Tutorial

Adicionando novas chamadas de sistema ao *kernel* Linux

Chamadas de sistema

Uma chamada de sistema é uma chamada para o *kernel* afim de executar uma função específica, um serviço, que controla um dispositivo ou executar uma instrução privilegiada. Pelo fato do serviço ser fornecido no *kernel*, uma chamada direta não pode ser realizada. Ao invés disso, deve-se cruzar a fronteira aplicação/*kernel* de uma forma específica. A forma de fazer isso difere de uma arquitetura para outra.

Neste tutorial estaremos restritos à arquitetura x86, no entanto, vamos discutir brevemente as diferenças de como sistemas x86 de 32 bits (x86-32) e de 64 bits (x86-64) realizam uma chamada de sistema.

APIs e ABIs

Programadores estão, naturalmente, interessados em garantir que seus programas executem em todos os sistemas que eles prometeram suportar, agora e no futuro. Eles querem se sentir seguros de que os programas que escreveram em suas distribuições Linux vão executar em outras ditribuições, assim como em outras arquiteturas suportadas pelo Linux e, ainda, em outras novas (ou antigas) versões do Linux.

Ao nível do sistema, há dois conjuntos de definições e descrições que impactam na portabilidade. Uma é a API (*Application Programming Interface*) e outra é a ABI (*Application Binary Interface*). Ambas definem e descrevem as interfaces entre as várias peças do software de um computador.

Um exemplo real de API são as interfaces definidas pelo padrão C e implementadas pela biblioteca padrão C (*standard C library*). Esta API define uma família de funções básicas e essenciais, tais como gerenciamento de memória, processos, arquivos, manipulação de cadeias de caracteres (*strings*) e outras.

Se uma API define uma interface de código fonte (*source interface*), uma ABI define a interface binária entre duas ou mais peças de software em uma arquitetura particular. Ela define como uma aplicação interage com ela mesma, como interage com o *kernel*, e como interage com as bibliotecas. Enquanto uma API garante compatibilidade de fonte, uma ABI garante compatibilidade binária, garantindo que uma peça de código objeto funcionará em qualquer sistema com a mesma ABI, sem necessidade de recompilação.

ABIs se preocupam com questões tais como: convenções de chamada, ordenação de bytes, uso de registradores, invocação de chamadas de sistema, *linking*, comportamento de bibliotecas e formato do binário (executável). As convenções de chamada, por exemplo, definem como funções são chamadas, como seus argumentos são passados, quais registradores são preservados e como o chamados recupera o valor retornado pela função.

A despeito das muitas tentativas de se definir uma única ABI para muitas arquiteturas através de vários sistemas operacionais (particularmente para i386 em sistemas Unix), esses esforços não obtiveram muito sucesso. Ao contrário, sistemas operacionais – incluindo o Linux – tendem a definir suas próprias ABIs. A ABI está intimamente ligada à arquitetura; a grande maioria das ABIs usam conceitos específicos da arquitetura, tais como certos registradores e instruções assembly. Assim, cada arquitetura de máquina tem sua própria ABI no Linux. De fato, costuma-se uma certa ABI pelo nome da arquitetura da máquina, como Alpha ou x86-64. Assim, a ABI é função

tanto do sistema operacional como da arquitetura.

Programadores de sistema devem estar cientes da ABI mas, geralmente, não precisam memorizá-la. A ABI é imposta pela cadeia de ferramentas (*tollchain*) – o compilador, o *link-editor* assim por diante. No entanto, o conhecimento da ABI pode levar a uma programação otimizada e é necessário quando se escreve código *assembly* ou no desenvolvimento da própria cadeia de ferramentas.

A interface de chamadas de sistema no Linux

Não é possível que aplicações de usuário executem código do *kernel* diretamente. Elas não podem simplesmente realizar uma chamada para uma função que existe no *kernel* porque o *kernel* existe em um espaço de memória protegido.

Ao contrário, a aplicação do usuário deve sinalizar o *kernel*, de alguma forma, que ela quer executar uma chamada de sistema, que irá chavear o processador para modo *kernel* e executar em espaço de *kernel* a função a pedido da aplicação de usuário.

