cada thread em um processo recebe o mesmo número PID como resultado da chamada getpid(2).

Atualmente existem duas maneiras de implementar o threading no FreeBSD. A primeira maneira é o threading M:N seguido pelo modelo de threading 1:1. A biblioteca padrão usada é o threading M:N (`libpthread`) e você pode alternar no tempo de execução para threading 1:1 (`libthr`). O plano é mudar para a biblioteca 1:1 por padrão em breve. Embora essas duas bibliotecas usem as mesmas primitivas do kernel, elas são acessadas por API(s) diferentes. A biblioteca M:N usa a família `kse_*` das syscalls enquanto a biblioteca 1:1 usa a família `thr_*` das syscalls. Por causa disso, não existe um conceito geral de ID de threading compartilhado entre o kernel e o espaço do usuário. Obviamente, as duas bibliotecas de threads implementam a API de ID de threading pthread. Todo threading do kernel (como descrito por `struct thread`) possui identificadores td tid, mas isso não é

diretamente acessível a partir do espaço do usuário e serve apenas as necessidades do kernel. Ele também é usado para a biblioteca de threading 1:1 como o ID de threading do pthread, mas a manipulação desta é interna à biblioteca e não pode ser confiável.

Como dito anteriormente, existem duas implementações de threads no FreeBSD. A biblioteca M:N divide o trabalho entre o espaço do kernel e o espaço do usuário. Thread é uma entidade que é agendada no kernel, mas pode representar vários números de threads do userspace. Threads M do userspace são mapeadas para threads N do kernel, economizando recursos e mantendo a capacidade de explorar o paralelismo de multiprocessadores. Mais informações sobre a implementação podem ser obtidas na página do manual ou [1]. A biblioteca 1:1 mapeia diretamente um segmento userland para uma thread do kernel, simplificando muito o esquema. Nenhum desses designs implementa um mecanismo justo (tal mecanismo foi implementado, mas foi removido recentemente porque causou séria lentidão e tornou o código mais difícil de lidar).

## 2.4. O que é Linux®

Linux® é um kernel do tipo UNIX® originalmente desenvolvido por Linus Torvalds, e agora está sendo contribuído por uma grande quantidade de programadores em todo o mundo. De seu simples começo até hoje, com amplo suporte de empresas como IBM ou Google, o Linux® está sendo associado ao seu rápido ritmo de desenvolvimento, suporte completo a hardware e seu benevolente modelo despota de organização.

O desenvolvimento do Linux® começou em 1991 como um projeto amador na Universidade de Helsinque na Finlândia. Desde então, ele obteve todos os recursos de um sistema operacional semelhante ao UNIX: multiprocessamento, suporte multiusuário, memória virtual, rede, basicamente tudo está lá. Também há recursos altamente avançados, como virtualização, etc.

A partir de 2006, o Linux parece ser o sistema operacional de código aberto mais utilizado com o apoio de fornecedores independentes de software como Oracle, RealNetworks, Adobe, etc. A maioria dos softwares comerciais distribuídos para Linux® só pode ser obtido de forma binária, portanto a recompilação para outros sistemas operacionais é impossível.

A maior parte do desenvolvimento do Linux® acontece em um sistema de controle de versão Git. O Git é um sistema distribuído, de modo que não existe uma fonte central do código Linux®, mas algumas ramificações são consideradas proeminentes e oficiais. O esquema de número de versão implementado pelo Linux® consiste em quatro números A.B.C.D. Atualmente, o desenvolvimento acontece em 2.6.C.D, onde C representa a versão principal, onde novos recursos são adicionados ou alterados, enquanto D é uma versão secundária somente para correções de bugs.

Mais informações podem ser obtidas em [3].

### 2.4.1. Detalhes técnicos

O Linux® segue o esquema tradicional do UNIX® de dividir a execução de um processo em duas metades: o kernel e o espaço do usuário. O kernel pode ser inserido de duas maneiras: via trap ou via syscall. O retorno é tratado apenas de uma maneira. A descrição mais detalhada aplica-se ao Linux® 2.6 na arquitetura i386™. Esta informação foi retirada de [2].

#### 2.4.1.1. Syscalls

Syscalls em Linux® são executados (no espaço de usuário) usando macros `syscallX` onde X substitui um número que representa o número de parâmetros da syscall dada. Essa macro traduz um código que carrega o registro `% eax` com um número da syscall e executa a interrupção `0x80`. Depois disso, um retorn da syscall é chamado, o que traduz valores de retorno negativos para valores `errno` positivos e define `res` para `-1` em caso de erro. Sempre que a interrupção `0x80` é chamada, o processo entra no kernel no manipulador de trap das syscalls. Essa rotina salva todos os registros na pilha e chama a entrada syscall selecionada. Note que a convenção de chamadas Linux® espera que os parâmetros para o syscall sejam passados pelos registradores como mostrado aqui:

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

Existem algumas exceções, onde Linux® usa diferentes convenções de chamada (mais notavelmente a syscall `clone`).

#### 2.4.1.2. Armadilhas (Traps)

Os manipuladores de traps são apresentados em arch/i386/kernel/traps.c e a maioria desses manipuladores vive em arch/i386/kernel/entry.S, onde a manipulação das traps acontecem.

#### 2.4.1.3. Saídas

O retorno da syscall é gerenciado pela syscall exit(3), que verifica se o processo não está concluído e verifica se usamos seletores fornecidos pelo usuário . Se isso acontecer, a correção da pilha é aplicada e, finalmente, os registros são restaurados da pilha e o processo retorna ao espaço do usuário.

#### 2.4.1.4. primitivas UNIX®

Na versão 2.6, o sistema operacional Linux® redefiniu algumas das primitivas tradicionais do UNIX®, especialmente PID, TID e thread. O PID é definido para não ser exclusivo para cada processo, portanto, para alguns processos (threading) getppid(2) retorna o mesmo valor. A identificação exclusiva do processo é fornecida pelo TID. Isso ocorre porque o *NPTL* (Nova Biblioteca de threading POSIX®) define threading para serem processos normais (assim chamado threading 1:1). Gerar um novo processo no Linux® 2.6 acontece usando a syscall `clone` (as variantes do fork são reimplementadas usando-o). Esta syscall clone define um conjunto de sinalizadores que afetam o comportamento do processo de clonagem em relação à implementação do threading. A semântica é um pouco confusa, pois não existe uma única bandeira dizendo a syscall para criar uma thread.

Flags de clone implementados são:

- `CLONE_VM` - os processos compartilham seu espaço de memória
- `CLONE_FS` - compartilha umask, cwd e namespace
- `CLONE_FILES` - compartilham arquivos abertos
- `CLONE_SIGHAND` - compartilha manipuladores de sinais e bloqueia sinais
- `CLONE_PARENT` - compartilha processo pai
- `CLONE_THREAD` - ser a thread (mais explicações abaixo)
- `CLONE_NEWNS` - novo namespace
- `CLONE_SYSVSEM` - compartilha SysV sob estruturas
- `CLONE_SETTLS` - configura o TLS no endereço fornecido
- `CLONE_PARENT_SETTID` - define o TID no processo pai
- `CLONE_CHILD_CLEARTID` - limpe o TID no processo filho
- `CLONE_CHILD_SETTID` - define o TID no processo filho

`CLONE_PARENT` define o processo real para o processo pai do requisitante. Isso é útil para threads porque, se a thread A criar a thread B, queremos que a thread B parenteada para o processo pai de todo o grupo de threads. `CLONE_THREAD` faz exatamente a mesma coisa que `CLONE_PARENT`, `CLONE_VM` e `CLONE_SIGHAND`, reescreve o PID para ser o mesmo que PID do requisitante, define o sinal de saída como none e entra no grupo de threads. `CLONE_SETTLS` configura entradas GDT para tratamento de TLS. O conjunto de flags `CLONE_*_*TID` define/limpa o endereço fornecido pelo usuário para TID ou 0.

Como você pode ver, o `CLONE_THREAD` faz a maior parte do trabalho e não parece se encaixar muito bem no esquema. A intenção original não é clara (mesmo para autores, de acordo com comentários no código), mas acho que originalmente havia uma flag de thread, que foi então dividida entre muitas outras flags, mas essa separação nunca foi totalmente concluída. Também não está claro para que serve esta partição, uma vez que a glibc não usa isso, portanto, apenas o uso do clone escrito à mão permite que um programador acesse esses recursos.

Para programas não segmentados, o PID e o TID são os mesmos. Para programas em threadings, os primeiros PID e TID da thread são os mesmos e todos os threading criados compartilham o mesmo PID e são atribuídos a um TID exclusivo (porque `CLONE_THREAD` é passado), o processo pai também é compartilhado para todos os processos que formam esse threading do programa.

