Emulação do Linux® no FreeBSD

Resumo

Esta tese de mestrado trata da atualização da camada de emulação do Linux® (chamada de *Linuxulator*). A tarefa consistiu em atualizar a camada para corresponder à funcionalidade do Linux® 2.6. Como implementação de referência, foi escolhido o kernel Linux® 2.6.16. O conceito é vagamente baseado na implementação do NetBSD. A maior parte do trabalho foi realizada no verão de 2006 como parte do programa de estudantes do Google Summer of Code. O foco foi trazer o suporte do *NPTL* (nova biblioteca de threads POSIX®) para a camada de emulação, incluindo *TLS* (armazenamento local de threads), *futexes* (mutexes de espaço do usuário rápidos), *PID mangling* e algumas outras pequenas coisas. Muitos problemas pequenos foram identificados e corrigidos durante o processo. Meu trabalho foi integrado ao repositório principal do FreeBSD e será incluído na próxima versão 7.0R. Nós, a equipe de desenvolvimento de emulação, estamos trabalhando para tornar a emulação do Linux® 2.6 a camada de emulação padrão no FreeBSD.

Índice

1. Introdução 1
2. Uma olhada por dentro... 2
3. Emulação 10
4. A camada de emulação do Linux® - parte MD 16
5. Camada de emulação do Linux® - Parte MI 20
6. Conclusão 31
7. Literaturas 32

1. Introdução

Nos últimos anos, os sistemas operacionais de código aberto baseados em UNIX® começaram a ser amplamente implantados em servidores e máquinas clientes. Entre esses sistemas operacionais, gostaria de destacar dois: o FreeBSD, por sua herança BSD, código comprovado ao longo do tempo e muitos recursos interessantes, e o Linux®, por sua ampla base de usuários, comunidade de desenvolvedores entusiastas e apoio de grandes empresas. O FreeBSD tende a ser usado em máquinas de classe servidor que executam tarefas de rede intensivas, com menos uso em máquinas de classe desktop para usuários comuns. Enquanto o Linux® tem o mesmo uso em servidores, mas é usado muito mais por usuários domésticos. Isso leva a uma situação em que há muitos programas somente binários disponíveis para Linux® que não possuem suporte para o FreeBSD.

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

Durante o verão de 2006, a Google Inc. patrocinou um projeto que se concentrou na extensão da

camada de emulação do Linux® (chamada de Linuxulator) no FreeBSD para incluir as funcionalidades do Linux® 2.6. Esta tese foi escrita como parte deste projeto.

2. Uma olhada por dentro...

Nesta seção, vamos descrever cada sistema operacional em questão. Como eles lidam com syscalls, trapframes, etc., tudo o que é de baixo nível. Também descrevemos a maneira como eles entendem os recursos comuns do UNIX®, como o que é um PID, o que é uma thread, etc. Na terceira subseção, falamos sobre como a emulação do UNIX® em cima do UNIX® poderia ser feita de maneira geral.

2.1. O que é UNIX®

UNIX® é um sistema operacional com uma longa história que influenciou praticamente todos os outros sistemas operacionais atualmente em uso. Desde os anos 1960, seu desenvolvimento continua até os dias de hoje (embora em projetos diferentes). O desenvolvimento do UNIX® logo se dividiu em duas principais vertentes: as famílias BSDs e System III/V. Elas se influenciaram mutuamente ao adotar um padrão comum para o UNIX®. Entre as contribuições originadas no BSD, podemos citar memória virtual, rede TCP/IP, FFS e muitas outras. O branch do System V contribuiu com primitivas de comunicação interprocesso do SysV, copy-on-write, etc. O UNIX® em si não existe mais, mas suas ideias foram utilizadas por muitos outros sistemas operacionais ao redor do mundo, formando o que chamamos de sistemas operacionais semelhantes ao UNIX®. Nos dias atuais, os mais influentes são Linux®, Solaris e possivelmente (em certa medida) o FreeBSD. Existem também derivados do UNIX® desenvolvidos por empresas (AIX, HP-UX etc.), mas eles têm migrado cada vez mais para os sistemas mencionados anteriormente. Vamos resumir as características típicas do UNIX®.

2.2. Detalhes técnicos

Cada programa em execução constitui um processo que representa um estado da computação. Um processo em execução é dividido entre o espaço do kernel e o espaço do usuário. Algumas operações só podem ser realizadas a partir do espaço do kernel (lidar com hardware etc.), mas o processo deve passar a maior parte de sua vida útil no espaço do usuário. O kernel é onde ocorre o gerenciamento dos processos, hardware e detalhes de baixo nível. O kernel fornece uma API UNIX® padronizada e unificada para o espaço do usuário. As mais importantes estão descritas 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 feita por meio de uma interrupção ou instrução especializada (pense nas instruções `SYSENTER`/`SYSCALL` para ia32). As syscalls são definidas por um número. Por exemplo, no FreeBSD, o número da syscall 85 é a swapon(2) e o número da syscall 132 é a mkfifo(2). Algumas syscalls requerem parâmetros, que são passados do espaço do usuário para o espaço do kernel de várias maneiras (dependendo da implementação). As syscalls são síncronas.

Outra forma possível de comunicação é por meio de uma *trap*. As traps ocorrem de forma

assíncrona após algum evento ocorrer (divisão por zero, falta de página etc.). Uma trap pode ser transparente para um processo (falta de página) ou pode resultar em uma reação, como o envio de um *sinal* (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 sinal. Os sinais são enviados por processos ou pelo kernel e recebidos por processos. Alguns sinais podem ser ignorados ou tratados por uma rotina fornecida pelo usuário, enquanto outros 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 de init). Todo processo em execução pode criar uma cópia idêntica de si mesmo usando a syscall fork(2). Algumas versões ligeiramente modificadas dessa syscall foram introduzidas, mas a semântica básica é a mesma. Todo processo em execução pode se transformar em outro processo usando a syscall exec(3). Foram introduzidas algumas modificações nessa syscall, mas todas servem ao mesmo propósito básico. Os processos encerram suas vidas chamando a syscall exit(2). Cada processo é identificado por um número único chamado PID. Todo processo possui um processo pai (parent) definido (identificado pelo seu PID).

2.2.4. Gerenciamento de threads

No traditional UNIX®, não é definida nenhuma API nem implementação para threads, enquanto o POSIX® define sua API de threads, mas a implementação é indefinida. Tradicionalmente, havia duas maneiras de implementar threads. Tratá-los como processos separados (threading 1:1) ou envolver todo o grupo de threads em um único processo e gerenciar as threads no espaço do usuário (threading 1:N). Vamos comparar as principais características de cada abordagem:

1:1 threading

- Threads pesadas
- O agendamento não pode ser alterado pelo usuário (ligeiramente atenuada 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 - As chamadas de sistema devem ser encapsuladas - Não pode utilizar mais do que uma CPU

2.3. O que é o FreeBSD?

O projeto FreeBSD é um dos sistemas operacionais de código aberto mais antigos atualmente disponíveis para uso diário. É um descendente direto do UNIX® genuíno, portanto, poderia ser considerado um verdadeiro UNIX®, embora questões de licenciamento não permitam isso. O início do projeto remonta ao início dos anos 1990, quando um grupo de usuários do BSD modificou o sistema operacional 386BSD. Com base neste conjunto de patches, um novo sistema operacional

surgiu, chamado FreeBSD por causa de sua licença liberal. Outro grupo criou o sistema operacional NetBSD com objetivos diferentes em mente. Vamos nos concentrar no FreeBSD.

O FreeBSD é um sistema operacional baseado em UNIX® moderno, com todos os recursos do UNIX®. Multitarefa preemptiva, facilidades multiusuário, rede TCP/IP, proteção de memória, suporte a multiprocessamento simétrico , memória virtual com cache de memória e buffer combinados, todos estão presentes. Uma das características interessantes e extremamente úteis é a capacidade de emular outros sistemas operacionais semelhantes ao UNIX®. A partir de dezembro de 2006 e do desenvolvimento 7-CURRENT, as seguintes funcionalidades de emulação são suportadas:

- Emulação FreeBSD/i386 no FreeBSD/amd64
- Emulação de FreeBSD/i386 no FreeBSD/ia64
- Emulação 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
- SVR4-emulação do UNIX® da revisão 4 do System V

As emulações desenvolvidas ativamente são a camada Linux® e várias camadas FreeBSD-on-FreeBSD. Outras não devem funcionar corretamente ou serem utilizáveis atualmente.

2.3.1. Detalhes técnicos

O FreeBSD é uma variante tradicional do UNIX® no sentido de dividir a execução dos processos em dois espaços: espaço do kernel e espaço do usuário. Existem dois tipos de entrada de processo no kernel: uma syscall e uma armadilha (trap). Existe apenas uma maneira de retornar. Nas seções subsequentes, descreveremos os três portões de/para o kernel. Toda a descrição se aplica à arquitetura i386, já que o Linuxulator existe apenas lá, mas o conceito é semelhante em outras arquiteturas. As informações foram retiradas de [1] e do código-fonte.