O mecanismo para sinalizar o *kernel* é uma interrupção de software. Esta provocará o chaveamento para modo *kernel* e executar o manipulador de chamadas de sistema (*system call handler*). A interrupção de software definida em arquiteturas x86 é a de número 0x80. O manipulador de chamadas de sistema é a função `system_call()`. Ela é dependente de arquitetura. Em arquiteturas x86-64 ela é implementada em *assembly* e pode ser encontrada em `arch/x86/kernel/entry_64.S`.

No entanto, processadores posteriores ao Pentim II (*confirmar*) – inclusive os da AMD – adicionaram instruções conhecidas como `sysenter/sysexit`. Esta fornece um mecanismo mais rápido e mais especializado de mudar para modo *kernel* e executar uma chamada de sistema do que a instrução mencionada acima (`int 0x80`). Maiores detalhes sobre este novo mecanismo pode ser encontrado em [7], [8] e [9].

Uma chamada de sistema é um ponto de entrada para o *kernel* Linux. Usualmente, chamadas de sistema não são invocadas diretamente. Ao contrário, a maioria delas tem funções *wrapper* na biblioteca padrão C (glibc) como ilustra a Figuras 1, abaixo.

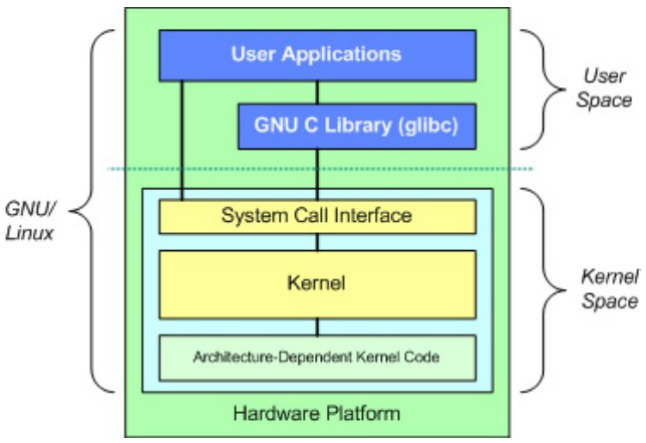


Figura 1 - A arquitetura fundamental do sistema operacional GNU/Linux

Na maioria dos casos, a função *wrapper* realiza não mais do que as seguintes ações:

- copiar argumentos e o número único da chamada de sistema para registradores onde o *kernel* os espera encontrar;
- mudar para modo *kernel*, quando o trabalho real da chamada é realizado pelo *kernel*
- definir `errno` se a chamada de sistema retorna um número de erro quando o *kernel* retorna a CPU para modo usuário.

No entanto, em algumas situações, como quando queremos adicionar uma nova chamada de sistema, não haverá uma função *wrapper* em glibc para manipular seu lançamento e seu retorno. Nesse casos, podemos utilizar a função de biblioteca `syscall()`. Como parâmetros, deve-se passar o **número único** da chamada de sistema e seus argumentos, se houver. O que esta função executa internamente é, basicamente, o que uma função *wrapper* executa, como descrito acima.

A Figura 2, abaixo, mostra um exemplo do encadeamento de ações desde o momento em que uma aplicação de usuário chama a função read(), passando pela função *wrapper* e pelo manipulador de chamadas no *kernel*.

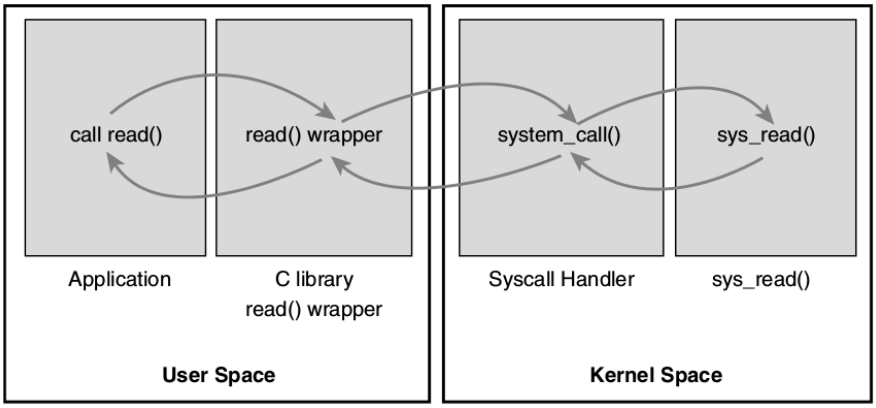


Figura 2 - Etapas da invocação da chamada de sistema read()

Como mencionado acima, no Linux a cada chamada de sistema é associado um número, que é usado para referenciá-la. Quando um processo de usuário executa uma chamada de sistema, é o número dessa chamada que a identifica. O processo não a referencia através de seu nome. O *kernel* mantém uma lista de todas as chamadas de sistema registradas em uma 'tabela'. Em uma arquitetura x86-64, esta tabela pode ser encontrada em `arch/x86/syscalls/syscall_64.tbl`.