O código que implementa pthread_create(3) no NPTL define as flags de clone como este:

```
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);
```

O `CLONE_SIGNAL` é definido como

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

o último 0 significa que nenhum sinal é enviado quando qualquer uma das threads finaliza.

## 2.5. O que é emulação

De acordo com uma definição de dicionário, emulação é a capacidade de um programa ou dispositivo de imitar um outro programa ou dispositivo. Isto é conseguido fornecendo a mesma reação a um determinado estímulo que o objeto emulado. Na prática, o mundo do software vê três tipos de emulação - um programa usado para emular uma máquina (QEMU, vários emuladores de consoles de jogos etc.), emulação de software de uma instalação de hardware (emuladores OpenGL, emulação de unidades de ponto flutuante etc.) e emulação do sistema (no kernel do sistema operacional ou como um programa de espaço do usuário).

Emulação é geralmente usada em um lugar, onde o uso do componente original não é viável nem possível a todos. Por exemplo, alguém pode querer usar um programa desenvolvido para um sistema operacional diferente do que eles usam. Então a emulação vem a calhar. Por vezes, não há outra maneira senão usar emulação - por ex. Quando o dispositivo de hardware que você tenta usar não existe (ainda/mais), então não há outro caminho além da emulação. Isso acontece com frequência ao transferir um sistema operacional para uma nova plataforma (inexistente). Às vezes é mais barato emular.

Olhando do ponto de vista da implementação, existem duas abordagens principais para a implementação da emulação. Você pode emular a coisa toda - aceitando possíveis entradas do objeto original, mantendo o estado interno e emitindo a saída correta com base no estado e/ou na entrada. Este tipo de emulação não requer condições especiais e basicamente pode ser implementado em qualquer lugar para qualquer dispositivo/programa. A desvantagem é que a implementação de tal emulação é bastante difícil, demorada e propensa a erros. Em alguns casos, podemos usar uma abordagem mais simples. Imagine que você deseja emular uma impressora que imprime da esquerda para a direita em uma impressora que imprime da direita para a esquerda. É óbvio que não há necessidade de uma camada de emulação complexa, mas a simples reversão do texto impresso é suficiente. Às vezes, o ambiente de emulação é muito semelhante ao emulado, portanto, apenas uma camada fina de alguma tradução é necessária para fornecer uma emulação totalmente funcional! Como você pode ver, isso é muito menos exigente de implementar, portanto, menos demorado e propenso a erros do que a abordagem anterior. Mas a condição necessária é que os dois ambientes sejam semelhantes o suficiente. A terceira abordagem combina os dois anteriores. Na maioria das vezes, os objetos não fornecem os mesmos recursos, portanto, em um caso de emulação, o mais poderoso é o menos poderoso que temos para emular os recursos ausentes com a emulação completa descrita acima.

Esta tese de mestrado lida com a emulação de UNIX® em UNIX®, que é exatamente o caso, onde apenas uma camada fina de tradução é suficiente para fornecer emulação completa. A API do UNIX® consiste em um conjunto de syscalls, que geralmente são autônomas e não afetam algum

estado global do kernel.

Existem algumas syscalls que afetam o estado interno, mas isso pode ser resolvido fornecendo algumas estruturas que mantêm o estado extra.

Nenhuma emulação é perfeita e as emulações tendem a não ter algumas partes, mas isso geralmente não causa nenhuma desvantagem séria. Imagine um emulador de console de jogos que emula tudo, menos a saída de música. Não há dúvida de que os jogos são jogáveis e pode-se usar o emulador. Pode não ser tão confortável quanto o console original, mas é um compromisso aceitável entre preço e conforto.

O mesmo acontece com a API do UNIX®. A maioria dos programas pode viver com um conjunto muito limitado de syscalls funcionando. Essas syscalls tendem a ser as mais antigas (read(2) /write(2), fork(2) family, signal(3) handling, exit(3), socket(2) API), portanto, é fácil emular porque sua semântica é compartilhada entre todos os UNIX®, que existem hoje.

# 3. Emulação

## 3.1. Como funciona a emulação no FreeBSD

Como dito anteriormente, o FreeBSD suporta a execução de binários a partir de vários outros UNIX®. Isso funciona porque o FreeBSD tem uma abstração chamada loader de classes de execução. Isso se encaixa na syscall execve(2), então quando execve(2) está prestes a executar um binário que examina seu tipo.

Existem basicamente dois tipos de binários no FreeBSD. Scripts de texto semelhantes a shell que são identificados por `#!` como seus dois primeiros caracteres e binários normais (normalmente *ELF*), que são uma representação de um objeto executável compilado. A grande maioria (pode-se dizer todos eles) de binários no FreeBSD é do tipo ELF. Os arquivos ELF contêm um cabeçalho, que especifica a ABI do OS para este arquivo ELF. Ao ler essas informações, o sistema operacional pode determinar com precisão o tipo de binário do arquivo fornecido.

Toda ABI de OS deve ser registrada no kernel do FreeBSD. Isso também se aplica ao sistema operacional nativo do FreeBSD. Então, quando execve(2) executa um binário, ele itera através da lista de APIs registradas e quando ele encontra a correta, ele começa a usar as informações contidas na descrição da ABI do OS (sua tabela syscall, tabela de tradução `errno`, etc.). Assim, toda vez que o processo chama uma syscall, ele usa seu próprio conjunto de syscalls em vez de uma global. Isso efetivamente fornece uma maneira muito elegante e fácil de suportar a execução de vários formatos binários.

A natureza da emulação de diferentes sistemas operacionais (e também alguns outros subsistemas) levou os desenvolvedores a invitar um mecanismo de evento manipulador. Existem vários locais no kernel, onde uma lista de manipuladores de eventos é chamada. Cada subsistema pode registrar um manipulador de eventos e eles são chamados de acordo com sua necessidade. Por exemplo, quando um processo é encerrado, há um manipulador chamado que possivelmente limpa o que o subsistema que ele precisa de limpeza.

Essas facilidades simples fornecem basicamente tudo o que é necessário para a infra-estrutura de

emulação e, de fato, essas são basicamente as únicas coisas necessárias para implementar a camada de emulação do Linux®.

# 3.2. Primitivas comuns no kernel do FreeBSD

Camadas de emulação precisam de algum suporte do sistema operacional. Eu vou descrever algumas das primitivas suportadas no sistema operacional FreeBSD.

## 3.2.1. Primitivas de Bloqueio

Contribuído por: Attilio Rao attilio@FreeBSD.org

O conjunto de primitivas de sincronização do FreeBSD é baseado na idéia de fornecer um grande número de diferentes primitivas de uma maneira que a melhor possa ser usada para cada situação específica e apropriada.

Para um ponto de vista de alto nível, você pode considerar três tipos de primitivas de sincronização no kernel do FreeBSD:

- operações atômicas e barreiras de memória
- locks
- barreiras de agendamento

Abaixo, há descrições para as 3 famílias. Para cada bloqueio, você deve verificar a página de manual vinculada (onde for possível) para obter explicações mais detalhadas.

### 3.2.1.1. Operações atômicas e barreiras de memória

Operações atômicas são implementadas através de um conjunto de funções que executam aritmética simples em operandos de memória de maneira atômica com relação a eventos externos (interrupções, preempção, etc.). Operações atômicas podem garantir atomicidade apenas em pequenos tipos de dados (na ordem de magnitude do tipo de dados C da arquitetura `.long.`), portanto raramente devem ser usados diretamente no código de nível final, se não apenas para operações muito simples (como configuração de flags em um bitmap, por exemplo). De fato, é bastante simples e comum escrever uma semântica errada baseada apenas em operações atômicas (geralmente referidas como lock-less). O kernel do FreeBSD oferece uma maneira de realizar operações atômicas em conjunto com uma barreira de memória. As barreiras de memória garantirão que uma operação atômica ocorrerá seguindo alguma ordem especificas em relação a outros acessos à memória. Por exemplo, se precisarmos que uma operação atômica aconteça logo depois que todas as outras gravações pendentes (em termos de instruções reordenando atividades de buffers) forem concluídas, precisamos usar explicitamente uma barreira de memória em conjunto com essa operação atômica. Portanto, é simples entender por que as barreiras de memória desempenham um papel fundamental na construção de bloqueios de alto nível (assim como referências, exclusões mútuas, etc.). Para uma explicação detalhada sobre operações atômicas, consulte atomic(9). É muito, no entanto, notar que as operações atômicas (e as barreiras de memória também) devem, idealmente, ser usadas apenas para construir bloqueios front-ending (como mutexes).