Entradas do sistema

O O FreeBSD tem uma abstração chamada de carregador de classe de execução, que é uma cunha no syscall execve(2). Isso utiliza uma estrutura `sysentvec`, que descreve uma ABI executável. Ela contém coisas como uma tabela de tradução de errno, uma tabela de tradução de sinais, várias funções para atender às necessidades de syscall (ajuste de pilha, coredumping, etc.). Cada ABI que o kernel do FreeBSD deseja suportar deve definir essa estrutura, pois ela é usada posteriormente no código de processamento de syscall e em alguns outros lugares. As entradas do sistema são tratadas por manipuladores de interrupção, onde podemos acessar tanto o espaço do kernel quanto o espaço do usuário de uma só vez.

Syscalls

As chamadas de sistema (syscalls) no FreeBSD são realizadas executando a interrupção `0x80` com o registro `%eax` definido para o número desejado da syscall e os argumentos passados na pilha.

Quando um processo emite a interrupção `0x80`, o tratador de interrupção `int0x80` da syscall é acionado (definido em sys/i386/i386/exception.s), que prepara os argumentos (ou seja, copia-os para a pilha) para uma chamada à 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`), determinar se a syscall é de 32 bits ou 64 bits (alterando o tamanho dos parâmetros), em seguida, os parâmetros são copiados, incluindo a syscall. Em seguida, a função da syscall real é executada com o processamento do código de retorno (casos especiais para erros `ERESTART` e `EJUSTRETURN`). Por fim, é agendado um `userret()`, alternando o processo de volta para o espaço do usuário. Os parâmetros para o manipulador da syscall real são passados na forma de `struct thread *td`, `struct syscall args *`, em que o segundo parâmetro é um ponteiro para a estrutura de parâmetros copiada.

Armadilhas (Traps)

O tratamento de traps no FreeBSD é semelhante ao tratamento de syscalls. Sempre que ocorre uma trap, um manipulador em assembly é chamado. Ele é escolhido entre alltraps, alltraps com registradores empurrados ou calltrap, dependendo do tipo de trap. Esse manipulador prepara os argumentos para uma chamada à função em C `trap()` (definida em sys/i386/i386/trap.c), que então processa a trap ocorrida. Após o processamento, pode ser enviado um sinal para o processo e/ou retornar para o espaço do usuário usando `userret()`.

Saídas

As saídas do kernel para o espaço do usuário acontecem usando a rotina em assembly `doreti`, independentemente se o kernel foi acessado por uma interrupção (trap) ou por uma chamada de sistema. Isso restaura o status do programa da pilha e retorna para o espaço do usuário.

Primitivas do UNIX®

O sistema operacional FreeBSD adere ao esquema tradicional do UNIX®, onde cada processo possui um número de identificação exclusivo, chamado de *PID* (Process ID). Os números de PID são alocados linearmente ou aleatoriamente, variando de `0` a `PID_MAX`. A alocação dos números de PID é feita usando busca linear no espaço de PID. Cada thread em um processo recebe o mesmo número de PID como resultado da chamada do getpid(2).

Atualmente, existem duas maneiras de implementar threading no FreeBSD. A primeira maneira é a modelagem de threads M:N, seguida pelo modelo de thread 1:1. A biblioteca padrão usada é a de thread M:N (`libpthread`), e você pode alternar em tempo de execução para a thread 1:1 (`libthr`). O plano é mudar em breve para a biblioteca 1:1 por padrão. Embora essas duas bibliotecas usem as mesmas primitivas do kernel, elas são acessadas por meio de APIs diferentes. A biblioteca M:N usa a família de syscalls `kse_*`, enquanto a biblioteca 1:1 usa a família de syscalls `thr_*`. Devido a isso, não há um conceito geral de ID de thread compartilhado entre o espaço do kernel e o espaço do usuário. Claro, ambas as bibliotecas de threads implementam a API de ID de thread pthread. Cada thread do kernel (conforme descrito por `struct thread`) tem um identificador `td tid`, mas isso não é acessível diretamente do espaço do usuário e serve exclusivamente às necessidades do kernel. Também é usado para a biblioteca de thread 1:1 como ID de thread pthread, mas o tratamento disso é interno à biblioteca e não se pode confiar.

Como mencionado anteriormente, existem duas implementações de threading no FreeBSD. A

biblioteca M:N divide o trabalho entre o espaço do kernel e o espaço do usuário. Uma thread é uma entidade agendada no kernel, mas pode representar vários threads no espaço do usuário. M threads no espaço do usuário são mapeadas para N threads no kernel, economizando recursos e aproveitando 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 thread do espaço do usuário para um thread do kernel, simplificando bastante o esquema. Nenhum desses designs implementa um mecanismo de justiça (um mecanismo desse tipo foi implementado, mas foi removido recentemente porque causava uma desaceleração significativa e tornava o código mais difícil de lidar).

2.4. O que é o Linux®

O Linux® é um kernel semelhante ao UNIX® originalmente desenvolvido por Linus Torvalds e que agora recebe contribuições de uma grande comunidade de programadores ao redor do mundo. Desde os seus humildes começos até os dias de hoje, com amplo suporte de empresas como IBM e Google, o Linux® é associado à sua rápida velocidade de desenvolvimento, suporte completo de hardware e modelo de organização com um ditador benevolente.

O desenvolvimento do Linux® começou em 1991 como um projeto de hobby na Universidade de Helsinki, na Finlândia. Desde então, ele adquiriu todas as características de um sistema operacional moderno semelhante ao UNIX®: suporte a multiprocessamento, suporte a vários usuários, memória virtual, rede, basicamente tudo está presente. Existem também recursos altamente avançados, como virtualização, entre outros.

A partir de 2006, o Linux® parece ser o sistema operacional de código aberto mais amplamente utilizado, com suporte de fornecedores independentes de software como Oracle, RealNetworks, Adobe, etc. A maioria do software comercial distribuído para Linux® só está disponível em forma binária, tornando impossível a recompilação para outros sistemas operacionais.

A maioria do desenvolvimento do Linux® ocorre em um sistema de controle de versão chamado Git. O Git é um sistema distribuído, então não há uma fonte central do código do Linux®, mas alguns branches são considerados proeminentes e oficiais. O esquema de numeração de versão implementado pelo Linux® consiste em quatro números A.B.C.D. Atualmente, o desenvolvimento ocorre na versão 2.6.C.D, onde C representa a versão principal, onde novos recursos são adicionados ou alterados, enquanto D é uma versão menor para correções de bugs apenas.

Mais informações podem ser obtidas em [3].

2.4.1. Detalhes técnicos

Linux® segue o esquema tradicional do UNIX® de dividir a execução de um processo em duas partes: o espaço do kernel e o espaço do usuário. O kernel pode ser acessado de duas maneiras: por meio de uma interrupção (trap) ou por meio de uma chamada de sistema (syscall). O retorno é tratado apenas de uma maneira. A descrição a seguir se aplica ao Linux® 2.6 na arquitetura i386™. Essas informações foram obtidas em [2].

Syscalls

As chamadas de sistema no Linux® são realizadas (no espaço do usuário) usando macros `syscallX`,

em que X substitui um número representando a quantidade de parâmetros da chamada de sistema específica. Essa macro é traduzida para um código que carrega o registro `%eax` com o número da chamada de sistema e executa a interrupção `0x80`. Após o retorno da chamada de sistema, é feita a chamada para tratar o retorno, que converte valores de retorno negativos em valores `errno` positivos e define `res` como `-1` em caso de erro. Sempre que a interrupção `0x80` é chamada, o processo entra no kernel no tratador de interrupção de chamada de sistema. Essa rotina salva todos os registros na pilha e chama a entrada da chamada de sistema selecionada. Observa-se que a convenção de chamada do Linux® espera que os parâmetros da chamada de sistema sejam passados via registros, como mostrado aqui:

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

Existem algumas exceções a isso, onde o Linux® usa convenções de chamada diferentes (a mais notável é a chamada de sistema `clone`).

Armadilhas (Traps)

Os tratadores de exceção são introduzidos em arch/i386/kernel/traps.c e a maioria desses tratadores ficam localizados em arch/i386/kernel/entry.S, onde o tratamento das exceções ocorre.

Saídas

O retorno da chamada de sistema é gerenciado pela função `exit` do sistema, que verifica se o processo possui trabalho inacabado e, em seguida, verifica se foram utilizados seletores fornecidos pelo usuário. Se isso ocorrer, é aplicada uma correção de pilha e, finalmente, os registros são restaurados da pilha e o processo retorna ao espaço do usuário.

