Peter Senna Tschudin
Red Hat Certified Engineer
peter@parahard.com
06/11/2006

v1.2

Sua Primeira Modificação do Kernel

Introdução:
Este documento é uma receita para criar duas chamadas de sistema e uma kernel_thread. O objetivo é mostrar que modificar o kernel é possível, não muito complicado e muito poderoso. Inicialmente eu proponho um problema que será resolvido passo a passo. Ao final eu proponho outro problema. Conhecimentos básicos de Linux e uma boa noção de C são desejáveis.

O problema resolvido:
Criar duas chamadas de sistema:

int pst_monitor (int PID) : recebe um inteiro como parâmetro. Verifica se este inteiro equivale ao PID de um processo em execução. Caso afirmativo, armazena em uma estrutura de dados alocada no espaço do kernel.

int pst_notify (int PID) : recebe um inteiro como parâmetro. Verifica se este inteiro já foi adicionando a estrutura de dados por pst_monitor (PID). Caso afirmativo, relaciona o parâmetro int PID com o número do processo que chamou pst_notify (PID).

Criar uma kernel_thread que percorre a estrutura de dados, periodicamente com pausas de um segundo, em busca de processos que não estão mais executando e mata os processos que chamaram notify (PID).

Sopa de letras:
Caso não se sinta confortável com os termos como chamadas de sistema, PID e kernel_thread, a leitura das obras abaixo vai ajudar:

- Understanding the Linux Kernel - http://www.oreilly.com/catalog/linuxkernel/chapter/ch10.html
- Linux Device Drivers, Third Edition - http://lwn.net/Kernel/LDD3/

Cuidado!
Todas as instruções deste documento foram elaboradas tendo em mente que você esta usando um ambiente de teste em que não existem aplicativos e arquivos importantes. Todos os comandos devem ser executados como root. Mas executar o rpmbuild como root pode ser catastrófico se algo sair errado. Então não siga esses passos em ambiente de produção. Para saber se o ambiente que você tem atende aos requisitos, você tem que responder "SIM!" a pergunta: Posso formatar seu ambiente de teste e apagar todos os arquivos?

PARTE 1: Preparação

Ambiente de teste:
Recomendo que você use uma maquina virtual para fazer o desenvolvimento. Minha preferência é o VMWare Workstation.
As principais vantagens de se usar uma maquina virtual são:

- Segurança: Apenas a maquina virtual corre o risco de ser danificada por eventuais erros
- Simplicidade: Você não precisa nem ter o Linux instalado no seu computador, a sua maquina virtual pode rodar dentro do Windows.
- Agilidade: Você vai precisar reiniciar várias vezes o seu ambiente de teste. Fazer isso em uma maquina virtual é mais rápido do que em um computador.

Porém o uso da maquina virtual é apenas uma recomendação. Uma instalação tradicional de Linux em um computador doméstico ou notebook é perfeita(Desde que a resposta a minha pergunta ainda seja SIM!).

Caso você não tenha uma cópia do VMware Workstation, faça o download gratuito do VMware Server e do VMware Server Console no endereço:
http://www.vmware.com/download/server/

Requisitos de hardware:

	Com Maquina Virtual	Instalação Convencional
Memória	512 MB	256 MB
Processador	O mais rápido disponível	O mais rápido disponível
Espaço em disco	10GB	10GB

Compilar o kernel pode ser bem demorado, por isso use o processador mais rápido que estiver a sua disposição. A estimativa de espaço em disco inclui o espaço para o Linux.

Distribuição:
Para evitar problemas de compatibilidade com as instruções, recomendo que você utilize o Fedora Core 5 sem atualizações. Faça o download dele em:
http://bart.las.ic.unicamp.br/pub/fedora/linux/core/5/i386/iso/

Código fonte do kernel:
O código fonte do kernel usado foi a versão 2.6.15-1.2054_FC5. Você pode fazer o download pelo link:
http://www.las.ic.unicamp.br/pub/fedora/linux/core/5/source/SRPMS/kernel-2.6.15-1.2054_FC5.src.rpm

Porém, tanto o uso do Fedora Core 5 sem atualizações, quanto o uso da versão 2.6.15-1.2054_FC5 do kernel são apenas recomendações. É necessário apenas que a versão do kernel seja superior a 2.6.