#### 3.2.1.2. Refcounts

Refcounts são interfaces para manipular contadores de referência. Eles são implementados por meio de operações atômicas e destinam-se a ser usados apenas para casos em que o contador de referência é a única coisa a ser protegida, portanto, até mesmo algo como um spin-mutex é obsoleto. Usar a interface de recontagem para estruturas, onde um mutex já é usado, geralmente está errado, pois provavelmente devemos fechar o contador de referência em alguns caminhos já protegidos. Uma manpage discutindo refcount não existe atualmente, apenas verifique sys/refcount.h para uma visão geral da API existente.

#### 3.2.1.3. Locks

O kernel do FreeBSD tem enormes classes de bloqueios. Cada bloqueio é definido por algumas propriedades peculiares, mas provavelmente o mais importante é o evento vinculado a detentores de contestação (ou, em outros termos, o comportamento de threading incapazes de adquirir o bloqueio). O esquema de bloqueio do FreeBSD apresenta três comportamentos diferentes para contendores:

1. spinning
2. blocking
3. sleeping

números não são casuais

#### 3.2.1.4. Spinning locks

Spin locks permitem que os acumuladores rotacionarem até que eles não consigam adquirir um lock. Uma questão importante é quando um segmento contesta em um spin lock se não for desmarcado. Uma vez que o kernel do FreeBSD é preventivo, isto expõe o spin lock ao risco de deadlocks que podem ser resolvidos apenas desabilitando as interrupções enquanto elas são adquiridas. Por essa e outras razões (como falta de suporte à propagação de prioridade, falta de esquemas de balanceamento de carga entre CPUs, etc.), os spin locks têm a finalidade de proteger endereçamentos muito pequenos de código ou, idealmente, não serem usados se não solicitados explicitamente ( explicado posteriormente).

#### 3.2.1.5. Bloqueio

Os locks em blocos permitem que as tarefas dos acumuladores sejam removidas e bloqueados até que o proprietário do bloqueio não os libere e ative um ou mais contendores. Para evitar problemas de fome, os locks em bloco fazem a propagação de prioridade dos acumuladores para o proprietário. Os locks em bloco devem ser implementados por meio da interface turnstile e devem ser o tipo mais usado de bloqueios no kernel, se nenhuma condição específica for atendida.

#### 3.2.1.6. Sleeping

Sleep locks permitem que as tarefas dos waiters sejam removidas e eles adormecem até que o suporte do lock não os deixe cair e desperte um ou mais waiters. Como os sleep locks se destinam a proteger grandes endereçamentos de código e a atender a eventos assíncronos, eles não fazem nenhuma forma de propagação de prioridade. Eles devem ser implementados por meio da

interface sleepqueue(9).

A ordem usada para adquirir locks é muito importante, não apenas pela possibilidade de deadlock devido a reversões de ordem de bloqueio, mas também porque a aquisição de lock deve seguir regras específicas vinculadas a naturezas de bloqueios. Se você der uma olhada na tabela acima, a regra prática é que, se um segmento contiver um lock de nível n (onde o nível é o número listado próximo ao tipo de bloqueio), não é permitido adquirir um lock de níveis superiores , pois isso quebraria a semântica especificada para um caminho. Por exemplo, se uma thread contiver um lock em bloco (nível 2), ele poderá adquirir um spin lock (nível 1), mas não um sleep lock (nível 3), pois os locks em bloco são destinados a proteger caminhos menores que o sleep lock ( essas regras não são sobre operações atômicas ou agendamento de barreiras, no entanto).

Esta é uma lista de bloqueio com seus respectivos comportamentos:

- spin mutex - spinning - mutex(9)
- sleep mutex - blocking - mutex(9)
- pool mutex - blocking - mtx_pool(9)
- família sleep - sleeping - sleep(9) pausa tsleep msleep msleep spin msleep rw msleep sx
- condvar - sleeping - condvar(9)
- wlock - blocking - rwlock(9)
- sxlock - sleeping - sx(9)
- lockmgr - sleeping - lockmgr(9)
- semáforos - sleeping - sema(9)

Entre esses bloqueios, apenas mutexes, sxlocks, rwlocks e lockmgrs são destinados a tratar recursão, mas atualmente a recursão é suportada apenas por mutexes e lockmgrs.

#### 3.2.1.7. Barreiras de agendamento

As barreiras de agendamento devem ser usadas para orientar o agendamento de threads. Eles consistem principalmente de três diferentes stubs:

- seções críticas (e preempção)
- sched_bind
- sched_pin

Geralmente, eles devem ser usados apenas em um contexto específico e, mesmo que possam substituir bloqueios, eles devem ser evitados porque eles não permitem o diagnóstico de problemas simples com ferramentas de depuração de bloqueio (como witness(4)).

#### 3.2.1.8. Seções críticas

O kernel do FreeBSD foi feito basicamente para lidar com threads de interrupção. De fato, para evitar latência de interrupção alta, os segmentos de prioridade de compartilhamento de tempo podem ser precedidos por threads de interrupção (dessa forma, eles não precisam aguardar para serem agendados como as visualizações de caminho normais). Preempção, no entanto, introduz

novos pontos de corrida que precisam ser manipulados também. Muitas vezes, para lidar com a preempção, a coisa mais simples a fazer é desativá-la completamente. Uma seção crítica define um pedaço de código (delimitado pelo par de funções critical_enter(9) e critical_exit(9), onde é garantido que a preempção não aconteça (até que o código protegido seja totalmente executado) Isso pode substituir um bloqueio efetivamente, mas deve ser usado com cuidado para não perder toda a vantagem essa preempção traz.

#### 3.2.1.9. sched_pin/sched_unpin

Outra maneira de lidar com a preempção é a interface `sched_pin()`. Se um trecho de código é fechado no par de funções `sched_pin()` e `sched_unpin()`, é garantido que a respectiva thread, mesmo que possa ser antecipada, sempre ser executada na mesma CPU. Fixar é muito eficaz no caso particular quando temos que acessar por dados do cpu e assumimos que outras threads não irão alterar esses dados. A última condição determinará uma seção crítica como uma condição muito forte para o nosso código.

#### 3.2.1.10. sched_bind/sched_unbind

`sched_bind` é uma API usada para vincular uma thread a uma CPU em particular durante todo o tempo em que ele executa o código, até que uma chamada de função `sched_unbind` não a desvincule. Esse recurso tem um papel importante em situações em que você não pode confiar no estado atual das CPUs (por exemplo, em estágios iniciais de inicialização), já que você deseja evitar que sua thread migre em CPUs inativas. Como `sched_bind` e `sched_unbind` manipulam as estruturas internas do agendador, elas precisam estar entre a aquisição/liberação de `sched_lock` quando usadas.

### 3.2.2. Estrutura Proc

Várias camadas de emulação exigem alguns dados adicionais por processo. Ele pode gerenciar estruturas separadas (uma lista, uma árvore etc.) contendo esses dados para cada processo, mas isso tende a ser lento e consumir memória. Para resolver este problema, a estrutura `proc` do FreeBSD contém `p_emuldata`, que é um ponteiro vazio para alguns dados específicos da camada de emulação. Esta entrada `proc` é protegida pelo mutex proc.

A estrutura `proc` do FreeBSD contém uma entrada `p_sysent` que identifica, qual ABI este processo está executando. Na verdade, é um ponteiro para o `sysentvec` descrito acima. Portanto, comparando esse ponteiro com o endereço em que a estrutura `sysentvec` da ABI especificada está armazenada, podemos efetivamente determinar se o processo pertence à nossa camada de emulação. O código normalmente se parece com:

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

Como você pode ver, usamos efetivamente o modificador `__predict_true` para recolher o caso mais comum (processo do FreeBSD) para uma operação de retorno simples, preservando assim o alto desempenho. Este código deve ser transformado em uma macro porque atualmente não é muito flexível, ou seja, não suportamos emulação Linux®64 nem processa A.OUT Linux® em i386.

### 3.2.3. VFS

O subsistema FreeBSD VFS é muito complexo, mas a camada de emulação Linux® usa apenas um pequeno subconjunto através de uma API bem definida. Ele pode operar em vnodes ou manipuladores de arquivos. Vnode representa um vnode virtual, isto é, representação de um nó no VFS. Outra representação é um manipulador de arquivos, que representa um arquivo aberto da perspectiva de um processo. Um manipulador de arquivos pode representar um socket ou um arquivo comum. Um manipulador de arquivos contém um ponteiro para seu vnode. Mais de um manipulador de arquivos pode apontar para o mesmo vnode.