Primitivas do UNIX®

Na versão 2.6, o sistema operacional Linux® redefiniu algumas das primitivas tradicionais do UNIX®, principalmente PID, TID e thread. O PID não é mais definido como único para cada processo, portanto, para alguns processos (threads), a função getppid(2) retorna o mesmo valor. A identificação única de um processo é fornecida pelo TID. Isso ocorre porque o *NPTL* (New POSIX® Thread Library) define threads como processos normais (chamados de 1:1 threading). A criação de um novo processo no Linux® 2.6 ocorre usando a chamada de sistema `clone` (as variantes de fork são reimplementadas usando essa chamada). Essa chamada clone define um conjunto de flags que afetam o comportamento do processo clonado em relação à implementação de threads. A semântica é um pouco complexa, pois não há uma única flag que indique à chamada de sistema 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` - compartilha arquivos abertos
- `CLONE_SIGHAND` - compartilha manipuladores de sinal e sinais bloqueados
- `CLONE_PARENT` - compartilha o processo pai
- `CLONE_THREAD` - ser uma thread (mais explicações abaixo)
- `CLONE_NEWNS` - novo namespace
- `CLONE_SYSVSEM` - compartilha estruturas de reversão SysV
- `CLONE_SETTLS` - configura o TLS no endereço fornecido
- `CLONE_PARENT_SETTID` - define o TID (Thread ID) no processo pai
- `CLONE_CHILD_CLEARTID` - limpa o TID (Thread ID) no processo filho
- `CLONE_CHILD_SETTID` - define o TID (Thread ID) no processo filho

A `CLONE_PARENT` define o pai real como o pai do chamador. Isso é útil para threads, porque se a thread A cria a thread B, queremos que a thread B tenha o mesmo pai do grupo de threads inteiro. A `CLONE_THREAD` faz exatamente a mesma coisa que `CLONE_PARENT`, `CLONE_VM` e `CLONE_SIGHAND`, reescreve o PID para ser o mesmo do chamador, define o sinal de saída como nenhum (none) e entra no grupo de threads. A `CLONE_SETTLS` configura as entradas do GDT (Global Descriptor Table) para manipulação de TLS (Thread Local Storage). O conjunto de flags `CLONE_*_*TID` define ou limpa o endereço fornecido pelo usuário para o TID ou 0.

Como você pode ver, o `CLONE_THREAD` faz a maior parte do trabalho e parece não se encaixar muito bem no esquema. A intenção original é incerta (até mesmo para os autores, de acordo com comentários no código), mas acredito que originalmente existia uma única flag de threading, que foi posteriormente dividida entre muitas outras flags, mas essa separação nunca foi totalmente concluída. Também não está claro para que serve essa partição, já que a glibc não a utiliza, então apenas o uso manual do clone permite que um programador acesse esses recursos.

Para programas não-threaded, o PID e TID são os mesmos. Para programas threaded, o PID e TID da primeira thread são os mesmos, e cada thread criada compartilha o mesmo PID e recebe um TID único (porque `CLONE_THREAD` é passado), também o pai é compartilhado por todos os processos que formam esse programa threaded.

O código que implementa o pthread_create(3) em NPTL define as flags de clone da seguinte forma:

```
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);
```

A `CLONE_SIGNAL` é definida 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 a definição de dicionário, emulação é a capacidade de um programa ou dispositivo imitar outro programa ou dispositivo. Isso é alcançado ao fornecer a mesma reação a um estímulo dado como o objeto emulado. Na prática, o mundo do software geralmente 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 funcionalidade de hardware (emuladores de OpenGL, emulação de unidades de ponto flutuante, etc.) e emulação de sistemas operacionais (seja no núcleo do sistema operacional ou como um programa no espaço do usuário).

A emulação é geralmente utilizada em situações em que não é viável ou possível utilizar o componente original. Por exemplo, alguém pode querer usar um programa desenvolvido para um sistema operacional diferente do que estão usando. Nesse caso, a emulação é útil. Às vezes, não há outra opção além da emulação - por exemplo, quando o dispositivo de hardware que você está tentando usar não existe (ainda/não mais), não há outra opção além da emulação. Isso ocorre com frequência ao portar um sistema operacional para uma plataforma nova (e inexistente). Às vezes, é apenas mais econômico utilizar a emulação.

Olhando a partir de um ponto de vista de implementação, existem duas abordagens principais para a implementação da emulação. Você pode emular o objeto inteiro - aceitando possíveis entradas do objeto original, mantendo o estado interno e emitindo a saída correta com base no estado e/ou na entrada. Esse 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 queira 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, apenas reverter o texto impresso é suficiente. Às vezes, o ambiente de emulação é muito semelhante ao ambiente emulado, então apenas uma fina camada de tradução é necessária para fornecer uma emulação totalmente funcional! Como você pode ver, isso é muito menos exigente de ser implementado, portanto, menos demorado e propenso a erros em comparação com a abordagem anterior. Mas a condição necessária é que os dois ambientes sejam suficientemente semelhantes. A terceira abordagem combina as duas anteriores. Na maioria das vezes, os objetos não fornecem as mesmas capacidades, então, ao emular um objeto mais poderoso em um objeto menos poderoso, temos que emular os recursos ausentes com emulação completa descrita anteriormente.

Esta tese de mestrado trata da emulação do UNIX® em UNIX®, que é exatamente o caso em que apenas uma camada fina de tradução é suficiente para fornecer uma 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 emulações tendem a ter algumas partes ausentes, mas isso geralmente não causa grandes inconvenientes. Imagine um emulador de console de jogos que emula tudo, exceto a saída de música. Sem dúvida, os jogos são jogáveis e é possível usar o emulador. Pode não ser tão confortável quanto o console de jogos original, mas é um compromisso aceitável entre preço e conforto.

O mesmo acontece com a API do UNIX®. A maioria dos programas pode funcionar com um conjunto muito limitado de syscalls. Essas syscalls tendem a ser as mais antigas (read(2)/write(2), família fork(2), manipulação de signal(3), exit(3), API socket(2)), o que torna mais fácil a emulação, pois sua semântica é compartilhada entre todos os sistemas UNIX® existentes hoje em dia.

3. Emulação

3.1. Como funciona a emulação no FreeBSD

Como mencionado anteriormente, o FreeBSD suporta a execução de binários de vários outros sistemas UNIX®. Isso é possível porque o FreeBSD possui uma abstração chamada de "execução do carregador de classe" (execution class loader). Isso é inserido na chamada de sistema execve(2), então quando o execve(2) está prestes a executar um binário, ele examina o tipo do binário.

Existem basicamente dois tipos de binários no FreeBSD. Scripts de texto semelhantes a shell, que são identificados pelos primeiros dois caracteres `#!`, e binários normais (geralmente ELF), que são uma representação de um objeto executável compilado. A grande maioria (pode-se dizer que todos) os binários no FreeBSD são do tipo ELF. Os arquivos ELF contêm um cabeçalho que especifica a ABI do sistema operacional para este arquivo ELF. Lendo essa informação, o sistema operacional pode determinar com precisão qual é o tipo de arquivo binário.

Cada ABI de sistema operacional deve ser registrada no kernel do FreeBSD. Isso também se aplica à ABI nativa do FreeBSD. Portanto, quando o execve(2) executa um binário, ele itera pela lista de APIs registradas e, quando encontra a correspondente, começa a usar as informações contidas na descrição da ABI do sistema operacional (sua tabela de syscalls, tabela de tradução de `errno`, etc.). Portanto, cada vez que o processo chama uma syscall, ele usa seu próprio conjunto de syscalls em vez de um conjunto global. Isso fornece uma maneira muito elegante e fácil de oferecer suporte à execução de vários formatos binários.

A natureza da emulação de diferentes sistemas operacionais (e também de outros subsistemas) levou os desenvolvedores a adotarem um mecanismo de tratamento de eventos. Existem vários pontos no kernel em que uma lista de manipuladores de eventos é chamada. Cada subsistema pode registrar um manipulador de evento e eles são chamados de acordo. Por exemplo, quando um processo é encerrado, é chamado um manipulador que possivelmente realiza a limpeza necessária no subsistema.

Essas facilidades simples fornecem basicamente tudo o que é necessário para a infraestrutura de

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

3.2. Primitivas comuns no kernel do FreeBSD

As camadas de emulação precisam de suporte por parte do sistema operacional. Vou descrever alguns dos primitivos suportados 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 estão as descrições das 3 famílias. Para cada trava, é recomendado verificar a página do manual vinculada (quando possível) para obter explicações mais detalhadas.

Operações atômicas e barreiras de memória

As operações atômicas são implementadas por meio de um conjunto de funções que realizam operações aritméticas simples em operandos de memória de maneira atômica em relação a eventos externos (interrupções, preempção, etc.). As operações atômicas podem garantir atomicidade apenas em tipos de dados pequenos (da ordem de magnitude do tipo de dados C `.long.` da arquitetura), portanto, devem ser raramente usadas diretamente no código de nível final, a menos que seja apenas para operações muito simples (como definir uma bandeira em um bitmap, por exemplo). Na verdade, é bastante simples e comum escrever uma semântica errada baseada apenas em operações atômicas (geralmente referidas como "sem bloqueio"). O kernel do FreeBSD oferece uma maneira de realizar operações atômicas em conjunto com uma barreira de memória. As barreiras de memória garantem que uma operação atômica ocorra seguindo alguma ordem especificada em relação a outros acessos à memória. Por exemplo, se precisamos que uma operação atômica ocorra logo após todas as gravações pendentes (em termos de reordenação de instruções nos buffers) sejam concluídas, precisamos usar explicitamente uma barreira de memória em conjunto com essa operação atômica. Portanto, é fácil entender por que as barreiras de memória desempenham um papel fundamental na construção de travas de nível superior (como refcount, mutexes, etc.). Para uma explicação detalhada sobre as operações atômicas, consulte o atomic(9). No entanto, é importante notar que as operações atômicas (assim como as barreiras de memória) idealmente devem ser usadas apenas para a construção de travas de nível superior (como mutexes).