Instalando o Linux no ambiente de teste:
Caso opte por instalar o Fedora Core 5 dentro do VMware, no momento de criar a

maquina virtual, escolha IDE em Virtual Disk Type ao invés de SCSI (Recommended).

Durante a instalação do Fedora 5, a resposta padrão é suficiente para a maioria das perguntas. É necessário apenas customizar os pacotes a serem instalados. Desmarque a opção "Office and Productivity" e marque "Software Development". É recomendável que o firewall e o SElinux sejam desabilitados.*

* Esta recomendação é válida apenas para ambientes de teste onde não há previsão para serviços de rede. Mantenha sempre o SELinux e o firewall ativos.

Preparando o kernel para desenvolvimento:
Após o término da instalação do Linux, é necessário instalar o código fonte do kernel do Linux e prepará-lo para desenvolvimento.

O primeiro passo é fazer o download do rpm do código fonte do kernel. Execute os comandos:

```
cd /tmp
wget http://www.las.ic.unicamp.br/pub/fedora/linux/core/5/source/SRPMS/kernel-
2.6.15-1.2054_FC5.src.rpm
```

O próximo passo é instalar o RPM. Execute os comandos:

```
cd /tmp
rpm -Uvh  kernel-2.6.15-1.2054_FC5.src.rpm
```

Agora precisamos preparar o código fonte para desenvolvimento:

```
cd /usr/src/redhat/SPECS
rpmbuild -bp --target $(uname -m) kernel-2.6.spec
```

O código fonte esta disponível em:
/usr/src/redhat/BUILD/kernel-2.6.15/linux-2.6.15.i686 *

Para simplificar, vamos utilizar o diretório /opt/linux. Execute os comandos:

```
mv /usr/src/redhat/BUILD/kernel-2.6.15/linux-2.6.15.i686 /opt
mkdir /opt/linux-2.6.15.i686-mod
cp -R /opt/linux-2.6.15.i686/* /opt/linux-2.6.15.i686-mod
ln -s /opt/linux-2.6.15.i686-mod/ /opt/linux
```

*Se a arquitetura do ambiente de teste for diferente de i686, faça as correções necessárias.

Testando o código fonte do kernel:
Antes de modificar o kernel, vamos testar nossa preparação do código fonte. Os comandos abaixo compilam e instalam o kernel que acabamos de preparar. É necessário modificar dois arquivos, /opt/linux/Makefile e /etc/grub.conf. Nos dois casos as modificações são mínimas e aparecem entre ( ).

```
cd /opt/linux
vi Makefile (EXTRAVERSION = -mod)
make clean
make oldconfig
make bzImage
make modules
make modules_install
make install
vi /etc/grub.conf (default=0)
```

Reinicie seu ambiente de teste e observe no grub "Fedora Core (kernel 2.6.15-mod)". O boot deve acontecer normalemte. Após reinicar execute o comando uname -r para verificar qual versão do kernel foi iniciada.

PARTE 2: Modificando o kernel

Criando chamadas de sistema e kernel_thread:

Criar chamadas de sistema é simples. É necessário modificar alguns arquivos do kernel e criar um arquivo contendo o código das chamadas de sistema. Para usar as chamadas de sistema, é necessário criar uma biblioteca para ser importada pelos programas que vão fazer as chamadas de sistema.