Indicando a chamada de sistema corretamente

Simplesmente passar para modo *kernel* através do mecanismo de interrupção de software para acionar o manipulador de chamadas de sistema não é suficiente, já que existem muitas chamadas de sistema e todas elas adentram o *kernel* da mesma forma. Por isso, o número da chamada de sistema precisa ser passado para o *kernel*. Em arquiteturas x86, este número é fornecido através do registrador `eax`. Outras arquiteturas fazer isso de forma parecida.

A função `syscall()` verifica a validade desse número consultando a tabela (por exemplo, `syscall_64.tbl`, em arquiteturas x86-64).

Passagem de parâmetros

Adicionalmente ao número da chamada de sistema, a maioria das funções precisam passar um ou dois parâmetros. De alguma forma, o espaço do usuário deve transmitir os parâmetros para o *kernel* durante o *trap*. A forma mais simples de fazer isso é a mesma usada para o número da chamada, isto é, utilizando registradores para receber os parâmetros. Em arquiteturas x86-32, os registradores ebx, ecx, edx, esi e edi recebem, na ordem, os primeiros cinco argumentos. No caso de ser necessário passar mais de cinco argumentos na chamada, um único registrador é usado para manter um ponteiro para o espaço do usuário armazenando todos os argumentos.

O valor de retorno é enviado para o espaço do usuário também via registrador. Em arquiteturas x86-32, este retorno é escrito para o registrador eax.

Contexto da chamada de sistema

Durante a execução de uma chamada de sistema, o *kernel* está em **contexto de processo**. Isto quer dizer que o ponteiro `current` aponta para a tarefa (*task*) atual, isto é, neste caso, o processo de usuário que disparou a chamada de sistema.

Em contexto de processo, o *kernel* é capaz de 'dormir' (por exemplo, se a chamada de sistema bloqueia por algum motivo ou explicitamente chama `schedule()`) e é completamente 'preemptável'.

O fato de que o contexto de processo é preemptável implica que, assim como no espaço de usuário, a tarefa atual pode ser preemptada por outra tarefa. Pelo fato de a nova tarefa poder também executar a mesma chamada de sistema, cuidados devem ser tomados para garantir que chamadas de sistema sejam **reentrantes**.

Kernel reentrante

Todo *kernel* Unix é reentrante, incluindo o Linux. Isto significa que vários processos podem estar executando em modo *kernel* ao mesmo tempo. Em processadores de um único núcleo, claro, apenas um processo progride de cada vez, mas muitos deles podem estar bloqueados em modo *kernel* esperando pela CPU ou por alguma operação de E/S.

Uma forma de prover reentrância é escrever funções reentrantes – chamadas de sistema são funções implementadas dentro do *kernel*.

Função é reentrante é uma função que pode ser usada por mais de uma tarefa concorrentemente sem perigo de se obter dados corrompidos. Ao contrário, uma função não-reentrante é uma função que não pode ser compartilhada por mais de uma tarefa, a menos que exclusão mútua à esta função seja garantida, ou usando semáforos ou desabilitando interrupções durante as sessões críticas do código. Uma função reentrante pode ser interrompida em qualquer instante e continuada em momento posterior sem perda de dados. Funções reentrantes ou utilizam variáveis locais ou protegem seus dados quando variáveis globais são usadas.

Uma função reentrante:

- não mantém dados estáticos (*static*) em chamadas sucessivas;
- não retorna um ponteiro para dados estáticos; todo os dados são fornecidos pelo chamador da função;
- utiliza dados locais ou garante proteção de dados globais fazendo uma cópia local deles;
- não devem chamar nenhuma função não-reentrante

Se acontece uma interrupção de hardware, um *kernel* reentrante é capaz de suspender o processo em execução, ainda que este processo esteja em modo *kernel*. Esta capacidade é muito importante, pois aumenta o *throughput* de controladores de dispositivos que disparam interrupções. Uma vez que um dispositivo tenha disparado uma interrupção, ele espera que a CPU reconheça o recebimento desta. Se o *kernel* consegue responder rapidamente, o controlador do dispositivo poderá realizar outras tarefas enquanto a CPU trata a interrupção.