Refcounts

Refcounts são interfaces para lidar com contadores de referência. Eles são implementados por meio de operações atômicas e destinam-se a serem usados apenas em casos em que o contador de referência é a única coisa a ser protegida, então até mesmo algo como um spin-mutex é considerado obsoleto. O uso da interface refcount para estruturas em que já é usado um mutex geralmente está incorreto, pois provavelmente devemos fechar o contador de referência em algum caminho já protegido. Atualmente, não existe uma página de manual que discuta refcount, apenas verifique sys/refcount.h para obter uma visão geral da API existente.

Locks

O kernel do FreeBSD possui várias classes de locks. Cada lock é definido por algumas propriedades específicas, mas provavelmente a mais importante é o evento vinculado aos detentores em disputa (ou em outras palavras, o comportamento das threads incapazes de adquirir o lock). O esquema de locking do FreeBSD apresenta três comportamentos diferentes para os contendores:

1. spinning
2. blocking
3. sleeping



números não são casuais

Spinning locks

Spin locks permitem que os aguardantes fiquem girando em um loop até que não possam adquirir o lock. Uma questão importante a lidar é quando uma thread disputa um spin lock se ela não for despachada. Como o kernel do FreeBSD é preemptivo, isso expõe o spin lock ao risco de deadlocks que podem ser resolvidos desabilitando as interrupções enquanto eles são adquiridos. Por esse e outros motivos (como a falta de suporte à propagação de prioridade, deficiência em esquemas de balanceamento de carga entre CPUs, etc.), os spin locks são destinados a proteger trechos muito pequenos de código, ou idealmente não devem ser usados se não forem explicitamente solicitados (explicado posteriormente).

Bloqueio

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

Sleeping

As travas de suspensão (sleep locks) permitem que os processos em espera sejam despachados (descheduled) e adormeçam até que o detentor da trava a solte e acorde um ou mais processos em espera. Como as travas de suspensão são projetadas para proteger grandes trechos de código e lidar com eventos assíncronos, elas não realizam qualquer forma de propagação de prioridade. Elas devem ser implementadas por meio da interface sleepqueue(9).

A ordem usada para adquirir locks é muito importante, não apenas devido à possibilidade de deadlock devido a inversões na ordem dos locks, mas também porque a aquisição de locks deve seguir regras específicas relacionadas às naturezas dos locks. Se você observar a tabela acima, a regra prática é que se um thread possui um lock de nível n (onde o nível é o número listado próximo ao tipo de lock), ele não pode adquirir um lock de níveis superiores, pois isso quebraria a semântica especificada para um determinado caminho. Por exemplo, se um thread possui um bloqueio de bloqueio (nível 2), é permitido adquirir um lock de rotação (nível 1), mas não um lock de suspensão (nível 3), pois os bloqueios de bloqueio são destinados a proteger caminhos menores do que os bloqueios de suspensão (essas regras não se aplicam a operações atômicas ou barreiras de agendamento, no entanto).

Esta é uma lista de bloqueio com seus respectivos comportamentos:

- spin mutex - girando - mutex(9)
- Sleep mutex - bloqueio - mutex(9)
- pool mutex - blocking - mtx(pool)
- A família de funções de suspensão (sleep family) - sleeping - sleep(9) pause tsleep msleep msleep spin msleep rw msleep sx
- condvar - sleeping - condvar(9)
- rwlock - blocking - rwlock(9)
- sxlock - sleeping - sx(9)
- lockmgr - sleeping - lockmgr(9)
- 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.

Barreiras de agendamento

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

- seções críticas (e preempção)
- sched_bind
- sched_pin

Em geral, esses devem ser usados apenas em um contexto específico e, mesmo que possam substituir bloqueios em muitos casos, eles devem ser evitados porque não permitem diagnosticar problemas simples com ferramentas de depuração de bloqueio (como witness(4)).

Seções críticas

The FreeBSD kernel has been made preemptive basically to deal with interrupt threads. In fact, to avoid high interrupt latency, time-sharing priority threads can be preempted by interrupt threads (in this way, they do not need to wait to be scheduled as the normal path previews). Preemption, however, introduces new racing points that need to be handled, as well. Often, to deal with

preemption, the simplest thing to do is to completely disable it. A critical section defines a piece of code (borderlined by the pair of functions critical_enter(9) and critical_exit(9), where preemption is guaranteed to not happen (until the protected code is fully executed). This can often replace a lock effectively but should be used carefully to not lose the whole advantage that preemption brings.

sched_pin/sched_unpin

Outra forma de lidar com a preempção é a interface `sched_pin()`. Se um trecho de código é envolvido pelas funções `sched_pin()` e `sched_unpin()`, é garantido que a respectiva thread, mesmo que possa ser preemptada, será sempre executada na mesma CPU. Fixar (pinning) é muito efetivo no caso particular em que precisamos acessar dados específicos de cada CPU e assumimos que outras threads não alterarão esses dados. A última condição determinará uma seção crítica como uma condição muito rigorosa para nosso código.

sched_bind/sched_unbind

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

3.2.2. Estrutura Proc

Em algumas camadas de emulação, às vezes é necessário ter dados adicionais específicos para cada processo. Pode-se gerenciar estruturas separadas (como uma lista, uma árvore etc.) que contenham esses dados para cada processo, mas isso pode ser lento e consumir muita memória. Para resolver esse problema, a estrutura `proc` do FreeBSD contém o campo `p_emuldata`, que é um ponteiro vazio para dados específicos da camada de emulação. Essa entrada `proc` é protegida pelo mutex do processo.

A estrutura `proc` do FreeBSD contém uma entrada `p_sysent` que identifica qual ABI esse processo está executando. Na verdade, é um ponteiro para a estrutura `sysentvec` descrita anteriormente. Portanto, ao comparar esse ponteiro com o endereço onde a estrutura `sysentvec` para a ABI específica está armazenada, podemos determinar efetivamente se o processo pertence à nossa camada de emulação. O código geralmente se parece com:

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

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

3.2.3. VFS

O subsistema VFS do FreeBSD é muito complexo, mas a camada de emulação do Linux® utiliza apenas um pequeno subconjunto por meio de uma API bem definida. Ela pode operar em vnodes ou manipuladores de arquivo. Vnode representa um vnode virtual, ou seja, uma representação de um nó no VFS. Outra representação é um manipulador de arquivo, que representa um arquivo aberto do ponto de vista de um processo. Um manipulador de arquivo pode representar um socket ou um arquivo comum. Um manipulador de arquivo contém um ponteiro para seu vnode. Mais de um manipulador de arquivo pode apontar para o mesmo vnode.

namei

A rotina namei(9) é um ponto de entrada central para a pesquisa e tradução de caminhos de nomes. Ela percorre o caminho ponto a ponto, do ponto de partida ao ponto final, usando a função de pesquisa, que é interna ao VFS. A chamada namei(9) pode lidar com links simbólicos, caminhos absolutos e relativos. Quando um caminho é pesquisado 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.

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 vnode específico (bloqueado). Esse processo é não confiável, mas funciona muito bem na maioria dos casos comuns. A falta de confiabilidade ocorre porque ela depende do cache do VFS (não percorre as estruturas no meio físico) e não funciona com links rígidos, entre outras limitações. Essa rotina é usada em vários lugares no Linuxulator.

Operações de vnode

- `fgetvp` - dado um thread 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) - busca um caminho para um diretório específico
- 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 de um vnode e desbloqueia
- vrele(9) - diminui o contador de uso para um vnode
- vref(9) - incrementa a contagem de uso de um vnode

Operações do manipulador de arquivos

- `fget` - dado um thread e um número de descritor de arquivo, ele retorna o file handler associado e o referencia

- `fdrop` - remove uma referência a um file handler
- `fhold` - referencia um file handler

4. A camada de emulação do Linux® - parte MD

Esta seção trata da implementação da camada de emulação do Linux® no sistema operacional FreeBSD. Ela descreve primeiramente a parte dependente da máquina, abordando como e onde a interação entre o espaço do usuário e o kernel é implementada. Ela fala sobre syscalls, sinais, ptrace, traps e ajuste de pilha. Essa parte discute o i386, mas é escrita de forma geral, então outras arquiteturas não devem diferir muito. A próxima parte é a parte independente da máquina do Linuxulator. Esta seção aborda apenas o i386 e o tratamento de arquivos ELF. O formato A.OUT está obsoleto e não foi testado.