Vamos modificar os arquivos:

/opt/linux/arch/i386/kernel/syscall_table.S

```
.long sys_unshare       /* 310 */
.long sys_pst_monitor   /* NOSSA SYSCALL 1 */
.long sys_pst_notify    /* NOSSA SYSCALL 2 */
```

/opt/linux/arch/i386/kernel/Makefile

```
obj-y    := process.o semaphore.o signal.o entry.o traps.o irq.o
ptrace.o time.o ioport.o ldt.o setup.o i8259.o sys_i386.o pci-dma.o
i386_ksyms.o   i387.o   dmi_scan.o   bootflag.o   quirks.o   i8237.o
topology.o pst_syscalls.o
```

/opt/linux/include/asm-i386/unistd.h

```
#define __NR_pst_monitor          311
#define __NR_pst_notify           312

#define NR_syscalls 313
```

/opt/linux/init/main.c

```
//após extern void prepare_namespace(void);
extern void pst_init(void);

//após acpi_early_init();
pst_init();
```

Vamos criar os arquivos:

/opt/linux/arch/i386/kernel/pst_syscalls.c

Veja anexo 1 ou:
http://www.parahard.com/peter/syscalls01/opt/linux/arch/i386/kernel/pst_syscalls.c

/opt/linux/include/linux/pst_syscalls.h

Veja anexo 2 ou:
http://www.parahard.com/peter/syscalls01/opt/linux/include/linux/pst_syscalls.h

/usr/include/linux/pst_syscalls.h

Veja anexo 3 ou:
http://www.parahard.com/peter/syscalls01/usr/include/linux/pst_syscalls.h

/root/pst/monitor.c

```
#include <linux/pst_syscalls.h>
#include <stdio.h>

int main ( int argc, char *argv[] )
{
      int pid = atoi(argv[1]);
      printf ( "Monitor: %d\n" , pst_monitor ( pid ) );
}
```

/root/pst/notify.c

```
#include <linux/pst_syscalls.h>
#include <stdio.h>

int main ( int argc, char *argv[] )
{
        int pid = atoi(argv[1]);

        printf ( "Notify: %d\n" , pst_notify ( pid ) );
        for ( ; ; )
        {
                //nada
        }
}
```

Recompilando o kernel após as modificações:
Não é necessário refazer todos os passos que fizemos quando compilamos o kernel pela primeira vez. Basta executar os comandos:

```
cd /opt/linux
make bzImage
make install
```

Caso não encontre nenhuma mensagem de erro, reinicie seu ambiente de teste. Durante a compilação, três mensagens de warning serão exibidas fazendo referencia ao arquivo pst_syscalls.h. Elas podem ser ignoradas.

Testando as chamadas de sistema e a kernel_thread:
Após reiniciar, execute os comandos:

```
cd /root/pst
gcc monitor.c -o monior
gcc notify.c -o notify
```

Agora que os executáveis que fazem as chamadas de sistema pst_monitor (pid) e pst_notify (pid) estão prontos, vamos testar.
Abra um segunto terminal e execute:

```
top
```

Volte ao terminal anterior e execute:

```
ps x |grep top
```

Aqui o pid do top é 1990. Substitua 1990 pelo pid do top que foi exibido pelo comando ps:

```
./monitor 1990
./notify 1990
```

Note que o notify não retornou ao bash. Se observar o código fonte notify.c, você vai encontrar um loop infinito. Agora volte ao terminal em que o top esta rodando e digite "q" para sair. Veja o que aconteceu com o notify. Ele foi morto pela kernel_thread.

O que esta acontecendo?
Antes de tentar fazer a sua modificação no kernel, recomendo que você faça alguns testes. Observe que existe uma função chamada printLista (), que esta comentada. Ative esta função e inclua chamadas para ela em pontos estratégicos do código(insira printLista() no início e no final de sys_pst_notify() por exemplo).

