



Universidade Federal de Lavras
PPGCC
PCC508 – Sistemas Operacionais

Tópico 10 Lista Avaliativa

Douglas Aquino T. Mendes
28 de janeiro de 2025

Sumário

1	Introdução	2
2	Questões	2
2.1	1	2
3	Desenvolvimento de Códigos	2
3.1	2	2
3.1.1	Código	2
3.1.2	Testes e Resultados	4
3.2	3	4
3.2.1	Código	4
3.2.2	Testes e Resultados	7

1 Introdução

Este documento tem como objetivo apresentar o desenvolvimento das atividades avaliativas para o tópico 10 da disciplina de Sistemas Operacionais, focando na implementação de códigos em linguagem C. Serão apresentadas as questões, a resolução, os códigos desenvolvidos, seguido da apresentação dos resultados da execução do código.

2 Questões

2.1 1

Pergunta: 1) O Kernel pode ser considerado um extenso executável que recebe solicitações dos programas em execução e é responsável por lidar com elas, além de fazer o gerenciamento de recursos. Explique em quais partes podem ser divididas as funções do Kernel.

Resposta: O kernel atua como uma ponte entre o hardware e o software, oferecendo uma camada de abstração para os programas de usuário. Ele é responsável por duas funções principais: fornecer aos programadores de aplicativos um conjunto limpo de recursos abstratos e gerenciar os recursos de hardware [1, p. 2]. As funções do kernel podem ser divididas em várias partes:

- Gerenciamento de processos: O kernel cria, gerencia e finaliza processos [1, p. 506].
- Gerenciamento de memória: O kernel gerencia a memória principal do computador, incluindo a alocação, proteção e mapeamento de endereços de memória para os processos [1, p. 233].
- Gerenciamento de entrada e saída (E/S): O kernel interage com os dispositivos de E/S, emitindo comandos para eles, interceptando interrupções e lidando com erros.
- Comunicação entre processos (IPC): O kernel fornece mecanismos para que os processos possam se comunicar e sincronizar uns com os outros [1, p. 604].
- Chamadas de sistema: O kernel fornece uma interface para que os programas de usuário possam solicitar serviços do sistema operacional, através de chamadas de sistema [1, p. 16].

3 Desenvolvimento de Códigos

3.1 2

Enunciado: 2) O Linux guarda a lista de todos os processos do sistema em uma lista circular. Cada elemento da lista é do tipo `struct task_struct` que contém informações sobre o processo. O primeiro módulo de kernel que deve ser criado desta lista deve percorrer a lista de processos do kernel imprimindo informações sobre cada processo.

3.1.1 Código

```
1 #include <linux/module.h>
2 #include <linux/kernel.h>
3 #include <linux/init.h>
4 #include <linux/sched.h>
5 #include <linux/sched/signal.h>
6 #include <linux/fs.h>
7 #include <linux/types.h>
8 #include <linux/kdev_t.h>
9 #include <linux/cdev.h>
```

```

10
11 MODULE_LICENSE("GPL");
12
13 dev_t dev;
14 struct cdev *meu_cdev;
15
16 static int __init listar_processos_init(void)
17 {
18     struct task_struct *task;
19
20     printk(KERN_INFO "Iniciando o module para listar processos...\n");
21
22     for_each_process(task) {
23         printk(KERN_INFO "PID: %d | Nome: %s | Estado: %ld\n",
24             task->pid, task->comm, task->state);
25     }
26
27     return 0;
28 }
29
30 int meu_open(struct inode *inode, struct file *filp){
31     int minor;
32     minor = iminor(inode);
33     printk(KERN_ALERT "Minor> %d\n", minor);
34     return 0;
35 }
36
37 struct file_operations m_fops =
38 {
39     .owner = THIS_MODULE,
40     .open = meu_open,
41 };
42
43 static int meu_init(void){
44     int major, minor;
45     printk(KERN_ALERT "Iniciando . . .\n");
46     alloc_chrdev_region(&dev,0,1,"drive_SO");
47     major = MAJOR(dev);
48     minor = MINOR(dev);
49     printk(KERN_ALERT "Major: %d Minor: %d\n",major,minor);
50
51     meu_cdev = cdev_alloc();
52     meu_cdev->ops = &m_fops;
53     cdev_add(meu_cdev,dev,1);
54
55     return 0;
56 }
57
58 static void meu_exit(void){
59
60     unregister_chrdev_region(dev,1);
61     cdev_del(meu_cdev);
62     printk(KERN_ALERT "Finalizando . . .\n");
63 }