4.1. Manipulação de Syscall

O tratamento de syscalls é principalmente escrito em linux_sysvec.c, que abrange a maioria das rotinas apontadas na estrutura `sysentvec`. Quando um processo do Linux® em execução no FreeBSD faz uma syscall, a rotina geral de syscall chama a rotina linux prepsyscall para a ABI do Linux®.

4.1.1. Linux® prepsyscall

No Linux®, os argumentos das syscalls são passados via registradores (por isso é limitado a 6 parâmetros no i386), enquanto no FreeBSD eles são passados pela pilha. A rotina linux prepsyscall deve copiar os parâmetros dos registradores para a pilha. A ordem dos registradores é: `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, `%ebp`. A questão é que isso é verdade apenas para *a maioria* das syscalls. Algumas (mais notavelmente `clone`) usam uma ordem diferente, mas felizmente é fácil corrigir isso inserindo um parâmetro fictício no protótipo `linux_clone`.

4.1.2. Escrevendo syscall

Cada chamada de sistema implementada no Linuxulator deve ter seu protótipo com várias flags no arquivo `syscalls.master`. A estrutura do arquivo é a seguinte:

```
...
    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 de syscall. Pode ser `STD`, `OBSOL`, `NOPROTO` ou `UNIMPL`. `STD` é uma syscall padrão com protótipo e implementação completos. `OBSOL` é obsoleta e define apenas o protótipo. `NOPROTO` significa que a syscall é implementada em outro lugar, então não é necessário adicionar o

prefixo ABI, etc. `UNIMPL` significa que a syscall será substituída pela syscall `nosys` (uma syscall que apenas imprime uma mensagem informando que a syscall não está implementada e retorna `ENOSYS`).

Do syscalls.master, um script gera três arquivos: linux_syscall.h, linux_proto.h e linux_sysent.c. O arquivo linux_syscall.h contém as definições dos nomes das syscalls 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, o 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 tem argumentos, o que é exibido pela estrutura de argumentos fictícia. Por outro lado, `close` é apenas um alias para o verdadeiro close(2) do FreeBSD, então não possui uma estrutura de argumentos específica do Linux associada e na tabela de entrada do sistema não é prefixado com linux, pois chama o verdadeiro close(2) no kernel.

4.1.3. Dummy syscalls

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

4.2. Manuseio de signals

O tratamento de sinais é feito geralmente no kernel do FreeBSD para todas as compatibilidades binárias com uma chamada a uma camada dependente de compatibilidade. A camada de

compatibilidade do Linux® define a rotina `linux_sendsig` para este propósito.

4.2.1. Linux® sendsig

Essa rotina primeiro verifica se o sinal foi instalado com `SA_SIGINFO`, caso contrário, ela chama a rotina `linux_rt_sendsig`. Além disso, aloca (ou reutiliza um já existente) o contexto do manipulador de sinal, em seguida, constrói uma lista de argumentos para o manipulador de sinal. Traduz o número do sinal com base na tabela de tradução de sinais, atribui um manipulador e traduz o conjunto de sinais. Em seguida, salva o contexto para a rotina `sigreturn` (diversos registradores, número de interrupção traduzido e máscara de sinais). Por fim, copia o contexto do sinal para o espaço do usuário e prepara o contexto para a execução real do manipulador de sinal.

4.2.2. linux_rt_sendsig

Esta rotina é semelhante à `linux_sendsig`, mas a preparação do contexto do sinal é diferente. Ela adiciona `siginfo`, `ucontext` e algumas partes POSIX®. Pode valer a pena considerar se essas duas funções não poderiam ser mescladas com o benefício de menos duplicação de código e possivelmente até execução mais rápida.

4.2.3. linux_sigreturn

Essa chamada de sistema é usada para retornar do manipulador de sinal. Ela realiza algumas verificações de segurança e restaura o contexto original do processo. Além disso, desativa o sinal na máscara de sinais do processo.

4.3. Ptrace

Many UNIX® derivates implement the ptrace(2) syscall to allow various tracking and debugging features. This facility enables the tracing process to obtain various information about the traced process, like register dumps, any memory from the process address space, etc. and also to trace the process like in stepping an instruction or between system entries (syscalls and traps). ptrace(2) also lets you set various information in the traced process (registers etc.). ptrace(2) is a UNIX®-wide standard implemented in most UNIX®es around the world.

A emulação do Linux® no FreeBSD implementa a facilidade ptrace(2) em linux_ptrace.c. As rotinas de conversão de registradores entre Linux® e FreeBSD e a própria syscall de emulação ptrace(2). A syscall é um longo bloco switch que implementa seu equivalente no FreeBSD para cada comando ptrace(2). Os comandos ptrace(2) são principalmente iguais entre Linux® e FreeBSD, então geralmente apenas uma pequena modificação é necessária. Por exemplo, `PT_GETREGS` no Linux® opera em dados diretos, enquanto o FreeBSD usa um ponteiro para os dados, então após a execução de uma syscall ptrace(2) (nativa), é necessário fazer uma cópia dos dados usando copyout para preservar a semântica do Linux®.

A implementação ptrace(2) no Linuxulator tem algumas fraquezas conhecidas. Houve panes vistas ao usar `strace` (que é um consumidor ptrace(2)) no ambiente Linuxulator. Além disso, `PT_SYSCALL` não está implementado.

4.4. Armadilhas (Traps)

Sempre que um processo Linux® executando na camada de emulação sofre uma interrupção, a própria interrupção é tratada de forma transparente com a única exceção da tradução da interrupção. O Linux® e o FreeBSD diferem em suas opiniões sobre o que é uma interrupção, 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 link-editor em tempo de execução RTLD espera que as chamadas AUX estejam presentes na pilha durante um `execve`, então é necessário fazer um ajuste para garantir isso. Naturalmente, cada sistema RTLD é diferente, então a camada de emulação deve fornecer sua própria rotina de ajuste da pilha para realizar isso. Isso também se aplica ao Linuxulator. A função `elf_linux_fixup` simplesmente copia as chamadas AUX para a pilha e ajusta o ponteiro da pilha do processo de espaço de usuário para apontar imediatamente após essas chamadas. Dessa forma, o RTLD funciona de maneira inteligente.

4.6. Suporte para A.OUT

A camada de emulação do Linux® no i386 também suporta binários A.OUT do Linux®. Praticamente tudo o que foi descrito nas seções anteriores deve ser implementado para oferecer suporte ao formato A.OUT (exceto a 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 esse formato. No entanto, isso não causa nenhum problema, já que o linux-base nos ports provavelmente não suporta binários A.OUT. Esse suporte provavelmente será removido no futuro. A maior parte do necessário para carregar binários A.OUT do Linux® está no arquivo imgact_linux.c.

5. Camada de emulação do Linux® - Parte MI

Esta seção aborda a parte independente de máquina do Linuxulator. Ela cobre a infraestrutura de emulação necessária para a emulação do Linux® 2.6, a implementação do armazenamento local de threads (TLS) (em i386) e futexes. Em seguida, falamos brevemente sobre algumas syscalls.

5.1. Descrição do NPTL

Um dos principais avanços no desenvolvimento do Linux® 2.6 foi a implementação de threads. Antes do 2.6, o suporte a threading no Linux® era implementado na biblioteca linuxthreads. Essa biblioteca era uma implementação parcial da threading POSIX®. A threading era implementada usando processos separados para cada thread, usando a syscall `clone` para permitir que compartilhassem o espaço de endereçamento (e outras coisas). As principais fraquezas dessa abordagem eram que cada thread tinha um PID diferente, o tratamento de sinais era problemático (do ponto de vista do pthreads), etc. Além disso, o desempenho não era muito bom (uso de sinais `SIGUSR` para sincronização de threads, consumo de recursos do kernel, etc.), então, para superar esses problemas, um novo sistema de threading foi desenvolvido e chamado de NPTL.

A biblioteca NPTL concentrou-se em duas coisas, mas uma terceira acabou sendo incluída e é geralmente considerada parte do NPTL. Essas duas coisas eram a incorporação de threads em uma estrutura de processo e futexes. A terceira coisa adicional foi TLS (Thread-Local Storage), que não é diretamente exigida pelo NPTL, mas toda a biblioteca NPTL do espaço do usuário depende dela. Essas melhorias resultaram em um desempenho muito melhorado e conformidade com padrões. O NPTL é uma biblioteca padrão de threading em sistemas Linux® nos dias de hoje.

A implementação do Linuxulator no FreeBSD aborda o NPTL em três áreas principais. O TLS (Thread-Local Storage), os futexes e a manipulação de PID, que tem como objetivo simular as threads do Linux®. Seções adicionais descrevem cada uma dessas áreas em detalhes.

5.2. Infraestrutura de emulação do Linux® 2.6

Essas seções lidam com a forma como as threads do Linux® são gerenciadas e como 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 em tempo de execução da versão emulada. Isso é feito por meio do sysctl(8), mais especificamente através do parâmetro `compat.linux.osrelease`. Configurar esse sysctl(8) afeta o comportamento em tempo de execução da camada de emulação. Ao definir para 2.6.x, ele define o valor de `linux_use_linux26`, enquanto definir para qualquer outro valor mantém esse valor como indefinido. Essa variável (juntamente com as variáveis específicas de cada prisão) determina se a infraestrutura 2.6 (principalmente o processo de manipulação de PID) é usada no código ou não. A configuração da versão é feita em todo o sistema e isso afeta todos os processos do Linux®. O sysctl(8) não deve ser alterado ao executar qualquer binário do Linux®, pois isso pode causar problemas.

5.2.2. Processos e Identificadores de Thread no Linux®

A semântica de threading no Linux® é um pouco confusa e utiliza uma nomenclatura completamente diferente do FreeBSD. Em um processo no Linux®, há uma estrutura chamada `struct task` que contém dois campos de identificação - PID e TGID. O PID *não* é um identificador de processo, mas sim um identificador de thread. O TGID identifica um grupo de threads, ou seja, um processo. Para processos com apenas uma thread, o PID é igual ao TGID.

No NPTL, uma thread é apenas um processo comum que tem o TGID diferente do PID e tem um líder de grupo diferente de si mesmo (além de compartilhar a memória virtual, é claro). Todo o resto acontece da mesma maneira que em um processo comum. Não há separação de um status compartilhado em uma estrutura externa, como no FreeBSD. Isso cria alguma duplicação de informações e possíveis inconsistências de dados. O kernel do Linux® parece usar informações de tarefa → grupo em alguns lugares e informações de tarefa em outros lugares, e isso realmente não é muito consistente e parece propenso a erros.

Cada thread NPTL é criado por meio de uma chamada ao syscall `clone` com um conjunto específico de flags (mais detalhes na próxima subseção). O NPTL implementa uma estrita relação de threading 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 FreeBSD que se conecta a esta estrutura. Os campos `child_se_tid` e `child_clear_tid` são usados para copiar o endereço TID quando um processo termina ou é criado. O ponteiro `shared` aponta para uma estrutura compartilhada entre as threads. A variável `pdeath_signal` identifica o sinal de término do 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 campo `refs` é um contador de referências usado para determinar quando podemos liberar a estrutura para evitar vazamentos de memória. O campo `group_pid` é usado para identificar o PID (= TGID) do processo completo (= grupo de threads). O ponteiro `threads` é a cabeça da lista de threads no processo.

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

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

Aqui, `proc` é o processo do qual queremos obter a estrutura emuldata e o parâmetro `locked` determina se queremos realizar o bloqueio ou não. Os valores aceitos são `EMUL_DOLOCK` e `EMUL_DOUNLOCK`. Mais sobre bloqueio será explicado posteriormente.

5.2.3. Maqueando PID

Devido à diferença na interpretação do conceito de PID e TID entre o FreeBSD e o Linux®, precisamos fazer uma tradução para conciliar essas visões. Fazemos isso através da manipulação de PIDs. Isso significa que simulamos o que seria um PID (=TGID) e TID (=PID) entre o kernel e o espaço do usuário. A regra geral é que, no kernel (no Linuxulator), PID = PID e TGID = shared→group_pid, e no espaço do usuário apresentamos `PID = shared→group_pid` e `TID = proc→p_pid`. O membro PID da estrutura `linux_emuldata` é um PID do FreeBSD.