Existem algumas funções estranhas no código fonte. Elas podem ser ignoradas por enquanto. Não se preocupe com os write_lock(), write_unlock(), read_lock() e read_unlock(). Se você remover estas funções do código é pouco provável que ocorra algum erro. Mas porque elas estão aqui? Estas funções controlam o acesso a partes do código para impedir, por exemplo, que uma variável seja modificada ao mesmo tempo por duas partes diferentes do código. Para ser sincero, eu não estou 100% seguro sobre o uso delas neste exemplo.

Outras funções estranhas são referentes a kernel_thread. Por exemplo, interruptible_sleep_on_timeout(&wq,HZ), é uma função que faz com que a kernel_thread "durma" por 1 segundo. HZ é uma constante que equivale ao número de cilcos de clock do processador no período de um segundo. Se o seu processador é de 100MHz então a constante HZ = 100.000.000. Se você substituir HZ por 5*HZ, a kernel_thread vai adormecer por cinco segundos ao invés de 1.

Para entender a ordem de execução das chamadas de sistema e da kernel_thread, insira quantos printk ("TESTE"); achar necessário. Insira um printk (" 1 "); após "while (thread_running == 1) {" e veja o que acontece.

Lembre-se que as alterações feitas no kernel só terão efeito, após executar, com sucesso, os comandos:

```
cd /opt/linux
make bzImage
make install
reboot
```

PARTE 3: Novo problema

Proposta de um novo problema:
Implemente uma chamada de sistema e uma kernel_thread:

int monitora_memoria (int usomem): chamada de sistema que recebe um int como parâmetro. Esta syscall recebe a quantidade máxima de memória, em MB, que um processo pode usar. Ela inicia uma kernel_thread que monitora os processos em execução e mata todos aqueles que estão usando mais memória do que é permitido.