Impacto da reentrância na organização do *kernel*

Um 'caminho de controle' do *kernel* (*control path*) representa uma sequência de instruções executadas pelo *kernel* para tratar de uma chamada de sistema, um exceção ou uma interrupção.

No caso mais simples, a CPU executa um caminho de controle sequencialmente, da primeira instrução até a última. No entanto, quando ocorre um dos eventos seguintes, a CPU interpõe (*interleaves*) os caminhos de controle.

- Um processo executando em modo usuário faz uma chamada de sistema e o *kernel* verifica que a requisição não pode ser satisfeita imediatamente. Ele, então, chama o escalonador para

selecionar um outro processo para executar. Como resultado, ocorre um chaveamento de processos. O caminho de controle inicial é deixado inacabado e a CPU reativa a execução de um outro caminho de controle. Nesse caso, os dois caminhos de controle são executados em nome de dois processos diferentes.

- A CPU detecta uma exceção – por exemplo, um acesso a uma página que não está presente na memória – durante a execução de um caminho de controle do *kernel*. O primeiro caminho de controle é suspenso e a CPU inicia a execução de um procedimento adequado. Quanto este termina, o primeiro caminho de controle continua. Nesse caso, os dois caminhos de controle são executados em nome do mesmo processo.

- Uma interrupção de hardware acontece enquanto a CPU está executando um caminho de controle do *kernel* com interrupções habilitadas. O primeiro caminho de controle é deixado inacabado e a CPU inicia um outro caminho de controle para tratar a interrupção. Quanto a interrupção termina, o primeiro caminho de controle é reativado. Nesse caso, os dois caminhos de controle executam no contexto do mesmo processo e o tempo gasto pelo sistema é contabilizado para ele. No entanto, o tratador da interrupção não opera, necessariamente, em nome deste processo.

Cada processo executa em seu espaço de endereçamento privado. Um processo executando em modo usuário, referencia áreas privadas de código, dados e pilha. Quando executando em modo *kernel*, o processo endereça áreas de código e dados do *kernel*, além de utilizar uma outra pilha. Como o *kernel* é reentrante, vários caminhos de controle de *kernel* – cada um relacionado diferentes processos – podem ser executados de uma vez. Assim, cada caminho de controle referencia sua própria pilha privada no *kernel*.

Implementar um *kernel* reentrante requer o uso de sincronização: se um caminho de controle é suspenso durante a utilização de uma estrutura de dados no *kernel*, nenhum outro caminho de controle terá permissão para utilizar a mesma estrutura de dados, a menos que esta tenha sido deixada em um estado consistente. De outra forma, a interação dos dois caminhos de controle poderá corromper as informações armazenadas.

Chamadas de sistema bloqueantes

Algumas chamadas de sistema retornam rapidamente após executarem seu trabalho. Este é o caso de, por exemplo, consultar o relógio do sistema ou obter o identificador de usuário (id). No entanto, muitas chamadas de sistema não retornam imediatamente. Nesse caso, para otimizar a utilização do processador, o processo que fez a chamada não pode prosseguir e, portanto, deve ser bloqueado. Assim, o escalonador é acionado para que outro processo possa ocupar o processador.

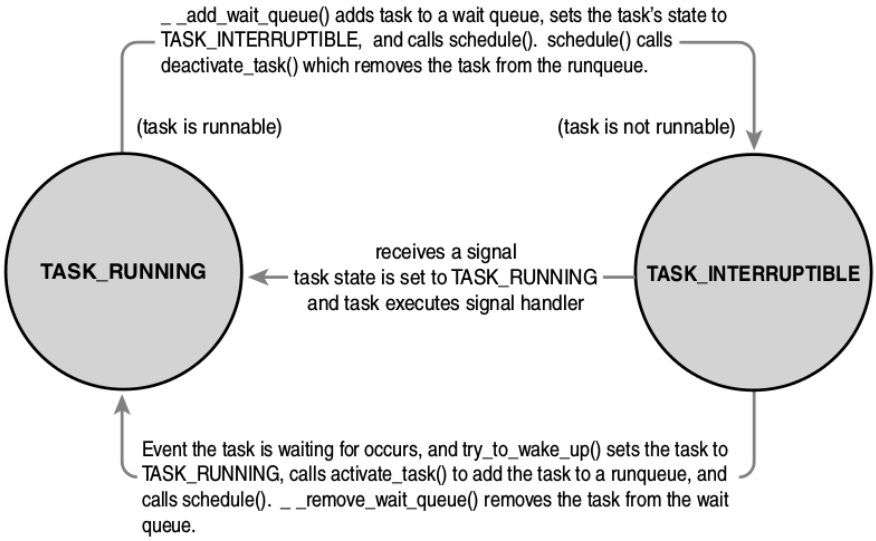
Exemplos de chamadas de sistema bloqueantes:
- read()/write() em um aquivo no disco
- read() em um buffer vazio
- write() em um buffer cheio
- fork() - cria um processo filho – implica em alocação de memória
- mmap() - mapeia um arquivo na memória – implica em alocação de novas páginas