#### 3.2.3.1. namei

A rotina namei(9) é um ponto de entrada central para a pesquisa e o nome do caminho. Ele percorre o caminho ponto a ponto do ponto inicial até o ponto final usando a função de pesquisa, que é interna ao VFS. A syscall namei(9)pode lidar com links simbólicos, absolutos e relativos. Quando um caminho é procurado usando namei(9) ele é inserido no cache de nomes. Esse comportamento pode ser suprimido. Essa rotina é usada em todo o kernel e seu desempenho é muito crítico.

#### 3.2.3.2. vn_fullpath

A função vn_fullpath(9) faz o melhor esforço para percorrer o cache de nomes do VFS e retorna um caminho para um determinado vnode (bloqueado). Esse processo não é confiável, mas funciona bem nos casos mais comuns. A falta de confiabilidade é porque ela depende do cache do VFS (ele não atravessa as estruturas intermediárias), não funciona com hardlinks, etc. Essa rotina é usada em vários locais no Linuxulator.

#### 3.2.3.3. Operações de vnode

- `fgetvp` - dado um encadeamento e um número de descritor de arquivo, ele retorna o vnode associado
- vn_lock(9) - bloqueia um vnode
- `vn_unlock` - desbloqueia um vnode
- VOP_READDIR(9) - lê um diretório referenciado por um vnode
- VOP_GETATTR(9) - obtém atributos de um arquivo ou diretório referenciado por um vnode
- VOP_LOOKUP(9) - procura um caminho para um determinado diretório
- VOP_OPEN(9) - abre um arquivo referenciado por um vnode
- VOP_CLOSE(9) - fecha um arquivo referenciado por um vnode
- vput(9) - decrementa a contagem de uso para um vnode e o desbloqueia
- vrele(9) - diminui a contagem de uso para um vnode
- vref(9) - incrementa a contagem de uso para um vnode

#### 3.2.3.4. Operações do manipulador de arquivos

- `fget` - dado uma thread e um número de file descriptor, ele retorna o manipulador de arquivos

associado e faz referência a ele

- `fdrop` - elimina uma referência a um manipulador de arquivos
- `fhold` - faz referência a um manipulador de arquivos

# 4. Parte da camada de emulação -MD do Linux®

Esta seção trata da implementação da camada de emulação do Linux® no sistema operacional FreeBSD. Ele primeiro descreve a parte dependente da máquina falando sobre como e onde a interação entre o usuário e o kernel é implementada. Ele fala sobre syscalls, signals, ptrace, traps, correção de pilha. Esta parte discute o i386, mas ele é escrito geralmente para que outras arquiteturas não sejam muito diferentes. A próxima parte é a parte independente da máquina do Linuxulator. Esta seção abrange apenas o tratamento de i386 e ELF. A.OUT está obsoleto e não foi testado.

## 4.1. Manipulação de Syscall

A manipulação de Syscall é principalmente escrita em linux_sysvec.c, que cobre a maioria das rotinas apontadas na estrutura `sysentvec`. Quando um processo Linux® executado no FreeBSD emite um syscall, a rotina syscall geral chama a rotina prepsyscall do linux para a ABI do Linux®.

### 4.1.1. Linux® prepsyscall

Linux® passa argumentos via registradores de syscalls (isso porque ele é limitado a 6 parametros no i386) enquanto o FreeBSD usa uma pilha. A rotina prepsyscall do Linux® deve copiar parametros dos registradores para a pilha. A ordem dos registradores é: `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, `%ebp`. O fato é que isso é verdadeiro apenas para *a maioria* das syscalls. Algumas (mais provavelmente `clone`) usam uma ordem diferente, mas é demasiadamente facil de arrumar inserindo um parametro dummy no prototype `linux_clone`.

### 4.1.2. Escrevendo syscall

Cada syscall implementada no Linuxulator deve ter seu protótipo com vários flags no syscalls.master. A forma do arquivo é:

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

A primeira coluna representa o número da syscall. A segunda coluna é para suporte de auditoria. A terceira coluna representa o tipo da syscall. É `STD`, `OBSOL`, `NOPROTO` e `UNIMPL`. `STD` é uma syscall padrão com protótipo e implementação completos. `OBSOL` é obsoleto e define apenas o protótipo. `NOPROTO` significa que a syscall é implementado em outro lugar, portanto, não precede o prefixo da ABI, etc.

UNIMPL significa que a syscall será substituída pela syscall nosys (uma syscall apenas imprime uma mensagem sobre a syscall não sendo implementado e retornando ENOSYS).

De um script syscalls.master, gera três arquivos: linux_syscall.h, linux_proto.h e linux_sysent.c. O linux_syscall.h contém definições de nomes de syscall e seus valores numéricos, por exemplo:

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

O linux_proto.h contém definições de estrutura de argumentos para cada syscall, por exemplo:

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

E finalmente, linux_sysent.c contém uma estrutura descrevendo a tabela de entrada do sistema, usada para realmente enviar um syscall, por exemplo:

```
{ 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 você pode ver, linux_fork é implementado no próprio Linuxulator, então a definição é do tipo STD e não possui argumento, que é exibido pela estrutura de argumento fictícia. Por outro lado, close é apenas um apelido para o verdadeiro close(2) do FreeBSD para que ele não possua estrutura de argumentos do linux associada e na tabela de entrada do sistema ele não é prefixado com linux, pois ele chama o verdadeiro close(2) no kernel.

### 4.1.3. Dummy syscalls

A camada de emulação do Linux® não está completa, pois algumas syscalls não estão implementadas corretamente e algumas não estão implementadas. A camada de emulação emprega um recurso para marcar syscalls não implementadas com a macro DUMMY. Estas definições fictícias residem em linux_dummy.c em uma forma de DUMMY(syscall);, que é então traduzido para vários arquivos auxiliares de syscall e a implementação consiste em imprimir uma mensagem dizendo que esta syscall não está implementada. O protótipo UNIMPL não é usado porque queremos ser capazes de identificar o nome da syscall que foi chamado para saber o que é mais importante implementar na syscalls.

## 4.2. Manuseio de signals

A manipulação de sinais é feita geralmente no kernel do FreeBSD para todas as compatibilidades binárias com uma chamada para uma camada dependente de compatibilidade. A camada de compatibilidade do Linux® define a rotina linux_sendsig para essa finalidade.

### 4.2.1. Linux® sendsig

Esta rotina primeiro verifica se o signal foi instalado com um `SA_SIGINFO`, caso em que chama a rotina `linux_rt_sendsig`. Além disso, ele aloca (ou reutiliza um contexto de identificador de sinal já existente) e cria uma lista de argumentos para o manipulador de signal. Ele traduz o número do signal baseado na tabela de tradução do signal, atribui um manipulador, traduz o sigset. Em seguida, ele salva o contexto para a rotina `sigreturn` (vários registradores, número da trap traduzida e máscara de signal). Finalmente, copia o contexto do signal para o espaço do usuário e prepara o contexto para que o manipulador de sinal real seja executado.

### 4.2.2. linux_rt_sendsig

Esta rotina é similar a `linux_sendsig` apenas a preparação do contexto do sinal é diferente. Adiciona `siginfo`, `ucontext` e algumas partes do POSIX®. Pode valer a pena considerar se essas duas funções não poderiam ser mescladas com um benefício de menos duplicação de código e, possivelmente, até mesmo execução mais rápida.

### 4.2.3. linux_sigreturn

Esta syscall é usada para retornar do manipulador de sinal. Ela faz algumas verificações de segurança e restaura o contexto do processo original. Também desmascara o sinal na máscara de sinal do processo.

## 4.3. Ptrace

Muitos derivados do UNIX® implementam a syscall ptrace(2) para permitir vários recursos de rastreamento e depuração . Esse recurso permite que o processo de rastreamento obtenha várias informações sobre o processo rastreado, como registros de despejos, qualquer memória do espaço de endereço do processo, etc. e também para rastrear o processo, como em uma instrução ou entre entradas do sistema (syscalls e traps). ptrace(2) também permite definir várias informações no processo de rastreamento (registros, etc.). ptrace(2) é um padrão de toda o UNIX® implementado na maioria dos UNIX®es em todo o mundo.

Emulação do Linux® no FreeBSD implementa a habilidade ptrace(2) em linux_ptrace.c. As rotinas para converter registradores entre Linux® and FreeBSD e a atual emulação de syscall, syscall ptrace(2). A syscall é um longo bloco de trocas que implementa em contraparte no FreeBSD para todo comando ptrace(2). Os comandos ptrace(2) são em sua maioria igual entre Linux® e FreeBSD então uma pequena modificação é necessária. Por exemplo, `PT_GETREGS` em Linux® opera diretamente dos dados enquanto o FreeBSD usa um ponteiro para o dado e depois performa a syscall ptrace(2) (nativa), uma cópia deve ser feita pra preservar a semantica do Linux®.

A implementação de ptrace(2) no Linuxulator tem algumas fraquezas conhecidas. Houve pânico ao usar o `strace` (que é um consumidor ptrace(2)) no ambiente Linuxulator. `PT_SYSCALL` também não está implementado.

## 4.4. Armadilhas (Traps)

Sempre que um processo Linux® executado na camada de emulação captura a própria trap, ela é

tratada de forma transparente com a única exceção da tradução de trap. Linux® e o FreeBSD difere de opinião sobre o que é uma trap, então isso é tratado aqui. O código é realmente muito curto:

```
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. Correção de pilha

O editor de links em tempo de execução do RTLD espera as chamadas tags AUX na pilha durante uma `execve`, portanto, uma correção deve ser feita para garantir isso. Naturalmente, cada sistema RTLD é diferente, portanto, a camada de emulação deve fornecer sua própria rotina de correção de pilha para fazer isso. O mesmo acontece com o Linuxulator. O `elf_linux_fixup` simplesmente copia tags AUX para a pilha e ajusta a pilha do processo de espaço do usuário para apontar logo após essas tags. Então, a RTLD funciona de maneira inteligente.

## 4.6. Suporte para A.OUT

A camada de emulação Linux® em i386 também suporta os binários Linux® A.OUT. Praticamente tudo o que foi descrito nas seções anteriores deve ser implementado para o suporte A.OUT (além da tradução de traps e o envio de sinais). O suporte para binários A.OUT não é mais mantido, especialmente a emulação 2.6 não funciona com ele, mas isso não causa nenhum problema, já que os ports linux-base provavelmente não suportam binários A.OUT. Esse suporte provavelmente será removido no futuro. A maioria das coisas necessárias para carregar os binários Linux® A.OUT está no arquivo imgact_linux.c.

# 5. Parte da camada de emulação -MI do Linux®

Esta seção fala sobre parte independente de máquina do Linuxulator. Ele cobre a infra-estrutura de emulação necessária para a emulação do Linux® 2.6, a implementação do TLS (thread local storage) (no i386) e os futexes. Então falamos brevemente sobre algumas syscalls.

## 5.1. Descrição do NPTL

Uma das principais áreas de progresso no desenvolvimento do Linux® 2.6 foi o threading. Antes do 2.6, o suporte ao threading Linux® era implementado na biblioteca linuxthreads. A biblioteca foi uma implementação parcial do threading POSIX®. A segmentação foi implementada usando processos separados para cada threading usando a syscall `clone` para permitir que eles compartilhem o espaço de endereço (e outras coisas). A principal fraqueza desta abordagem era que cada thread tinha um PID diferente, o tratamento de sinal era quebrado (da perspectiva pthreads), etc. O desempenho também não era muito bom (uso de sinais `SIGUSR` para sincronização de threads) , consumo de recursos do kernel, etc.) para superar esses problemas, um novo sistema de threading foi desenvolvido e denominado NPTL.

A biblioteca NPTL focou em duas coisas, mas uma terceira coisa apareceu, então é normalmente considerada parte do NPTL. Essas duas coisas eram a incorporação de threads em uma estrutura de processo e futexes. A terceira coisa adicional foi o TLS, que não é diretamente exigido pelo NPTL, mas toda a biblioteca de usuário do NPTL depende dele. Essas melhorias resultaram em muito melhor desempenho e conformidade com os padrões. O NPTL é uma biblioteca de threading padrão nos sistemas Linux® atualmente.

A implementação do FreeBSD Linuxulator se aproxima do NPTL em três áreas principais. O TLS, futexes e PID mangling, que serve para simular as threadings Linux®. Outras seções descrevem cada uma dessas áreas.

## 5.2. Infra-estrutura de emulação do Linux® 2.6

Estas seções tratam da maneira como as threadings Linux® são gerenciadas e como nós simulamos isso no FreeBSD.

### 5.2.1. Determinação de tempo de execução de emulação 2.6

A camada de emulação do Linux® no FreeBSD suporta a configuração de tempo de execução da versão emulada. Isso é feito via sysctl(8), a saber `compat.linux.osrelease`. A configuração dessa sysctl(8) afeta o comportamento de tempo de execução da camada de emulação. Quando definido como 2.6.x, ele configura o valor de `linux_use_linux26` enquanto a configuração para algo mais o mantém não definido. Essa variável (mais variáveis por prisão do mesmo tipo) determina se a infraestrutura 2.6 (principalmente o PID) é usada no código ou não. A configuração da versão é feita em todo o sistema e isso afeta todos os processos Linux®. A sysctl(8) não deve ser alterada ao executar qualquer binário do Linux®, pois pode causar danos .

### 5.2.2. Processos e identificadores de threading Linux®

A semântica de threading Linux® é um pouco confusa e usa uma nomenclatura inteiramente diferente do FreeBSD. Um processo em Linux® consiste em uma `struct task` incorporando dois campos identificadores - PID e TGID. O PID *não é* um ID de processo, mas é um ID de thread. O TGID identifica um grupo de threads em outras palavras, um processo. Para o processo single-threaded, o PID é igual ao TGID.

A thread no NPTL é apenas um processo comum que acontece de ter TGID diferente de PID e ter um líder de grupo diferente de si mesmo (e VM compartilhada, é claro). Tudo o mais acontece da mesma maneira que em um processo comum. Não há separação de um status compartilhado para alguma estrutura externa como no FreeBSD. Isso cria alguma duplicação de informações e possível inconsistência de dados. O kernel Linux® parece usar a tarefa → grupo de informações em alguns lugares e informações de tarefas em outros lugares e isso não é muito consistente e parece propenso a erros.

Cada threading NPTL é criada por uma chamada a syscall `clone` com um conjunto específico de flags (mais na próxima subseção). O NPTL implementa segmentação rígida de 1:1.

No FreeBSD nós emulamos threads NPTL com processos comuns do FreeBSD que compartilham espaço de VM, etc. e a ginástica PID é apenas imitada na estrutura específica de emulação anexada ao processo. A estrutura anexada ao processo se parece com:

```
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 */
};
```

O PID é usado para identificar o processo do FreeBSD que liga esta estrutura. `child_se_tid` e `child_clear_tid` são usadas para cópia do endereço TID quando um processo existe e é criado. O ponteiro `shared` aponta para uma estrutura compartilhada entre as threads. A variável `pdeath_signal` identifica o sinal de morte do processo pai e o ponteiro `threads` é usado para vincular essa estrutura à lista de threads. A estrutura `linux_emuldata_shared` se parece com:

```
struct linux_emuldata_shared {

  int refs;

  pid_t group_pid;
```

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

O `refs` é um contador de referência sendo usado para determinar quando podemos liberar a estrutura para evitar vazamentos de memória. O `group_pid` é para identificar o PID (=TGID) de todo o processo (=grupo de threads). O ponteiro `threads` é o cabeçalho da lista de threading no processo.

A estrutura `linux_emuldata` pode ser obtida a partir do processo usando `em_find`. O protótipo da função é:

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

Aqui, `proc` é o processo em que queremos a estrutura emuldata e o parâmetro locked determina se queremos ou não bloquear. Os valores aceitos são `EMUL_DOLOCK` e `EMUL_DOUNLOCK`. Mais sobre o bloqueio mais tarde.

### 5.2.3. Maqueando PID

Por causa da visão diferente descrita sabendo o que é um ID de processo e ID de thread entre o FreeBSD e o Linux® nós temos que traduzir a view de alguma forma. Nós fazemos isso pelo manuseio do PID. Isto significa que nós falsificamos o que um PID (=TGID) e um TID (=PID) é entre o kernel e o userland. A regra é que no kernel (no Linuxulator) PID=PID e TGID=grupo de id → compartilhado e para userland nós apresentamos `PID=shared → group_pid` e `TID=proc → p_pid`. O membro PID da estrutura `linux_emuldata` é um PID do FreeBSD.

O acima afeta principalmente syscalls getyscl, getppid, gettid. Onde usamos PID/TGID, respectivamente. Em cópia de TIDs em `child_clear_tid` e `child_set_tid` copiamos o PID FreeBSD.