PARTE 4: ANEXOS

1 . /opt/linux/arch/i386/kernel/pst_syscalls.c

```
// Peter Senna Tschudin - peter@parahard.com - http://www.parahard.com/peter
#include <linux/pst_syscalls.h>


// Estrutura principal
typedef struct sc_pst_struct
```

```
{
        struct sc_pst_struct    *next;
        int                     monitor;
        int                     notify;
} c_pst_struct;
typedef c_pst_struct* pst_struct;


// Variaveis Globais
int proc_pid , ret , thread_running, ktr;
pst_struct lista;
wait_queue_head_t wq;

// locks
rwlock_t thread_running_lock = RW_LOCK_UNLOCKED;
rwlock_t lista_lock = RW_LOCK_UNLOCKED;

//Prototipos
int insereOrdem ( pst_struct esse ,  int monitor , int notify);
int removeElemento ( pst_struct esse , int monitor );
//void printLista ( );
int pst_daemon2 ( void *unused );
void __init pst_init ( void );


int insereOrdem ( pst_struct esse , int monitor , int notify)
{
      // estruturas auxiliares
      pst_struct tmp, ant, pos;

      // alocacao de memoria
      tmp = ( pst_struct ) kmalloc( sizeof ( c_pst_struct ) , GFP_KERNEL );

      if ( tmp == NULL )
      {
            // erro!
            printk ( "Falta de memoria" );
            return -1;
      }

      // adicionando valores
      tmp->notify = notify;
      tmp->monitor = monitor;

      // lock de leitura
      read_lock ( &lista_lock );

      // cabeca da lista
      ant = esse;

      // primeiro elemento efetivo da lista
      pos = esse->next;

      while ( pos->monitor < monitor )
      {
            ant = pos;
            pos = pos->next;
      }

      // fim do lock de leitura
      read_unlock ( &lista_lock );

      // lock de escrita
      write_lock ( &lista_lock );
```

```
	// notify eh diferente de -1 se e somente se insereOrdem for chamado pela
syscall pst_notify
	if ( notify != -1 )
	{
		if ( pos->monitor == monitor )
		{
			// Caso notify != -1, um novo elemento deve ser adicionado a
estrutura
			if ( pos->notify == -1 )
			{
				pos->notify = notify;
				kfree ( tmp );
				write_unlock ( &lista_lock );
				return 1;
			} else {
				ant->next = tmp;
				tmp->next = pos;
				write_unlock ( &lista_lock );
				return 1;
			}
		} else {
			printk ( "Antes de chamar notify, chame monitor" );
			write_unlock ( &lista_lock );
			return -2;
		}
	} else {
		//nova celula, from monitor where notify = -1
		ant->next = tmp;
		tmp->next = pos;
		write_unlock ( &lista_lock );
		return 1;
	}
	write_unlock ( &lista_lock );
	return 0;
}

/*
void printLista ( )
{
	// lock de leitura
	read_lock ( &lista_lock );

	pst_struct tmp = lista;

	printk ( "\n" );

	while ( tmp->monitor < MAXINT )
	{
		printk ( " ( m:%d,n:%d ) ->" , tmp->monitor , tmp->notify );
		tmp = tmp->next;
	}

	// fim do lock de leitura
	read_unlock ( &lista_lock );

	printk ( " ( m:MAXINT,n:MAXINT ) -> -| \n" );

}
*/

int removeElemento ( pst_struct esse , int monitor )
{
	  // estruturas auxiliares
```

```
        pst_struct ant, pos;

      // lock de leitura
      read_lock ( &lista_lock );

        // cabeca da lista
        ant = esse;

        // primeiro elemento efetivo da lista
        pos = esse->next;

      // fim do lock de leitura
      read_unlock ( &lista_lock );

      // lock de escrita
      write_lock ( &lista_lock );

      //percorre a lista
        while ( pos->monitor < MAXINT )
        {
            // pos e o elemento a ser removido?
            if ( pos->monitor == monitor )
            {
                  //sim, removendo
                  ant->next = pos->next;
                  kfree ( pos );
                  write_unlock ( &lista_lock );
                  return 1;
            }
                ant = pos;
                pos = pos->next;
        }

      //nao, erro!
      write_unlock ( &lista_lock );
      return -1;
}


asmlinkage int sys_pst_notify ( int proc_pid )
{
      // pid nao pode ser -1
      if ( current->pid == -1 )
      {
            printk ( "Erro muito, muito grave! O PID nao pode ser -1. Como voce
fez isso?" );
            return -1;
      }
      ret = insereOrdem ( lista , proc_pid , current->pid );
      if ( ret > 0 )
      {
            // lock de escrita
            write_lock ( &thread_running_lock );

              if ( thread_running == 0 )
            {
                        thread_running = 1;
                  ktr = kernel_thread ( pst_daemon2 , NULL , CLONE_KERNEL );
                    if ( ktr < 0 )
                      {
                          thread_running = 0;
                            printk ( "Erro ao criar kernel_thread" );
                      } else {
                              //printk ( "kernel_thread criada" );
```

```
                }
            }

            // fim do lock de escrita
            write_unlock ( &thread_running_lock );
      }
      return ret;
}


asmlinkage int sys_pst_monitor ( int proc_pid )
{
        //verifica se o procespst com o pid proc_pid existe
        if ( find_task_by_pid ( proc_pid ) == NULL )
        {
                printk ( "O processo %d nao existe" , proc_pid );
                return -1;
        }

      ret = insereOrdem ( lista , proc_pid , -1 );
      return ret;
}


int pst_daemon2 ( void *unused )
{

        // estruturas auxiliares
        pst_struct pos;


      // necessario para a kernel_thread
      init_waitqueue_head(&wq);
      daemonize("pst_daemon2",0);


      // lock de leitura
      read_lock ( &lista_lock );


        // primeiro elemento efetivo da lista
        pos = lista->next;


      // lock de leitura
      read_lock ( &thread_running_lock );


      // loop principal da kernel_thread
      while ( thread_running == 1 )
      {
            // fim do lock de leitura
            read_unlock ( &thread_running_lock );


            // percorre a lista
            while ( ( pos ) && ( pos->monitor < MAXINT ) )
            {
                  // o processo cujo pid eh pos->monitor esta rodando?
                  if ( ( pos->notify > 0 ) && ( find_task_by_pid ( pos-
>monitor ) == NULL ) )
                  {
                        sys_kill ( pos->notify , SIGKILL );
```

```
                        //remove elemento da lista

                        // fim do lock de leitura
                        read_unlock ( &lista_lock );

                        if ( ( removeElemento ( lista ,  pos->monitor ) ) < 0 )
                        {

                              write_lock ( &thread_running_lock );
                              thread_running = 0;
                              write_unlock ( &thread_running_lock );

                              return -1;
                        } else {
                              //lista esta vazia?
                              if ( lista->next->next == NULL )
                              {

                                    write_lock ( &thread_running_lock );
                                    thread_running = 0;
                                    write_unlock ( &thread_running_lock );

                                    return 1;
                              }
                              pos = lista;
                        }
                  }

                  // caminha na lista
                        pos = pos->next;
            }

            // dorme por um segundo
            interruptible_sleep_on_timeout(&wq,HZ);

            //prepara as estruturas auxiliares para percorrer a lista novamente
            pos = lista;

            // lock de leitura
            read_lock ( &thread_running_lock );
      }

      // fim do lock de leitura
      read_unlock ( &thread_running_lock );


      write_lock ( &thread_running_lock );
      thread_running = 0;
      write_unlock ( &thread_running_lock );

      return 0;
}


void __init pst_init ( void )
{
      // variavel das syscalls
      proc_pid = 0;

      // variavel do pst_daemon2
      thread_running = 0;

      //LISTA ENCADEADA ORDENADA COM CABECA E SENTINELA
```

```
      //cabeca da lista
      lista = ( pst_struct ) kmalloc( sizeof ( c_pst_struct ) , GFP_KERNEL );
      lista->monitor = -1;
      lista->notify = -1;

      //sentinela da lista
      lista->next = ( pst_struct ) kmalloc( sizeof ( c_pst_struct ) , GFP_KERNEL
);
      lista->next->monitor = MAXINT;
      lista->next->notify = MAXINT;
      lista->next->next = NULL;
}
```

