# Laboratório 1.9: Kernel do Linux e Módulos - Parte II

## Introdução

Neste laboratório, continuaremos nosso estudo sobre o kernel do Linux. Mais especificamente, iremos aprender a criar um novo módulo e expandir as funcionalidades do kernel.

## 1. Módulos Hello World

Um programa geralmente começa com uma função main(), executa um monte de instruções e termina após a conclusão dessas instruções. Módulos do kernel trabalham um pouco diferente. Um módulo sempre começa com uma função de entrada para módulos que informa ao kernel qual funcionalidade o módulo fornece e configura o kernel para executar as funções do módulo quando elas forem necessárias. Uma vez que isso acontece, o módulo não faz nada até que o kernel queira fazer algo com o código que o módulo fornece. Todos os módulos terminam chamando a função de saída que desfaz as tarefas executadas na função de entrada.

Vamos começar com um modulo hello world que demonstra os diferentes aspectos dos fundamentos da escrita de um kernel módulo. Aqui está o módulo mais simples possível. Primeiramente faça um diretório de teste:

```
$ mkdir ~/devtitans-kernel
$ cd ~/devtitans-kernel
```

Como mencionado anteriormente, os módulos do kernel devem ter pelo menos duas funções: um "start" (inicialização) que aqui utilizaremos o `init_module`, que é chamada quando o módulo é inserido no kernel, e uma função "end" (limpeza), que é chamada logo antes de ser removido do kernel. Veremos mais a frente que podemos usar o nome que quisermos para essas funções, e aprenderemos a fazer isso mais tarde.

Normalmente, a função `init_module` substitui uma das funções do kernel por seu próprio código. A função `cleanup_module` deve desfazer o que `init_module` fez, e então o módulo pode ser descarregado com segurança.

Para entender melhor, cole o código abaixo em um editor de texto e salve-o como `newlkm-iniciais.c` (troque "**iniciais**" pelas iniciais do seu nome) dentro do diretório criado anteriormente:

```c
/*
 * O módulo do kernel mais simples.
 */
#include <linux/module.h>
#include <linux/kernel.h>

MODULE_LICENSE("GPL");                                  // Tipo de
licença -- afeta o comportamento do tempo de execução
MODULE_AUTHOR("Seu nome");                              // Autor -
- visivel quando usado o modinfo
MODULE_DESCRIPTION("Um simples driver para o devtitans");  //
Descrição do módulo -- visível no modinfo
MODULE_VERSION("0.1");                                  // Versão
do módulo

int init_module(void) {
    printk(KERN_INFO "Bem-vindo à criação do módulo ...\n");
    return 0;
}

void cleanup_module(void) {
    printk(KERN_INFO "O novo módulo foi removido ...\n");
}
```

Se você observar, verá que usamos `printk` ao invés do `printf`. Isso ocorre porque não é uma programação C normal, é uma programação em nível de kernel que é um pouco diferente da programação normal em nível de usuário. Os cabeçalhos `module.h` e `kernel.h` devem ser incluídos para que o código seja compilado.

Uma declaração `MODULE_LICENSE("GPL")` fornece informações (via modinfo) sobre os termos de licenciamento do módulo que você desenvolveu, permitindo assim que os usuários do seu LKM garantem que estão usando o software livre. Como o kernel é lançado sob a GPL, sua escolha de licença afeta a maneira como o kernel trata seu módulo.

Agora você vai precisar de um `Makefile`. Nas versões modernas do kernel, o `Makefile` faz a maior parte da construção para um desenvolvedor. Ele inicia o sistema de construção do kernel e fornece ao kernel informações sobre os componentes necessários para construir o módulo.

Copie e cole o código abaixo e salve o arquivo como `Makefile` na mesma pasta. Se você copiar e colar isso, certifique-se que os recuos no início das linhas com "make" usam **tabulações**, não espaços, senão ocorrerão alguns erros.

```
obj-m += newlkm-iniciais.o
PWD := $(CURDIR)

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Finalmente salve o arquivo `Makefile` e compile usando o comando `make` no terminal aberto no diretório atual:

```
$ make
```

```
  LD [M]  /home/masp/devtitans-kernel2/newlkm-masp.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-109-
```

**Possível Erro:** Se você estiver usando seu próprio laptop/PC e o comando anterior deu erro de compilação, o motivo é possivelmente a falta de algum pacote ou biblioteca. Neste caso, execute o comando `sudo apt-get install build-essential linux-headers-$(uname -r)` para instalar os pacotes necessários.

Se tudo ocorrer bem, você deve descobrir que tem um arquivo `newlkm-iniciais.ko` no diretório atual.