### 5.2.4. syscall Clone

A syscall `clone` é o modo como as threads são criadas no Linux®. O protótipo syscall é assim:

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

O parâmetro `flags` informa a syscall como exatamente os processos devem ser clonados. Como descrito acima, o Linux® pode criar processos compartilhando várias coisas independentemente, por exemplo, dois processos podem compartilhar file descriptors, mas não VM, etc. Ultimo byte do parametro `flags` é o sinal de saída do processo recém-criado. O parâmetro `stack` se não `NULL` diz, onde está a pilha de threading e se é `NULL` nós devemos copiar-na-escrita chamando a pilha de processos (isto é, faz a rotina normal de fork(2)). O parâmetro `parent_tidptr` é usado como um endereço para copiar o PID do processo (ou seja, o id do thread), uma vez que o processo esteja suficientemente instanciado, mas ainda não seja executável. O parâmetro `dummy` está aqui devido à convenção de chamada muito estranha desta syscall em i386. Ele usa os registradores diretamente e não deixa o compilador fazer o que resulta na necessidade de uma syscall falsa. O parâmetro `child_tidptr` é usado como um endereço para copiar o PID assim que o processo terminar de

bifurcar e quando o processo terminar.

O syscall prossegue definindo flags correspondentes dependendo dos flags passadas. Por exemplo, mapas `CLONE_VM` para RFMEM (compartilhamento de VM), etc. O único nit aqui é `CLONE_FS` e `CLONE_FILES` porque o FreeBSD não permite configurar isso separadamente, então nós o falsificamos não configurando RFFDG (copiando a tabela fd e outras informações fs) se qualquer uma delas estiver definida. Isso não causa nenhum problema, porque essas flags são sempre definidas juntas. Depois de definir as flags, o processo é bifurcado usando a rotina `fork1` interna, o processo é instrumentado para não ser colocado em uma fila de execução, ou seja, não deve ser definido como executável. Depois que a bifurcação é feita, possivelmente reparamos o processo recém-criado para emular a semântica `CLONE_PARENT`. A próxima parte está criando os dados de emulação. Threads no Linux® não sinalizam seus processos pais, então nós definimos o sinal de saída como 0 para desabilitar isso. Depois que a configuração de `child_set_tid` e `child_clear_tid` é executada, habilitando a funcionalidade posteriormente no código. Neste ponto, copiamos o PID para o endereço especificado por `parent_tidptr`. A configuração da pilha de processos é feita simplesmente reescrevendo o registro do quadro de linha `% esp` (`% rsp` no amd64). A próxima parte é configurar o TLS para o processo recém-criado. Depois disso, a semântica vfork(2) pode ser emulada e, finalmente, o processo recém-criado é colocado em uma fila de execução e copiando seu PID para o processo pai através do valor de retorno `clone` é feito.

A syscall `clone` é capaz e de fato é usado para emulação de syscalls fork() e vfork(2). O glibc mais novo em um caso de kernel 2.6 usa o `clone` para implementar syscalls fork(2) e vfork(2).

### 5.2.5. Bloqueio

O bloqueio é implementado como per-subsystem porque não esperamos muita disputa sobre eles. Existem dois bloqueios: `emul_lock` usado para proteger a manipulação de `linux_emuldata` e `emul_shared_lock` usado para manipular `linux_emuldata_shared`. O `emul_lock` é um mutex bloqueador não tolerável, enquanto `emul_shared_lock` é um bloqueio travável `sx_lock`. Devido ao bloqueio por subsistema, podemos unir alguns bloqueios e é por isso que o em-find oferece o acesso sem bloqueio.

## 5.3. TLS

Esta seção trata do TLS também conhecido como armazenamento local de thread.

### 5.3.1. Introdução ao threading

Threads na ciência da computação são entidades com um processo que podem ser agendados independentemente de qualquer outro. As threads nos processos compartilham amplos dados de processos (file descriptors, etc.) mas também tem sua prŕopria pilha para seus próprios dados. Algumas vezes é preciso para um processamento amplo de dados dado uma thread. Imagine um nome de uma thread algo assim. A tradicional API de threading do UNIX®, pthreads prove um caminho para isso em pthread_key_create(3), pthread_setspecific(3) and pthread_getspecific(3) onde a thread pode criar uma chave para os dados da thread local pthread_getspecific(3) ou pthread_getspecific(3) para manipular esses dados. Você pode ver que esse não é o caminho mais confortavel que poderia ser usado. Então varios produtores de compiladores C/C++ introduziram um caminho melhor. Eles definiram uma nova chave modificadora de thread que especifica que a

variavel é especifica de uma thread. Um novo método de acessar as variaveis foi desenvolvio como (ao menos no i386). O método pthreads tende a ser implementado no espaço de usuário como uma tabela de lookup trivial. A performance como uma solução não é muito boa. Então o novo método (no i386) registradores de segmentos para endereçar um segmento, onde a área do TLS é armazenada, então o atual acesso da variável de uma thread é apenas adicionada ao registrador de segmentos para o endereçamento via it. Os registradores de segmentos são usualmente `%gs` e `%fs` agindo como seletores de segmento. Toda thread tem sua própria área onde os dados da thread local são armazenados e o segmento deve ser carregado em toda troca de contexto. Esse método é muito rapido e usado em todo mundo em volta do UNIX® i386. Ambos FreeBSD e Linux® Implementam sua abordagem e seus resultados tem sido muito bons. Unico ponto negativo é ter que recarregar o segmento em toda troca de contexto que pode deixar o processo lento. FreeBSD tenta evitar essa sobrecarga usando apenas 1 descritor de segmento enquanto Linux® usa 3. Interessante que isso quase nunca usa mais que 1 descritor (apenas o Wine parece usar 2) então o Linux® paga esse preço desnecessário na troca de contexto.

### 5.3.2. Segmentos em i386

A arquitetura i386 implementa os então chamados segmentos.Um segmento é uma descrição de um espaço na memória. A base de endereço (baixa) na area da memória, o fim disso (teto), tipo, proteção, etc. A memória descrita por um segmento pode ser acessada usando um seletor de segmento (`%cs`, `%ds`, `%ss`, `%es`, `%fs`, `%gs`). Por exemplo, deixe nos supor que temos um segmento com base no endereço 0x1234 e comprimento e esse codigo:

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

Isso carregará o conteúdo do registro `% edx` na localização da memória 0x1244. Alguns registradores de segmento têm um uso especial, por exemplo `% cs` é usado para segmento de código e `% ss` é usado para o segmento de pilha, mas `% fs` e `% gs` geralmente não são usados. Os segmentos são armazenados em uma tabela GDT global ou em uma tabela LDT local. O LDT é acessado por meio de uma entrada no GDT. O LDT pode armazenar mais tipos de segmentos. LDT pode ser por processo. Ambas as tabelas definem até 8191 entradas.

### 5.3.3. Implementação no Linux® i386

Existem duas maneiras principais de configurar o TLS no Linux®. Pode ser definido ao clonar um processo usando a syscall `clone` ou ele pode chamar `set_thread_area`. Quando um processo passa a flag `CLONE_SETTLS` para `clone`, o kernel espera que a memória apontada pelo registrador `% esi` uma representação Linux® do espaço do usuário de um segmento, que é traduzido para a representação da máquina de um segmento e carregado em um slot GDT. O slot GDT pode ser especificado com um número ou -1 pode ser usado, o que significa que o próprio sistema deve escolher o primeiro slot livre. Na prática, a grande maioria dos programas usa apenas uma entrada de TLS e não se importa com o número da entrada. Nós exploramos isso na emulação e dependemos disso.

### 5.3.4. Emulação de TLS do Linux®

#### 5.3.4.1. i386

O carregamento de TLS para o segmento atual acontece chamando `set_thread_area` enquanto o TLS é carregado para um segundo processo em `clone` é feito no bloco separado em `clone`. Essas duas funções são muito semelhantes. A única diferença é o carregamento real do segmento GDT, que acontece na próxima troca de contexto para o processo recém-criado, enquanto `set_thread_area` deve carregar isso diretamente. O código basicamente faz isso. Ele copia o descritor de segmento de formulário Linux® da área de usuário. O código verifica o número do descritor, mas como isso difere entre o FreeBSD e o Linux®, maquiamos um pouco. Nós suportamos apenas índices de 6, 3 e -1. O número 6 é genuíno do Linux®, 3 é genuíno do FreeBSD one e -1 significa uma auto seleção. Em seguida, definimos o número do descritor como constante 3 e copiamos isso para o espaço do usuário. Contamos com o processo em espaço de usuário usando o número do descritor, mas isso funciona na maior parte do tempo (nunca vi um caso em que isso não funcionou), como o processo em espaço de usuário normalmente passa em 1. Então, convertemos o descritor da classe do Linux® para um formulário dependente da máquina (isto é, independente do sistema operacional) e copie isto para o descritor de segmento definido pelo FreeBSD. Finalmente podemos carregá-lo. Atribuímos o descritor às threads PCB (bloco de controle de processo) e carregamos o segmento `%gs` usando `load_gs`. Este carregamento deve ser feito em uma seção crítica para que nada possa nos interromper. O caso `CLONE_SETTLS` funciona exatamente como este, apenas o carregamento usando `load_gs` não é executado. O segmento usado para isso (segmento número 3) é compartilhado para este uso entre os processos do FreeBSD e do Linux® para que a camada de emulação Linux® não adicione nenhuma sobrecarga sobre o FreeBSD.