2. /opt/linux/include/linux/pst_syscalls.h

```
// Peter Senna Tschudin - peter@parahard.com - http://www.parahard.com/peter
#ifndef __LINUX_PST_SYSCALLS_H
#define __LINUX_PST_SYSCALLS_H

#include <linux/syscalls.h>
#include <linux/linkage.h>
#include <linux/unistd.h>
#include <linux/errno.h>
#include <linux/sched.h>
#include <linux/kernel.h>
#include <asm/current.h>
#include <linux/slab.h>
#include <linux/signal.h>
#include <linux/stat.h>

#define MAXINT 0x7FFFFFFF

long (*pst_monitor)(int proc_pid) = NULL;
EXPORT_SYMBOL(pst_monitor);

long (*pst_notify)(int proc_pid) = NULL;
EXPORT_SYMBOL(pst_notify);

#endif
```

3. /usr/include/linux/pst_syscalls.h

```
// Peter Senna Tschudin - peter@parahard.com - http://www.parahard.com/peter
#include <linux/unistd.h>
#include <errno.h>

#define __NR_pst_monitor        311
#define __NR_pst_notify         312

_syscall1(long, pst_monitor, int, proc_pid);
_syscall1(long, pst_notify, int, proc_pid);
```

PARTE 5: Referências:

Proposta do problema resolvido:
http://fubica.lsd.ufcg.edu.br/hp/cursos/oiLinux/index.htm

Preparação do kernel para desenvolvimento:
http://www.fedora.redhat.com/docs/release-notes/fc5/release-notes-ISO/#id3126937

Obras Consultadas:
Understanding the Linux Kernel 3rd edition
Linux Device Drivers

Funções de lista encadeada:
http://www.parahard.com/peter/mini-thesaurus.c