```

```

64
65 module_init(listar_processos_init);
66 module_exit(meu_exit);

```

3.1.2 Testes e Resultados

Como resultado da execução do código exibido na subseção 3.2.1, obtivemos a saída ilustrada na figura 1.

```

53412.388609] PID: 1 | Nome: systemd | Estado: 0
53412.388611] PID: 2 | Nome: kthreadd | Estado: 0
53412.388613] PID: 3 | Nome: pool_workqueue_ | Estado: 0
53412.388615] PID: 4 | Nome: kworker/R-rcu_g | Estado: 0
53412.388617] PID: 5 | Nome: kworker/R-rcu_p | Estado: 0
53412.388618] PID: 6 | Nome: kworker/R-slub_ | Estado: 0
53412.388620] PID: 7 | Nome: kworker/R-netns | Estado: 0
53412.388622] PID: 10 | Nome: kworker/0:0H | Estado: 0
53412.388624] PID: 12 | Nome: kworker/R-mm_pe | Estado: 0
53412.388625] PID: 13 | Nome: rcu_tasks_kthre | Estado: 0
53412.388627] PID: 14 | Nome: rcu_tasks_rude_ | Estado: 0
53412.388629] PID: 15 | Nome: rcu_tasks_trace | Estado: 0
53412.388630] PID: 16 | Nome: ksoftirqd/0 | Estado: 0
53412.388632] PID: 17 | Nome: rcu_preempt | Estado: 0
53412.388633] PID: 18 | Nome: migration/0 | Estado: 0
53412.388635] PID: 19 | Nome: idle_inject/0 | Estado: 0
53412.388637] PID: 20 | Nome: cpuhp/0 | Estado: 0
53412.388639] PID: 21 | Nome: cpuhp/1 | Estado: 0
53412.388640] PID: 22 | Nome: idle_inject/1 | Estado: 0
53412.388642] PID: 23 | Nome: migration/1 | Estado: 0
53412.388644] PID: 24 | Nome: ksoftirqd/1 | Estado: 0
53412.388645] PID: 26 | Nome: kworker/1:0H | Estado: 0
53412.388647] PID: 27 | Nome: cpuhp/2 | Estado: 0
53412.388649] PID: 28 | Nome: idle_inject/2 | Estado: 0
53412.388651] PID: 29 | Nome: migration/2 | Estado: 0

```

Figura 1: Resultado da execução do programa

3.2 3

Enunciado: 3) Esse exercício é continuação de um feito na lista de estudo. Desenvolva um módulo de Kernel do Linux que seja um driver de dispositivo do tipo caractere. Cada vez que um dado é escrito no Driver, ele deve ser guardado em uma lista do kernel criada para isso. No caso de leitura, cada leitura deve retirar um nó dessa lista e retornar os dados. Se a lista estiver vazia, deve retornar a palavra “vazia”.

3.2.1 Código

```

1 #include <linux/module.h>
2 #include <linux/types.h>
3 #include <linux/kdev_t.h>
4 #include <linux/cdev.h>

```

```

5 #include <linux/fs.h>
6 #include <linux/slab.h>
7 #include <linux/uaccess.h>
8 #include <linux/list.h>
9
10 MODULE_LICENSE("GPL");
11
12 dev_t dev;
13 struct cdev *meu_cdev;
14 struct list_head data_list;
15
16 struct data_node {
17     struct list_head list;
18     char *data;
19 };
20
21 int meu_open(struct inode *inode, struct file *filp) {
22     return 0;
23 }
24
25 ssize_t meu_read(struct file *filp, char __user *buf, size_t len, loff_t *off)
26 {
27     struct data_node *node;
28     ssize_t ret;
29
30     if (list_empty(&data_list)) {
31         if (*off == 0) {
32             if (copy_to_user(buf, "vazia", 5)) {
33                 return -EFAULT;
34             }
35             *off += 5;
36             return 5;
37         } else {
38             return 0;
39         }
40     }
41
42     node = list_first_entry(&data_list, struct data_node, list);
43     list_del(&node->list);
44
45     if (copy_to_user(buf, node->data, strlen(node->data))) {
46         ret = -EFAULT;
47     } else {
48         ret = strlen(node->data);
49         *off += ret;
50     }
51
52     kfree(node->data);
53     kfree(node);
54
55     return ret;
56 }