Tarefas que estão bloqueadas (*sleeping*) estão em um estado especial não-executável. Isto é importante porque, sem este estado, o escalonador selecionaria tarefas que não podem ser executadas. Uma tarefa bloqueia por uma variedade de razões, mas sempre que estiver esperando por algum evento. Uma tarefa também pode involuntariamente dormir quando tenta obter semáforo ocupado no *kernel*. Um motivo comum para dormir é uma operação de E/S em um arquivo – por exemplo, a tarefa faz um read() em um arquivo que precisa ser lido do disco. Um outro exemplo, a tarefa pode estar esperando por uma entrada no teclado. Seja qual for o caso o

comportamento do *kernel* é o mesmo: a tarefa se marca como 'dormindo' (bloqueada), se auto insere em uma fila de espera (*wait queue*), se auto remove da árvore *red-black* de executáveis e chama o escalonador para selecionar um novo processo para executar. Ao voltar do bloqueio acontecem as ações contrárias: a tarefa é marcada como executável, removida da fila de espera e adicionada à árvore *red-black*.

Quando no estado bloqueado, uma tarefa pode estar em um de dois estados: TASK_UNINTERRUPTIBLE ou TASK_INTERRUPTIBLE. Eles diferem pelo fato de que, no primeiro, a tarefa ignora sinais, enquanto que no segundo, a tarefa acorda prematuramente e responde a um sinal que receber. Ambos os tipos de tarefas bloqueadas residem em uma fila de espera, esperando um evento ocorrer e não são executáveis.

A figura abaixo ilustra a dinâmica da mudança de estados de uma tarefa quando há um bloqueio e o momento e a razão de sua volta para o estado executável.



Adicionando uma nova chamada de sistema

Para adicionar uma nova função ao seu sistema, isto é, no *kernel*, você vai precisar dos códigos fonte do seu sistema. Assim, você deve baixá-los e descompactá-los em uma pasta a sua escolha ou em /usr/src, como fizemos neste tutorial.

Além disso, ao final do processo, será necessário recompilar o *kernel*. Por isso, você também vai precisar das ferramentas específicas para tal tarefa. Um bom guia para estes dois requisitos – baixar os fontes, compilar e instalar o novo *kernel* – considerando a distribuição Ubuntu, pode ser encontrado em [10].

Os passos abaixo descrevem como inserir uma chamada de sistema em *kernel*s 2.6 e superiores, considerando arquitetura x86-64.

Além disso, todos os caminhos de diretório tomarão como referência o local onde residem os fontes descompactados, em nosso caso: `/usr/src/linux`.

Inicialmente precisamos associar um número para nossa chamada de sistema na tabela. Esta pode ser encontrada em `arch/x86/syscalls/syscall_64.tbl`.

O formato deste arquivo é mostrado abaixo:

```
<number> <abi> <name> <entry point>
```

Onde:

`<number>`: o número da sua chamada de sistema deverá ser o primeiro após o último número da lista. Se a última chamada na lista é a de número 314, então, sua nova chamada de sistema terá número 315.

`<abi>`: a ABI a ser usada ('64', 'x32', 'common'). Escolha 'common' para que sua chamada de sistema nova seja compilada com suporte para as duas ABIs.[1]

`<name>`: o nome de sua chamada de sistema

`<entry point>`: é o nome da função a ser chamada a fim de tratar (*handle*) a chamada. Por convenção o nome para esta função é formado adicionando-se o prefixo `sys_` ao nome da sua chamada definido em <name>.