#### 5.3.4.2. amd64

A implementação do amd64 é semelhante à do i386, mas inicialmente não havia um descritor de segmento de 32 bits usado para esse propósito (por isso nem usuários nativos de TLB de 32 bits trabalhavam), então tivemos que adicionar esse segmento e implementar seu carregamento em cada troca de contexto (quando a flag sinalizando uso de 32 bits está definida). Além disso, o carregamento de TLS é exatamente o mesmo, apenas os números de segmento são diferentes e o formato do descritor e o carregamento diferem ligeiramente.

# 5.4. Futexes

## 5.4.1. Introdução à sincronização

Threads precisam de algum tipo de sincronização e POSIX® fornece alguns deles: mutexes para exclusão mútua, bloqueios de leitura/gravação para exclusão mútua com relação de polarização de leituras e gravações e variáveis de condição para sinalizar um mudança de status. É interessante observar que a API de thread POSIX® não tem suporte para semáforos. Essas implementações de rotinas de sincronização são altamente dependentes do tipo de suporte a threading que temos. No modelo puro 1:M (espaço de usuário), a implementação pode ser feita apenas no espaço do usuário e, portanto, ser muito rápida (as variáveis de condição provavelmente serão implementadas usando sinais, ou seja, não rápido) e simples. No modelo 1:1, a situação também é bastante clara - as threading devem ser sincronizadas usando as facilidades do kernel (o que é muito lento porque uma syscall deve ser executada). O cenário M:N misto combina apenas a primeira e a segunda abordagem ou depende apenas do kernel. A sincronização de threads é uma parte vital da programação ativada por threads e seu desempenho pode afetar muito o programa resultante.

Benchmarks recentes no sistema operacional FreeBSD mostraram que uma implementação sx_lock melhorada gerou 40% de aceleração no *ZFS* (um usuário sx pesado), isso é algo in-kernel, mas mostra claramente quão importante é o desempenho das primitivas de sincronização. .

Os programas em threading devem ser escritos com o mínimo de contenção possível em bloqueios. Caso contrário, em vez de fazer um trabalho útil, a threading apenas espera em um bloqueio. Devido a isso, os programas encadeados mais bem escritos mostram pouca contenção de bloqueios.

### 5.4.2. Introdução a Futexes

O Linux® implementa a segmentação 1:1, ou seja, tem de utilizar primitivas de sincronização no kernel. Como afirmado anteriormente, programas encadeados bem escritos possuem pouca contenção de bloqueio. Assim, uma sequência típica poderia ser executada como dois contador de referência de mutex de aumento/redução atômico, que é muito rápido, conforme apresentado pelo exemplo a seguir:

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

O threading 1:1 nos força a executar dois syscalls para as chamadas mutex, o que é muito lento.

A solução que o Linux® 2.6 implementa é chamada de futexes. Futexes implementam a verificação de contenção no espaço do usuário e chama primitivas do kernel apenas em um caso de contenção. Assim, o caso típico ocorre sem qualquer intervenção do kernel. Isso produz uma implementação de primitivas de sincronização razoavelmente rápida e flexível.

### 5.4.3. API do Futex

A syscall do futex é assim:

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

Neste exemplo `uaddr` é um endereço do mutex no espaço do usuário, `op` é uma operação que estamos prestes a executar e os outros parâmetros têm significado por operação.

Futexes implementam as seguintes operações:

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

#### 5.4.3.1. FUTEX_WAIT

Esta operação verifica que no endereço `uaddr` o valor `val` é gravado. Se não, `EWOULDBLOCK` é retornado, caso contrário, a thread é enfileirada no futex e é suspensa. Se o argumento `timeout` for diferente de zero, ele especificará o tempo máximo para a suspensão, caso contrário, a suspensão será infinita.

#### 5.4.3.2. FUTEX_WAKE

Esta operação tem um futex em `uaddr` e acorda os primeiros futexes `val` enfileirados neste futex.

#### 5.4.3.3. FUTEX_FD

Esta operação associa um descritor de arquivo com um determinado futex.

#### 5.4.3.4. FUTEX_REQUEUE

Esta operação pega threads `val` enfileirados no futex em `uaddr`, acorda-os e pega as próximas threads `val2` e enfileira-os no futex em `uaddr2`.

#### 5.4.3.5. FUTEX_CMP_REQUEUE

Essa operação faz o mesmo que `FUTEX_REQUEUE`, mas verifica se `val3` é igual a `val` primeiro.

#### 5.4.3.6. FUTEX_WAKE_OP

Esta operação executa uma operação atômica em `val3` (que contém algum outro valor codificado) e `uaddr`. Então, ele acorda threads `val` em futex em `uaddr` e se a operação atômica retornar um número positivo, ele ativa os threadings `val2` em futex em `uaddr2`.

As operações implementadas em `FUTEX_WAKE_OP`:

- `FUTEX_OP_SET`
- `FUTEX_OP_ADD`
- `FUTEX_OP_OR`
- `FUTEX_OP_AND`
- `FUTEX_OP_XOR`

Não existe um parâmetro `val2` no protótipo do futex. O `val2` é obtido do parâmetro `struct timespec *timeout` para as operações `FUTEX_REQUEUE`, `FUTEX_CMP_REQUEUE` e `FUTEX_WAKE_OP`.

### 5.4.4. Emulação de Futex no FreeBSD

A emulação de futex no FreeBSD é retirada do NetBSD e posteriormente estendida por nós. Ele é colocado nos arquivos linux_futex.c e linux_futex.h. A estrutura `futex` se parece com:

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

```
  int f_refcount;

  LIST_ENTRY(futex) f_list;

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

E a estrutura `waiting_proc` é:

```
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

Um futex é obtido usando a função `futex_get`, que busca uma lista linear de futexes e retorna o encontrado ou cria um novo futex. Ao liberar um futex do uso, chamamos a função `futex_put`, que diminui um contador de referência do futex e, se o refcount chegar a zero, ele é liberado.

#### 5.4.4.2. futex_sleep

Quando um futex enfileira uma thread para dormir, ele cria uma estrutura `working_proc` e coloca essa estrutura na lista dentro da estrutura do futex, então apenas executa um tsleep(9) para suspender a threading. O sleep pode ser expirado. Depois de tsleep(9) retornar (a thread foi acordada ou expirou) a estrutura `working_proc` é removida da lista e é destruído. Tudo isso é feito na função `futex_sleep`. Se nós formos acordados de `futex_wake` nós temos `wp_new_futex` setado então nós dormimos nele. Desta forma, um novo enfileiramento é feito nesta função.

#### 5.4.4.3. futex_wake

Acordar uma thread em sleep em uma futex é performado na função `futex_wake`. Primeiro nesta função nós imitamos o comportamento estranho do Linux®, onde ele acorda N threads para todas as operações, a única exceção é que as operações REQUEUE são executadas em threads N+1. Mas isso geralmente não faz diferença, pois estamos acordando todos as threads. Em seguida na função no loop nós acordamos n threads, depois disso nós checamos se existe um novo futex para requeuering. Se assim for, nós enfileiramos novamente até n2 threads no novo futex. Isso coopera com o `futex_sleep`.

#### 5.4.4.4. futex_wake_op

A operação `FUTEX_WAKE_OP` é bastante complicada. Primeiro nós obtemos dois futexes nos endereços `uaddr` e `uaddr2` e então executamos a operação atômica usando `val3` e `uaddr2`. Então os waiters `val` no primeiro futex são acordados e se a condição de operação atômica se mantém, nós acordamos o

waiter `val2` (ex `timeout`) no segundo futex.

#### 5.4.4.5. operação atômica futex

A operação atômica usa dois parâmetros `encoded_op` e `uaddr`. A operação codificada, codifica a operação em si, comparando valor, argumento de operação e argumento de comparação. O pseudocódigo da operação é como este:

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

E isso é feito atomicamente. Primeiro, uma cópia do número em `uaddr` é executada e a operação é concluída. O código manipula falhas de página e, se nenhuma falha de página ocorrer, `oldval` é comparado ao argumento `cmparg` com o comparador cmp.