```

```

57 ssize_t meu_write(struct file *filp, const char __user *buf, size_t len,
58     loff_t *off) {
59     struct data_node *node;
60
61     node = kmalloc(sizeof(*node), GFP_KERNEL);
62     if (!node) {
63         return -ENOMEM;
64     }
65
66     node->data = kmalloc(len, GFP_KERNEL);
67     if (!node->data) {
68         kfree(node);
69         return -ENOMEM;
70     }
71
72     if (copy_from_user(node->data, buf, len)) {
73         kfree(node->data);
74         kfree(node);
75         return -EFAULT;
76     }
77
78     list_add_tail(&node->list, &data_list);
79
80     return len;
81 }
82
83 struct file_operations m_fops = {
84     .owner = THIS_MODULE,
85     .open = meu_open,
86     .read = meu_read,
87     .write = meu_write,
88 };
89
90 static int __init meu_init(void) {
91     int major, minor;
92     printk(KERN_ALERT "Iniciando . . .\n");
93     alloc_chrdev_region(&dev, 0, 1, "drive-SO");
94     major = MAJOR(dev);
95     minor = MINOR(dev);
96     printk(KERN_ALERT "Major: %d Minor: %d\n", major, minor);
97
98     meu_cdev = cdev_alloc();
99     meu_cdev->ops = &m_fops;
100     cdev_add(meu_cdev, dev, 1);
101
102     INIT_LIST_HEAD(&data_list);
103
104     return 0;
105 }
106
107 static void __exit meu_exit(void) {
108     struct data_node *node, *tmp;
109
110     list_for_each_entry_safe(node, tmp, &data_list, list) {

```


```

110         list_del(&node->list);
111         kfree(node->data);
112         kfree(node);
113     }
114
115     cdev_del(meu_cdev);
116     unregister_chrdev_region(dev, 1);
117     printk(KERN_ALERT "Saindo . . .\n");
118 }
119
120 module_init(meu_init);
121 module_exit(meu_exit);

```

3.2.2 Testes e Resultados

Como resultado da execução do código exibido na subseção 3.2.1, obtivemos a saída ilustrada na figura 4.

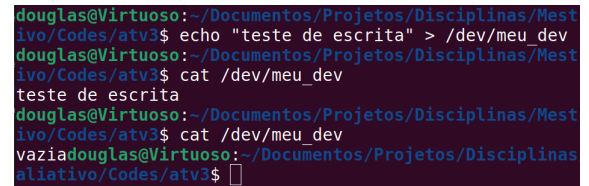


```

070267] Iniciando . . .
070276] Major: 236 Minor: 0
@Virtuoso:~/Documentos/Projetos/Disciplinas/Mest
ivo/Codes/atv3$ sudo mknod /dev/meu_dev c 236 0
@Virtuoso:~/Documentos/Projetos/Disciplinas/Mest
ivo/Codes/atv3$

```

Figura 2: Criando o dispositivo



```

douglass@Virtuoso:~/Documentos/Projetos/Disciplinas/Mest
ivo/Codes/atv3$ echo "teste de escrita" > /dev/meu_dev
douglass@Virtuoso:~/Documentos/Projetos/Disciplinas/Mest
ivo/Codes/atv3$ cat /dev/meu_dev
teste de escrita
douglass@Virtuoso:~/Documentos/Projetos/Disciplinas/Mest
ivo/Codes/atv3$ cat /dev/meu_dev
vaziadouglass@Virtuoso:~/Documentos/Projetos/Disciplinas
aliativo/Codes/atv3$

```

Figura 3: Escrevendo e lendo do dispositivo

Figura 4: Modulo instalado e Teste do dispositivo

Referências

- [1] A. S. Tanenbaum e H. Bos, *SISTEMAS OPERACIONAIS MODERNOS*, 4^a ed. Boston: Pearson, 2021.