Neste exemplo, minha nova chamada de sistema vai se chamar myothersc e seu número será 314, já que a última da lista em minha tabela era a de número 313. Assim, a linha a ser inserida ao final da lista fica assim:

```
314   common    myothersc       sys_myothersc
```

Agora que temos a nova chamada de sistema na tabela, temos que fornecer um protótipo para a função '`entry point`'. Isto é feito no arquivo `include/linux/syscalls.h` Edite este arquivo de forma a inserir uma nova linha, ou seja, o protótipo, ao final do arquivo.

O protótipo para nossa função ficará assim:

```
asmlinkage long sys_myothersc(int i);
```

A parte curiosa dessa linha é o marcador `asmlinkage`. Esta é uma macro que instrui o compilador gcc de que a função deve esperar que seus argumentos sejam colocados na pilha ao invés de registradores.

Nesse ponto, estamos prontos para de fato escrevermos nosso chamada de sistema. Para isso, criei um novo arquivo em `/kernel/myothersc.c`

---

1 Como não encontrei informação sobre 'common' enviei uma mensage para kernelnewbies.com. A cópia da pergunta e da resposta estão abaixo.

Date: Thu, 20 Nov 2014 13:52:01 -0500
From: Valdis.Kletnieks@vt.edu
Subject: Re: ABI choice in adding new syscall
To: Marcelo Silva Freitas <marcelo.caj.ufg@gmail.com>
Cc: kernelnewbies@kernelnewbies.
Message-ID: <31544.1416509521@turing->
Content-Type: text/plain; charset="us-ascii"

On Thu, 20 Nov 2014 16:44:25 -0200, Marcelo Silva Freitas said:

> I am studing how to add new syscall to the kernel.
> When editing the file syscall_64.tlb there are 3 choices for the ABI: 64,
> common, x32
> I have read about and I don't found clearly the meaning and effect of
> 'common' choice.
> In what imply this choice?

I'm pretty sure it means "this ABI works for both 32 and 64 bit invocations",
which is a good design feature to have if you can do it. That way, your syscall
code doesn't have to jump through ugly hoops depending on whether the
userspace program was in 32 or 64 bit mode - we've had a *lot* of bugs
relating to that.

O conteúdo do arquivo é mostrado abaixo:

```
#include <linux/linkage.h>
#include <linux/kernel.h>

asmlinkage long sys_myothersc(int i) {

   printk("myothersc: Inteiro fornecido: %d\n", i);
   printk("myothersc: Somei 10 a este inteiro.\n");
   return( (long)(i+10) );

}
```

Esta função simplesmente recebe um inteiro, imprime duas mensagens e retorna o resultado da soma do valor de entrada com a constante 10.

Vale observar que as mensagens não serão visualizadas na saída padrão pois a função `printk()` é utilizada. Esta é equivalente à função `printf()` e o que ela imprime para um *buffer* de mensagens do *kernel* e pode ser vista com o comando `dmesg`.

Agora que escrevemos o código para nossa chamada de sistema, o último passo é inserir nosso novo arquivo em um Makefile, de forma que ele possa ser ligado (*linked*) ao *kernel* junto com todos os outros arquivos objeto após a compilação. Como criamos nosso arquivo `myothersc.c` no diretório */kernel*, temos que modificar o arquivo `/kernel/Makefile`.

Para tal, edite este arquivo Makefile e observe que, logo no início, há uma linha que começa com:

```
obj-y       = fork.o  exec_domain.o  panic.o \
            …
            …
```

Ao final dessa lista de nomes de arquivo com extensão .o, adicionamos o nosso:

… `myothersc.o`

Estas são as modificações que precisam ser feitas.

Agora você deve compilar seu novo *kernel*, instalá-lo e reiniciar o sistema com ele.

Após a reinicialização, vamos checar se o número da nossa chamada de sistema foi gerado.

Para isso, vá para o diretório `/usr/src` e digite o comando:

```
grep -nR __NR_myothersc
```

Ele pesquisa recursivamente os diretórios e mostra as linhas dos arquivos onde acha o padrão `__NR_myothersc`. Você deve encontrá-la no arquivo `unistd_64.h`, entre outros.