```
$ ls
```

```
Makefile         newlkm-zzzz.c    newlkm-zzzz.mod.c
modules.order    newlkm-zzzz.ko   newlkm-zzzz.mod.o
```

Este arquivo `.ko` é o módulo que será carregado no kernel. Você pode encontrar informações sobre ele com o comando:

```
$ modinfo newlkm-iniciais.ko
```

```
retpoline:      Y
name:           newlkm_zzzz
vermagic:       5.4.0-109-generic SMP mod_unload modversions
```

Neste momento o comando `lsmod | grep newlkm_iniciais` deve não retornar nada pois o módulo ainda não foi habilitado:

```
$ lsmod | grep newlkm_iniciais
```

OBS: note que o "-" (traço) virou "_" (underscore) no nome do módulo. Está correto mesmo, pois o nome é modificado pelo kernel.

Após uma compilação e criação bem-sucedida do módulo, agora é a hora de inseri-lo no kernel para que ele seja carregado em tempo de execução. A inserção do módulo pode ser feita usando os dois utilitários: `modprobe` e `insmod`. A diferença entre os dois está no fato de que `modprobe` cuida do fato de que, se o módulo for dependente de algum outro módulo, esse módulo será carregado primeiro e, em seguida, o módulo principal será carregado. Enquanto o utilitário `insmod` apenas insere o módulo (cujo nome é especificado) no kernel. Portanto, `modprobe` é um utilitário melhor, mas como nosso módulo não depende de nenhum outro módulo, usaremos apenas `insmod`:

```
$ sudo insmod newlkm-iniciais.ko
```

Se este comando não der nenhum erro, isso significa que o LKM foi carregado com sucesso no kernel. Quando você tentar o comando lsmod novamente, agora você deve ver seu módulo carregado:

```
$ lsmod | grep newlkm_iniciais
```

```
newlkm_zzzz            16384  0
```

Veja a mensagem impressa no log do kernel utilizando o comando dmseg:

```
$ dmesg
```

*Copie do terminal e cole abaixo a última linha da saída do comando acima (que contém o Bem-vindo):*

```
[1856941.128440] Bem-vindo à criação do módulo ...
```

Agora que o módulo está inserido no kernel, podemos removê-lo a qualquer momento utilizando o comando:

```
$ sudo rmmod newlkm-iniciais
```

Novamente, se este comando não apresentar nenhum erro, isso significa que o LKM foi descarregado com sucesso no kernel. Para ver o que aconteceu nos logs, utilize novamente o comando dmseg ou o comando abaixo:

```
$ dmesg
```

*Copie do terminal e cole abaixo a última linha da saída do comando acima (que contém o módulo removido):*

```
[1856987.068276] O novo módulo foi removido ...
```

Agora você conhece o básico sobre como criar, compilar, instalar e remover módulos. Este era apenas um LKM fictício. Desta forma, muitos LKM funcionais (que realizam tarefas significativas) funcionam dentro do kernel Linux.

## 2. Modificando as Funções de Início e Fim e Inserindo Parâmetros

Nas primeiras versões do kernel você tinha que usar as funções init_module e cleanup_module como no primeiro exemplo acima, mas hoje em dia você pode nomear tudo o que você quiser usando as macros module_init e module_exit. Essas macros são definidas em include/linux/init.h. A única exigência é que suas funções de inicialização e limpeza devem ser definidas antes de chamar essas macros, caso contrário, você receberá erros de compilação. Aqui está um exemplo desta técnica. Crie um novo arquivo chamado `newlkm-iniciais2.c` e copie o código abaixo utilizando o editor de texto:

```
/*
 * newlkm-iniciais2.c - O módulo do kernel mais simples.
 */
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

MODULE_LICENSE("GPL");                                  // Tipo de
licença -- afeta o comportamento do tempo de execução
MODULE_AUTHOR("Insira Seu nome");                       // Autor -
- visível quando usado o modinfo
MODULE_DESCRIPTION("Um simples driver para o devtitans");  //
Descrição do módulo -- visível no modinfo
MODULE_VERSION("0.1");                                  // Versão
do módulo

static char *name = "mundo";
module_param(name, charp, S_IRUGO);
MODULE_PARM_DESC(name, "O nome para ser mostrado em
/var/log/kern.log");  // Descrição do parâmetro

static int __init hello_2_init(void) {
    printk(KERN_INFO "LKM: Olá %s\n", name);
    return 0;
}
```

```
static void __exit hello_2_exit(void) {
    printk(KERN_INFO "LKM: Finalizando - %s\n", name);
}

module_init(hello_2_init);
module_exit(hello_2_exit);
```

Além disso, note que no código acima, usamos a macro `module_param`. Para permitir que os argumentos sejam passados para o seu módulo, declare as variáveis que tomarão os valores dos argumentos da linha de comando e então use a macro `module_param`, definida em `include/linux/moduleparam.h`. Em tempo de execução, o comando `insmod` preencherá as variáveis com qualquer argumento de linha de comando fornecido, como: `insmod newlkm-iniciais2.ko name=5`.

A macro `module_param` recebe 3 argumentos: o nome da variável, seu tipo e as permissões para o arquivo correspondente no `/sysfs`. Os tipos inteiros podem ser assinados como de costume ou não assinados.

Por último, existe uma macro `MODULE_PARM_DESC`, que é usada para documentar os argumentos que o módulo pode receber. São necessários dois parâmetros: um nome de variável e uma string de forma livre descrevendo essa variável.

Agora abra o arquivo `Makefile` e modifique a primeira linha pra renomear o nome do arquivo a ser compilado e salve o `Makefile`:

```
$ nano Makefile

// Modifique:
obj-m += newlkm-iniciais2.o
```

Finalmente, basta compilar novamente usando o comando make no terminal aberto no diretório atual:

```
$ make
```

```
  LD [M]  /home/masp/devtitans-kernel/newlkm-masp2.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.4.0-109-
```

Liste os arquivos .ko para verificar quais arquivos de módulo estão disponíveis:

```
$ ls -l *.ko
```

```
-rw-rw-r-- 1 masp masp 5616 May 12 01:33 newlkm-masp2.ko
-rw-rw-r-- 1 masp masp 4504 May 12 01:26 newlkm-zzzz.ko
```

Insira o novo módulo modificado no Kernel:

```
$ sudo insmod newlkm-iniciais2.ko
```

Verifique se o módulo foi inserido com sucesso:

```
$ lsmod | grep newlkm_iniciais2
```

```
newlkm_masp2           16384  0
```

Verique novamente as informações do novo módulo:

```
$ modinfo newlkm-iniciais2.ko
```

```
vermagic:       5.4.0-109-generic SMP mod_unload modversions
parm:           name:O nome para ser mostrado em
/var/log/kern.log (charp)
```

Observe que diferente da primeira versão, agora o modinfo exibe novas informações como version, name e param.

Agora procure no log através do dmesg a mensagem informada na inserção do kernel:

```
$ dmesg
```

*Copie do terminal e cole abaixo a última linha da saída do comando acima (que contém o "olá mundo"):*

```
[1857246.558832] LKM: Olá mundo
```

Como não passamos o parâmetro informando o valor da variável name o kernel utilizou o valor padrão setado no código através da linha static char *name = "mundo";<

## 3. Testando o Parâmetro Personalizado LKM

O código anterior contém um parâmetro personalizado, que permite que um argumento seja passado para o módulo do kernel na inicialização. Como não passamos nenhum argumento no teste anterior, o valor padrão usado foi "mundo". Para modificar esse valor passando por parâmetro na inicialização do módulo, primeiramente vamos remover o módulo atual:

```
$ sudo rmmod newlkm-iniciais2
```

Novamente, se nenhum erro for mostrado o módulo foi removido com sucesso. Agora vamos inseri-lo novamente mas agora passando o nome que queremos por parâmetro:

```
$ sudo insmod newlkm-iniciais2.ko name=Devtitans
```

Ao invés de usar o comando `lsmod`, você também pode descobrir informações sobre o módulo do kernel que está carregado através do arquivo `/proc/modules`. Esta é a mesma informação fornecida pelo comando `lsmod`, mas também fornece o deslocamento de memória do kernel atual para o módulo carregado, que é útil para depuração:

```
$ cat /proc/modules | grep newlkm_iniciais2
```

```
newlkm_masp2 16384 0 - Live 0x0000000000000000 (OE)
```

O LKM também possui uma entrada em /sys/module, que fornece acesso direto ao estado do parâmetro personalizado. O sysfs permite que você interaja com o kernel em execução a partir do espaço do usuário lendo ou definindo variáveis dentro dos módulos. Isso pode ser útil para fins de depuração, ou apenas como uma interface para aplicativos ou scripts:

```
$ cd /sys/module/newlkm_iniciais2
$ ls -l
```

```
--w------- 1 root root 4096 May 12 01:35 uevent
-r--r--r-- 1 root root 4096 May 12 01:36 version
```

O parâmetro personalizado pode ser visualizado da seguinte forma: Você pode ver que o estado da variável é exibido e que as permissões de superusuário não são necessárias para ler o valor. Este último é devido ao argumento que foi utilizado na definição do parâmetro do módulo. É possível configurar esse valor para acesso de gravação, mas o código do módulo precisará detectar essa mudança de estado e agir de acordo. Para verificar as informações do parâmetro, faça:

```
$ cd parameters/
$ ls -l
$ cat name
```

```
-r--r--r-- 1 root root 4096 May 12 15:31 name
Devtitans
```

Finalmente, você pode remover o módulo e observar a saída:

```
$ sudo rmmod newlkm-iniciais2.ko
```

Como esperado, isso resultará na mensagem de saída nos logs do kernel:

```
$ dmesg
```

*Copie do terminal e cole abaixo a última linha da saída do comando acima (que contém o Finalizando):*

```
[1857536.988901] LKM: Finalizando - Devtitans
```

*The only true wisdom consists in knowing that you know nothing.* — Socrates

π