A situação acima afeta principalmente as chamadas de sistema getpid, getppid e gettid. Nelas, utilizamos o PID/TGID respectivamente. Na cópia de TIDs em `child_clear_tid` e `child_set_tid`, copiamos o PID do FreeBSD.

5.2.4. syscall Clone

A chamada de sistema `clone` é a forma como as threads são criadas no Linux®. O protótipo da syscall é assim:

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

O parâmetro `flags` informa à syscall como exatamente os processos devem ser clonados. Como descrito anteriormente, o Linux® pode criar processos que compartilham várias coisas de forma independente, por exemplo, dois processos podem compartilhar descritores de arquivo, mas não a VM, etc. O último byte do parâmetro `flags` é o sinal de saída do novo processo criado. O parâmetro `stack`, se não for `NULL`, indica onde está a pilha da thread, e se for `NULL`, devemos fazer uma cópia em escrita-compartilhada da pilha do processo chamador (ou seja, fazer o que a rotina normal do fork(2) faz). O parâmetro `parent_tidptr` é usado como endereço para copiar o PID do processo (ou seja, ID da thread) assim que o processo estiver suficientemente instanciado, mas ainda não estiver

em execução. O parâmetro `dummy` está aqui por causa da convenção de chamada muito estranha dessa syscall no i386. Ela usa os registradores diretamente e não permite que o compilador o faça, o que resulta na necessidade de um syscall dummy. O parâmetro `child_tidptr` é usado como endereço para copiar o PID assim que o processo terminar de criar um novo processo e quando o processo terminar.

A syscall em si prossegue configurando as flags correspondentes com base nas flags passadas. Por exemplo, `CLONE_VM` mapeia para RFMEM (compartilhamento de VM), etc. O único detalhe aqui é `CLONE_FS` e `CLONE_FILES`, porque o FreeBSD não permite configurar isso separadamente, então simulamos não configurando RFFDG (cópia da tabela de descritores de arquivo e outras informações do sistema de arquivos) se algum desses estiver definido. Isso não causa problemas, porque essas flags são sempre configuradas juntas. Após configurar as flags, o processo é bifurcado usando a rotina interna `fork1`, e instruímos o processo a não ser colocado em uma fila de execução, ou seja, não ser definido como executável. Depois que a bifurcação é concluída, possivelmente reparentamos o processo recém-criado para emular a semântica `CLONE_PARENT`. A próxima parte é a criação dos dados de emulação. As threads no Linux® não sinalizam seus pais, então definimos o sinal de saída como 0 para desativar isso. Em seguida, é feita a configuração de `child_set_tid` e `child_clear_tid`, habilitando a funcionalidade posteriormente no código. Nesse ponto, copiamos o PID para o endereço especificado por `parent_tidptr`. A configuração da pilha do processo é feita simplesmente reescrevendo o registro `%esp` do quadro da thread (`%rsp` no amd64). A próxima parte é configurar o TLS para o processo recém-criado. Após isso, as semânticas de vfork(2) podem ser emuladas e, finalmente, o processo recém-criado é colocado em uma fila de execução e a cópia de seu PID para o processo pai é feita por meio do valor de retorno do `clone`.

O syscall `clone` é capaz e, de fato, é usado para emular as chamadas de sistema clássicas fork(2) e vfork(2). O glibc mais recente, no caso do kernel 2.6, usa `clone` para implementar as chamadas de sistema fork(2) e vfork(2).

5.2.5. Bloqueio

O sistema de bloqueio é implementado por subsistema, pois não esperamos muita contenção nesses pontos. 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 uma mutex de bloqueio não adormecível, enquanto o `emul_shared_lock` é um bloqueio `sx_lock` de bloqueio adormecível. Devido ao bloqueio por subsistema, podemos combinar alguns bloqueios e é por isso que a função `em_find` oferece acesso sem bloqueio.

5.3. TLS

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

5.3.1. Introdução ao threading

As threads na ciência da computação são entidades dentro de um processo que podem ser agendadas independentemente umas das outras. As threads no processo compartilham dados em todo o processo (descritores de arquivos, etc.), mas também possuem sua própria pilha para seus próprios dados. Às vezes, há necessidade de dados específicos de um thread em todo o processo, como o nome do thread em execução, por exemplo. A API de threads tradicional do UNIX®,

pthreads, oferece uma maneira de fazer isso usando as funções `pthread_key_create`, `pthread_setspecific` e `pthread_getspecific`, onde um thread pode criar uma chave para os dados específicos do thread local e manipular esses dados usando as funções `pthread_setspecific` e `pthread_getspecific`. É fácil perceber que essa não é a maneira mais conveniente de fazer isso. Portanto, vários fabricantes de compiladores C/C++ introduziram uma maneira melhor. Eles definiram uma nova palavra-chave de modificação, `thread`, que especifica que uma variável é específica de um thread. Também foi desenvolvido um novo método de acesso a essas variáveis, pelo menos na arquitetura i386. O método tradicional do pthreads tende a ser implementado no espaço do usuário como uma tabela de pesquisa trivial. O desempenho de tal solução não é muito bom. Portanto, o novo método usa (no i386) registradores de segmento para endereçar um segmento, onde a área TLS é armazenada, de modo que o acesso real a uma variável do thread é apenas a concatenação do registrador de segmento ao endereço, permitindo o acesso direto através do registrador de segmento. Os registradores de segmento geralmente são `%gs` e `%fs`, agindo como seletores de segmento. Cada thread tem sua própria área onde os dados locais do thread são armazenados, e o registrador de segmento precisa ser carregado em cada troca de contexto. Esse método é muito rápido e amplamente usado em todo o mundo UNIX® na arquitetura i386. Tanto o FreeBSD quanto o Linux® implementam essa abordagem e obtêm resultados muito bons. A única desvantagem é a necessidade de recarregar o segmento em cada troca de contexto, o que pode retardar as trocas de contexto. O FreeBSD tenta evitar essa sobrecarga usando apenas um descritor de segmento para isso, enquanto o Linux® usa 3. É interessante observar que quase nada usa mais de 1 descritor (apenas o Wine parece usar 2), então o Linux® paga um preço desnecessário pelas trocas de contexto.

5.3.2. Segmentos em i386

A arquitetura i386 implementa os chamados segmentos. Um segmento é uma descrição de uma área de memória. Ele contém informações como o endereço base (início) da área de memória, o final (teto), tipo, proteção, etc. A memória descrita por um segmento pode ser acessada usando registradores de seleção de segmento (`%cs`, `%ds`, `%ss`, `%es`, `%fs`, `%gs`). Por exemplo, vamos supor que temos um segmento cujo endereço base é 0x1234 e comprimento e o seguinte código:

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

Isso irá carregar o conteúdo do registro `%edx` no endereço de memória 0x1244. Alguns registradores de segmento têm um uso especial, por exemplo, `%cs` é usado para o 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 global GDT ou em uma tabela local LDT. A LDT é acessada por meio de uma entrada na GDT. A LDT pode armazenar mais tipos de segmentos. A LDT pode ser por processo. Ambas as tabelas definem até 8191 entradas.

5.3.3. Implementação no Linux® i386

Existem duas principais maneiras de configurar o TLS no Linux®. Pode ser definido ao clonar um processo usando a chamada de sistema `clone` ou pode chamar `set_thread_area`. Quando um processo passa a flag `CLONE_SETTLS` para `clone`, o kernel espera que a memória apontada pelo registro `%esi` seja uma representação do espaço do usuário Linux® de um segmento, que é traduzido para a representação do segmento da máquina 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 sistema deve escolher automaticamente o primeiro slot livre. Na prática, a grande maioria dos programas usa apenas uma entrada TLS e não se preocupa com o número da entrada. Exploramos isso na emulação e, na verdade, dependemos disso.

5.3.4. Emulação de TLS do Linux®

i386

O carregamento do TLS para a thread atual é feito chamando `set_thread_area`, enquanto o carregamento do TLS para um segundo processo no `clone` é feito em um bloco separado no `clone`. Essas duas funções são muito semelhantes. A única diferença é o carregamento real do segmento GDT, que ocorre na próxima troca de contexto para o processo recém-criado, enquanto o `set_thread_area` deve carregar isso diretamente. O código basicamente faz isso. Ele copia o descritor do segmento no formato Linux® do espaço do usuário. O código verifica o número do descritor, mas como isso difere entre FreeBSD e Linux®, nós fazemos uma pequena manipulação. Nós suportamos apenas os índices 6, 3 e -1. O 6 é um número genuíno do Linux®, o 3 é um número genuíno do FreeBSD e -1 significa seleção automática. Em seguida, definimos o número do descritor como o valor constante 3 e copiamos isso de volta para o espaço do usuário. Nós confiamos no processo do espaço do usuário usando o número do descritor, e isso funciona na maioria das vezes (nunca vi um caso em que isso não funcionasse), pois o processo do espaço do usuário normalmente passa o número 1. Em seguida, convertemos o descritor do formato Linux® para um formato dependente da máquina (ou seja, independente do sistema operacional) e copiamos isso para o descritor de segmento definido no FreeBSD. Finalmente, podemos carregá-lo. Atribuímos o descritor ao PCB (process control block) das threads e carregamos o segmento `%gs` usando `load_gs`. Esse carregamento deve ser feito em uma seção crítica para que nada possa nos interromper. O caso `CLONE_SETTLS` funciona exatamente da mesma maneira, apenas o carregamento usando `load_gs` não é realizado. O segmento usado para isso (número do segmento 3) é compartilhado entre processos FreeBSD e processos Linux®, portanto, a camada de emulação do Linux® não adiciona nenhum overhead além do FreeBSD puro.

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 (portanto, nem mesmo os usuários nativos de TLS de 32 bits funcionavam), então tivemos que adicionar esse segmento e implementar seu carregamento em cada troca de contexto (quando uma flag sinalizando o uso de 32 bits é 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 um pouco.

5.4. Futexes

5.4.1. Introdução à sincronização

As threads necessitam de algum tipo de sincronização, o POSIX® fornece alguns mecanismos de sincronização, como mutexes para exclusão mútua, read-write locks para exclusão mútua com uma proporção enviesada de leituras e escritas, e variáveis de condição para sinalizar mudanças de status. É interessante notar que a API de threads POSIX® não oferece suporte para semáforos. A

implementação dessas rotinas de sincronização depende fortemente do tipo de suporte a threading disponível. Em um modelo puro 1:M (userspace), a implementação pode ser feita exclusivamente no espaço de usuário, resultando em uma abordagem rápida e simples (embora as variáveis de condição possam ser implementadas usando sinais, o que pode ser mais lento). Em um modelo 1:1, as threads devem ser sincronizadas usando recursos do kernel, o que pode ser mais lento devido à necessidade de chamadas de sistema. O cenário misto M:N combina as abordagens anteriormente mencionadas ou depende exclusivamente do kernel. A sincronização de threads é uma parte vital da programação com threads, e seu desempenho pode afetar significativamente o programa resultante. Testes recentes no sistema operacional FreeBSD demonstraram um aumento de desempenho de 40% na implementação aprimorada do sx_lock no *ZFS* (que faz uso intensivo de primitivas de sincronização). Embora esse exemplo se refira a operações no kernel, ele destaca a importância de primitivas de sincronização eficientes para o desempenho geral.

Programas com threads devem ser escritos com o mínimo possível de contenção em locks. Caso contrário, em vez de realizar um trabalho útil, a thread apenas espera em um lock. Como resultado disso, os programas com threads bem escritos mostram pouca contenção em locks.

5.4.2. Introdução a Futexes

O Linux® implementa threading 1:1, ou seja, ele utiliza primitivas de sincronização no kernel. Como mencionado anteriormente, programas com threads bem escritos têm pouca contenção em locks. Assim, uma sequência típica pode ser realizada com o aumento/diminuição atômica do contador de referência do mutex, o que é muito rápido, conforme apresentado no seguinte exemplo:

```
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 implementada pelo Linux® 2.6 é chamada de "futexes". Os futexes implementam a verificação de contenção no espaço do usuário e chamam as primitivas do kernel apenas em caso de contenção. Dessa forma, o caso típico ocorre sem qualquer intervenção do kernel. Isso resulta em 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 o endereço do mutex no espaço do usuário, `op` é a operação que estamos prestes a realizar e os outros parâmetros têm significado específico para cada operação.

Futexes implementam as seguintes operações:

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

FUTEX_WAIT

Essa operação verifica se o valor `val` está escrito no endereço `uaddr`. Se não estiver, retorna `EWOULDBLOCK`. Caso contrário, a thread é colocada na fila do futex e é suspensa. Se o argumento `timeout` for diferente de zero, ele especifica o tempo máximo de suspensão. Caso contrário, a suspensão é infinita.

FUTEX_WAKE

Essa operação pega um futex em `uaddr` e acorda os primeiros `val` futexes na fila desse futex.

FUTEX_FD

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

FUTEX_REQUEUE

Essa operação pega `val` threads na fila do futex em `uaddr`, acorda-as e pega `val2` threads seguintes e as coloca novamente na fila do futex em `uaddr2`.

FUTEX_CMP_REQUEUE

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

FUTEX_WAKE_OP

Essa operação realiza uma operação atômica em `val3` (que contém codificado algum outro valor) e `uaddr`. Em seguida, acorda `val` threads no futex em `uaddr` e, se a operação atômica retornar um número positivo, acorda `val2` threads no futex em `uaddr2`.