Por fim, para testar nossa nova chamada de sistema precisamos criar um programa que a chame. Testamos a nossa com o seguinte código:

```
#include <stdio.h>
#include <sys/syscall.h>
#include <asm/unistd_64.h>
#include <string.h>
#include <errno.h>

int main(void) {
 long ret;
 printf("\n\nDescendo para o nivel do kernel");
 if( (ret=syscall(__NR_myothersc, 1000)) == -1) {
     fprintf(stderr, "\nErro ao chamar myothersc: %s\n", strerror(errno));
```

```
    return 1;
  }
 printf("\nValor retornado de myothersc: %ld\n", ret);
 printf("\nVoltando para o nivel do usuario\n\n");
 return 0;
}
```

A este arquivo demos o nome de `teste_myothersc.c`
Compile com o comando:

```
gcc -o teste_myothersc teste_myothersc.c
```

Se necessário, inclua '`-I /usr/src/linux/usr/include/`'. Onde 'linux' deve ser substituído pelo nome da pasta onde estão os códigos fonte que você baixou no início.
Utilize o comando abaixo para ver as linha impressas pela função `printk()`:

```
dmesg | grep myothersc
```

Chamadas de sistema: prós e contras

Este tutorial mostrou que não é complicado implementar uma nova chamada de sistema, mas isso não deve, de forma alguma, encorajá-lo a criar uma. Vale como estudo sobre o funcionamento do *kernel* e sua interface com o mundo do usuário. Frequentemente, existem alternativas muito mais viáveis do que criar uma nova chamada de sistema.

Vejamos os prós e os contras, além de alternativas.

Prós:

- Chamadas de sistema são fáceis de implementar e de usar.
- O desempenho de chamadas de sistema no Linux é excelente.

Contras:

- Você precisa de um número para sua chamada que precisa, oficialmente, ser reservado
- Uma vez que seu número de chamada estiver na tabela de um *kernel* estável, ela está 'escrita na pedra'. Mudar implica em quebrar aplicações do espaço do usuário
- Chamadas de sistema precisam ser suportadas por cada arquitetura
- Para alterar uma chamada de sistema é necessário recompilar o *kernel*

Alternativas:
Módulos: podem ser carregados e descarregados a qualquer tempo e não necessitam de recompilação quando alterações forem feitas.

Bibliografia

1. Jones, M. T. *kernel command using Linux system calls*, 2010. Disponível em: http://www.ibm.com/developerworks/linux/library/l-system-calls/
2. Love, R. *Linux kernel Development*, 3ed., Addison-Wesley, 2010
3. Love, R. *Linux System Programming: Talking Directly to the kernel and C Library*, 2ec., O'Reilly, 2013
4. *intro(2), Introduction to System Calls*, Linux Programmer's Manual. Disponível em:

http://man7.org/linux/man-pages/man2/intro.2.html
5. *syscalls(2), Linux system calls*, Linux Programmer's Manual. Disponível em: http://man7.org/linux/man-pages/man2/syscalls.2.html
6. Arora, H. *Linux System Calls*, Blog Real World Linux, 2012. Disponível em: https://www.ibm.com/developerworks/community/blogs/58e72888-6340-46ac-b488-d31aa4058e9c/entry/linux_system_calls20?lang=en
7. Brouwer, A. *The Linux kernel*, 2003. Disponível em: http://www.win.tue.nl/~aeb/linux/lk/lk.html#toc1
8. Drysdale, D. *Anatomy of a system call – part 1*, 2014. Disponível em: http://lwn.net/Articles/604287/
9. Drysdale, D. *Anatomy of a system call – part 1*, 2014. Disponível em: http://lwn.net/Articles/604515/
10. *GitkernelBuild*, Ubuntu Wiki. Disponível em: https://wiki.ubuntu.com/*kernel*Team/Git*kernel*Build
11. Tully, S. *Adding a Syscall to Linux 3.14*, 2014. Disponível em: https://shanetully.com/2014/04/adding-a-syscall-to-linux-3-14/
12. Park, C. H. *Adding a System call for Linux 3.10 x86_64*, 2014. Disponível em: http://heartinpiece.blogspot.com.br/2014/01/adding-system-call-for-linux-310-x8664.html
13. Jha, D. *Use reentrant functions for safer signal handling*, IBM, 2005. Disponível em: http://www.ibm.com/developerworks/library/l-reent/
14. Bovet, Daniel P., Cesati, M. Understanding the Linux Kernel, 3 ed., O'Reilly, 2008