#### 5.4.4.6. Bloqueio Futex

A implementação do Futex usa duas listas de lock que protegndo `sx_lock` e locks globais (Giant ou outra `sx_lock`). Cada operação é executada bloqueada desde o início até o final.

## 5.5. Implementação de várias syscalls

Nesta seção, descreverei algumas syscalls menores que merecem destaque, pois sua implementação não é óbvia ou as syscalls são interessantes de outro ponto de vista.

### 5.5.1. *na família de syscalls

Durante o desenvolvimento do kernel 2.6.16 do Linux®, os *at syscalls foram adicionados. Essas syscalls (`openat`, por exemplo) funcionam exatamente como suas contrapartes sem-menos, com a pequena exceção do parâmetro `dirfd`. Este parâmetro muda onde o arquivo dado, no qual a syscall deve ser executado, está. Quando o parâmetro `filename` é absoluto `dirfd` é ignorado, mas quando o caminho para o arquivo é relativo, ele é checado. O parâmetro `dirfd` é um diretório relativo ao qual o nome do caminho relativo é verificado. O parâmetro `dirfd` é um file descriptor de algum diretório ou `AT_FDCWD`. Então, por exemplo, a syscall `openat` pode ser assim:

```
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
*/
```

Esta infra-estrutura é necessária para evitar corridas ao abrir arquivos fora do diretório de trabalho. Imagine que um processo consiste em duas threads, thread A e thread B. Thread A emite `open (./tmp/foo/bah., Flags, mode)` e antes de retornar ele se antecipa e a thread B é executada. A thread B não se preocupa com as necessidades da thread A e renomeia ou remove o /tmp/foo/. Nós

temos uma corrida. Para evitar isso, podemos abrir o /tmp/foo e usá-lo como `dirfd` para a syscall `openat`. Isso também permite que o usuário implemente diretórios de trabalho por thread.

A família do Linux® de *at syscalls contém: `linux_openat`, `linux_mkdirat`, `linux_mknodat`, `linux_fchownat`, `linux_futimesat`, `linux_fstatat64`, `linux_unlinkat`, `linux_renameat`, `linux_linkat` , `linux_symlinkat`, `linux_readlinkat`, `linux_fchmodat` e `linux_faccessat`. Tudo isso é implementado usando a rotina modificada namei(9) e a simples camada de quebra automática.

#### 5.5.1.1. Implementação

A implementação é feita alterando a rotina namei(9) (descrita acima) para obter o parâmetro adicional `dirfd` no sua estrutura `nameidata` , que especifica o ponto inicial da pesquisa do nome do caminho, em vez de usar o diretório de trabalho atual todas as vezes. A resolução de `dirfd` do número do file descriptor para um vnode é feita em *at syscalls nativo. Quando `dirfd` é `AT_FDCWD`, a entrada `dvp` na estrutura `nameidata` é `NULL`, mas `dirfd` é um número diferente, obtemos um arquivo para este file descriptor, verificamos se este arquivo é válido e se há vnode anexado a ele, então obtemos um vnode. Então nós verificamos este vnode por ser um diretório. Na rotina real namei(9) simplesmente substituímos a variável `dvp` vnode pela variável `dp` na função namei(9), que determina o ponto de partida. O namei(9) não é usado diretamente, mas através de um rastreamento de diferentes funções em vários níveis. Por exemplo, o `openat` é assim:

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

Por esse motivo, `kern_open` e `vn_open` devem ser alterados para incorporar o parâmetro `dirfd` adicional. Nenhuma camada de compatibilidade é criada para esses, porque não há muitos usuários disso e os usuários podem ser facilmente convertidos. Esta implementação geral permite ao FreeBSD implementar suas próprias *at syscalls. Isso está sendo discutido agora.

### 5.5.2. Ioctl

A interface ioctl é bastante frágil devido à sua generalidade. Nós temos que ter em mente que os dispositivos diferem entre Linux® e FreeBSD, então alguns cuidados devem ser aplicados para fazer o trabalho de emulação de ioctl corretamente. O manuseio ioctl é implementado em linux_ioctl.c, onde a função `linux_ioctl` é definida. Esta função simplesmente itera sobre conjuntos de manipuladores ioctl para encontrar um manipulador que implementa um dado comando. A syscall ioctl tem três parâmetros, o file descriptor, comando e um argumento. O comando é um número de 16 bits, que, em teoria, é dividido em alta classe determinante de 8 bits do comando ioctl e 8 bits baixos, que são o comando real dentro do conjunto dado. A emulação aproveita essa divisão. Implementamos manipuladores para cada conjunto, como `sound_handler` ou `disk_handler`. Cada manipulador tem um comando máximo e um comando mínimo definido, que é usado para determinar qual manipulador é usado. Existem pequenos problemas com esta abordagem porque Linux® não usa a divisão definida consistentemente, por isso as ioctls para um conjunto diferente estão dentro de um conjunto ao qual não devem pertencer (ioctls genéricos SCSI dentro do cdrom conjunto, etc.). O FreeBSD atualmente não implementa muitos ioctls do Linux® (comparado ao NetBSD, por exemplo), mas o plano é portar os do NetBSD. A tendência é usar o ioctls Linux® mesmo nos drivers nativos do FreeBSD, devido à fácil portabilidade dos aplicativos.

### 5.5.3. Depuração

Cada syscall deve ser debugável. Para isso, introduzimos uma pequena infra-estrutura. Nós temos o recurso ldebug, que informa se uma dada syscall deve ser depurada (configurável através de um sysctl). Para impressão, temos as macros LMSG e ARGS. Essas são usadas para alterar uma string imprimível para mensagens uniformes de depuração.

# 6. Conclusão

## 6.1. Resultados

Em abril de 2007, a camada de emulação do Linux® é capaz de emular o kernel Linux® 2.6.16 muito bem. Os problemas remanescentes dizem respeito a futexes, inacabado na família de syscalls *at, entrega de sinais problemáticos, falta de `epoll` e `inotify` e provavelmente alguns bugs que ainda não descobrimos. Apesar disso, somos capazes de executar basicamente todos os programas Linux® incluídos na coleção de ports do FreeBSD com o Fedora Core 4 em 2.6.16 e há alguns relatos rudimentares de sucesso com o Fedora Core 6 em 2.6.16. O linux_base do Fedora Core 6 foi recentemente comprometido permitindo alguns testes adicionais da camada de emulação e nos dando mais algumas dicas onde devemos nos esforçar para implementar o material que está faltando.

Nós podemos rodar os aplicativos mais usados como o www/linux-firefox, net-im/skype e alguns jogos da coleção dos ports. Alguns dos programas exibem mau comportamento na emulação 2.6, mas isso está atualmente sob investigação e, espera-se, será corrigido em breve. A única grande aplicação que se sabe que não funciona é o Java™ Development Kit do Linux® e isto é devido ao requisito de `epoll` habilidade que não está diretamente relacionada ao kernel do Linux® 2.6.

Esperamos habilitar a emulação 2.6.16 por padrão algum tempo depois que o FreeBSD 7.0 for lançado pelo menos para expor as partes da emulação 2.6 para alguns testes mais amplos. Feito isso, podemos mudar para o Fedora Core 6 linux_base, que é o plano final.

## 6.2. Trabalho futuro

O trabalho futuro deve focar na correção dos problemas remanescentes com futexes, implementar o restante da família de syscalls, corrigir a entrega de sinal e possivelmente implementar os recursos de `epoll` e `inotify`.

Esperamos poder executar os programas mais importantes com perfeição em breve, por isso poderemos alternar para a emulação 2.6 por padrão e fazer do Fedora Core 6 o linux_base padrão porque o nosso atualmente usado Fedora Core 4 não é mais suportado.

O outro objetivo possível é compartilhar nosso código com o NetBSD e o DragonflyBSD. O NetBSD tem algum suporte para emulação 2.6, mas está longe de ser concluído e não foi realmente testado. O DragonflyBSD manifestou algum interesse em portar as melhorias do 2.6.

Geralmente, como o Linux® se desenvolve, gostaríamos de acompanhar seu desenvolvimento, implementando a syscalls recém-adicionado. Splice vem em mente primeiro. Algumas syscalls já

implementadas também são altamente danificadas, por exemplo `mremap` e outras. Alguns aprimoramentos de desempenho também podem ser feitos, um lock mais refinado e outros.

## 6.3. Equipe

Eu colaborei neste projeto com (em ordem alfabética):

- 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

Gostaria de agradecer a todas as pessoas por seus conselhos, revisões de código e apoio geral.

# 7. Literaturas


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