As operações implementadas em FUTEX_WAKE_OP são:

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



Não há 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 baseada no NetBSD e posteriormente estendida por nós. Ela é implementada 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;
};
```

futex_get / futex_put

Um futex é obtido usando a função `futex_get`, que busca em 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 contador de referência chegar a zero, ele é liberado.

futex_sleep

Quando um futex coloca uma thread em espera, ele cria uma estrutura `working_proc` e a coloca na lista dentro da estrutura do futex. Em seguida, ele executa um `tsleep(9)` para suspender a thread. A suspensão pode ter um tempo limite. Após o retorno do `tsleep(9)` (quando a thread foi acordada ou quando expirou o tempo limite), a estrutura `working_proc` é removida da lista e destruída. Tudo isso é feito na função `futex_sleep`. Se acordamos de um `futex_wake`, `wp_new_futex` é definido para que durmamos nele. Dessa forma, o reenfileiramento real é feito nessa função.

futex_wake

Acordar uma thread que está dormindo em um futex é realizado na função `futex_wake`. Primeiro, nesta função, imitamos o comportamento estranho do Linux®, onde ele acorda N threads para

todas as operações, com a única exceção de que as operações REQUEUE são realizadas em N+1 threads. Mas isso geralmente não faz diferença, pois estamos acordando todas as threads. Em seguida, no loop da função, acordamos n threads e, em seguida, verificamos se há um novo futex para reenfileiramento. Se houver, reenfileiramos até n2 threads no novo futex. Isso coopera com a função `futex_sleep`.

futex_wake_op

A operação `FUTEX_WAKE_OP` é bastante complexa. Primeiro, obtemos dois futexes nos endereços `uaddr` e `uaddr2`, em seguida, realizamos a operação atômica usando `val3` e `uaddr2`. Em seguida, acordamos as threads em espera com valor `val` no primeiro futex e, se a condição da operação atômica for satisfeita, acordamos a thread em espera com valor `val2` (ou seja, `timeout`) no segundo futex.

operação atômica futex

A operação atômica recebe dois parâmetros: `encoded_op` e `uaddr`. A operação codificada codifica a própria operação, o valor de comparação, o argumento da operação e o argumento de comparação. O pseudocódigo para a operação é semelhante a este:

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

E isso é feito atomicamente. Primeiro, é feita uma cópia do número em `uaddr` e, em seguida, a operação é realizada. O código lida com faltas de página e, se nenhuma falta de página ocorrer, `oldval` é comparado com o argumento `cmparg` usando o comparador `cmp`.

Bloqueio Futex

A implementação do futex utiliza duas listas de bloqueio para proteger o `sx_lock` e os bloqueios globais (seja Giant ou outro `sx_lock`). Cada operação é realizada com bloqueio 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 Linux® 2.6.16, as chamadas de sistema *at foram adicionadas. Essas chamadas de sistema (`openat`, por exemplo) funcionam exatamente como suas contrapartes sem o "at", com a pequena exceção do parâmetro `dirfd`. Esse parâmetro altera onde o arquivo fornecido, no qual a chamada de sistema será executada, está localizado. Quando o parâmetro `filename` é absoluto, o `dirfd` é ignorado, mas quando o caminho para o arquivo é relativo, ele entra em jogo. O parâmetro `dirfd` é um diretório relativo ao qual o caminho relativo do arquivo é verificado. O `dirfd` é um descritor de arquivo de algum diretório ou `AT_FDCWD`. Portanto, por exemplo, a chamada de sistema `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
*/
```

Essa infraestrutura é necessária para evitar corridas ao abrir arquivos fora do diretório de trabalho. Imagine que um processo consista em dois threads, thread A e thread B. A thread A chama `open(./tmp/foo/bah., flags, mode)` e antes de retornar, ela é preemptada e a thread B é executada. A thread B não se importa com as necessidades da thread A e renomeia ou remove /tmp/foo/. Temos uma corrida. Para evitar isso, podemos abrir /tmp/foo e usá-lo como `dirfd` para a chamada de sistema `openat`. Isso também permite que o usuário implemente diretórios de trabalho específicos por thread.

A família de syscalls *at do Linux® 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`. Todas essas syscalls são implementadas usando a rotina modificada namei(9) e uma camada de encapsulamento simples.

Implementação

A implementação é feita alterando a rotina namei(9) (descrita anteriormente) para receber um parâmetro adicional `dirfd` em sua estrutura `nameidata`, que especifica o ponto de partida da pesquisa do caminho em vez de usar o diretório de trabalho atual todas as vezes. A resolução do `dirfd` do número de descritor de arquivo para um vnode é feita nas syscalls *at nativas. Quando `dirfd` é `AT_FDCWD`, a entrada `dvp` na estrutura `nameidata` é `NULL`, mas quando `dirfd` é um número diferente, obtemos um arquivo para esse descritor de arquivo, verificamos se esse arquivo é válido e, se houver um vnode associado a ele, obtemos um vnode. Em seguida, verificamos se esse vnode é um diretório. Na própria rotina namei(9), simplesmente substituímos o vnode `dvp` pela variável `dp`, que determina o ponto de partida. A rotina namei(9) não é usada diretamente, mas sim através de uma sequência de diferentes funções em vários níveis. Por exemplo, o `openat` funciona da seguinte maneira:

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

Por esse motivo, `kern_open` e `vn_open` devem ser alterados para incorporar o parâmetro adicional `dirfd`. Nenhuma camada de compatibilidade é criada para essas funções porque não há muitos usuários desse recurso e os usuários existentes podem ser facilmente convertidos. Essa implementação geral permite que o FreeBSD implemente seus próprios syscalls *at. Isso está sendo discutido atualmente.

5.5.2. Ioctl

A interface ioctl é bastante frágil devido à sua generalidade. Devemos ter em mente que os dispositivos diferem entre o Linux® e o FreeBSD, então é necessário ter cuidado para garantir que

a emulação do ioctl funcione corretamente. O tratamento do ioctl é implementado em linux_ioctl.c, onde a função `linux_ioctl` é definida. Essa função simplesmente itera sobre conjuntos de manipuladores de ioctl para encontrar um manipulador que implemente um determinado comando. A chamada de sistema ioctl possui três parâmetros: o descritor de arquivo, o comando e um argumento. O comando é um número de 16 bits, que teoricamente é dividido em 8 bits superiores que determinam a classe do comando ioctl e 8 bits inferiores, 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 possui um comando máximo e um comando mínimo definido, que é usado para determinar qual manipulador será usado. Existem pequenos problemas com essa abordagem porque o Linux® não usa a divisão de conjunto de forma consistente, então às vezes os ioctl de um conjunto diferente estão dentro de um conjunto ao qual não deveriam pertencer (ioctl genéricos do SCSI dentro do conjunto de cdrom, etc.). Atualmente, o FreeBSD não implementa muitos ioctl do Linux® (em comparação com o NetBSD, por exemplo), mas o plano é portá-los do NetBSD. A tendência é usar os ioctl do Linux® mesmo nos drivers nativos do FreeBSD devido à facilidade de portar aplicativos.

5.5.3. Depuração

Cada chamada de sistema deve ser passível de depuração. Para esse propósito, introduzimos uma pequena infraestrutura. Temos a facilidade ldebug, que indica se uma determinada chamada de sistema deve ser depurada (configurável através de um sysctl). Para impressão, temos as macros LMSG e ARGS. Essas macros são usadas para modificar uma string imprimível para mensagens de depuração uniformes.

6. Conclusão

6.1. Resultados

A partir de abril de 2007, a camada de emulação do Linux® é capaz de emular bem o kernel Linux® 2.6.16. Os problemas restantes envolvem futexes, chamadas de sistema inacabadas da família *at, entrega problemática de sinais, falta de suporte a `epoll` e `inotify`, e provavelmente alguns bugs que ainda não foram descobertos. Apesar disso, somos capazes de executar basicamente todos os programas Linux® incluídos na Coleção de Ports do FreeBSD com o Fedora Core 4 no kernel 2.6.16, e existem alguns relatos rudimentares de sucesso com o Fedora Core 6 no kernel 2.6.16. O linux_base do Fedora Core 6 foi recentemente adicionado, permitindo testes adicionais da camada de emulação e fornecendo mais informações sobre onde devemos concentrar nossos esforços na implementação do que está faltando.

Somos capazes de executar os aplicativos mais usados, como o pacote `www/linux-firefox`, o pacote `net-im/skype` e alguns jogos da Coleção de Ports. Alguns desses programas apresentam comportamento inadequado sob a emulação do 2.6, mas isso está atualmente em investigação e esperamos que seja corrigido em breve. O único grande aplicativo conhecido por não funcionar é o Java™ Development Kit do Linux®, devido ao requisito da facilidade `epoll`, que não está diretamente relacionada ao kernel do Linux® 2.6.

Esperamos habilitar a emulação do 2.6.16 como padrão algum tempo depois do lançamento do FreeBSD 7.0, pelo menos para expor as partes de emulação 2.6 para um teste mais amplo. Uma vez

feito isso, podemos mudar para o linux_base do Fedora Core 6, que é o plano final.

6.2. Trabalho futuro

O trabalho futuro deve se concentrar em corrigir os problemas restantes com futexes, implementar o restante da família de chamadas de sistema *at, corrigir a entrega de sinais e possivelmente implementar as facilidades `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.

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 estar completo e não foi realmente testado. O DragonflyBSD mostrou interesse em portar as melhorias do 2.6.

Em geral, à medida que o Linux® se desenvolve, gostaríamos de acompanhar o seu desenvolvimento, implementando as novas chamadas de sistema adicionadas. O `splice` é uma delas que vem à mente em primeiro lugar. Algumas chamadas de sistema já implementadas também podem ser aprimoradas, por exemplo, `mremap` e outras. Também podem ser feitas melhorias de desempenho, como bloqueio 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. Neville-Neil. Design and Implementation of the FreeBSD operating system. Addison-Wesley, 2005.
2. https://tldp.org
3. https://www.kernel.org