カーネルハック

筑波大学 情報学群 情報科学類 3 年
学籍番号 202213581

高名 典雅

2022 年 8 月 4 日

第 1 章

カーネルのコンパイルとパラメタの設定

この章では課題 1 で行った作業について説明する．

1.1 config の生成と編集

config の生成には defconfig をパラメタの設定には menuconfig を用いた．

$ make defconfig $ make menuconfig

defconfig は menuconfig に比べて最小限の設定しか生成しないため，まず defconfig を用いて軽量な設定ファイルを作ってから menuconfig を使って TUI で Network や Virtualization 内にある不要そうなドライバやモジュールを無効化し，更に設定を削った．defconfig よりも小さな設定ファイルを生成する tinyconfig というオプションもあるが，これらはカーネルを起動させるのに追加で設定を足していく必要があるため今回は用いなかった．[1]不要なオプションを削ることにより余計なモジュールの build にかかる時間を削減できるため結果的に高速に build が可能である．

高速化のための設定の見直しだけでなくこの左記で必要な項目の設定も行った．まず TUI 経由でカーネルのデバッグ情報を有効化した．これによりデバッグ情報が付加されてデバッガでコードが追えるようになる．他に FIle systems の Root file system on NFS を無効化し Ext4 のファイルシステムが有効になっているかを確認した．

更に 2 章で gdb を使ってカーネルのデバッグをするためカーネルイメージのアドレスのランダム化のオプションを切っておく．この操作は TUI を使わずにソースのルートディレクトリ直下にある.config を直接編集して CONFIG_RANDOMIZE_BASE を無効化し，CONFIG_DEBUG_INFO を有効化した．

1.2 ビルドの準備と実行

ビルドする前に準備としてターミナル上で以下のコマンドを実行した．

$ ./scripts/config –disable SYSTEM_TRUSTED_KEYS $ ./scripts/config –disable SYSTEM_REVOCATION_KEYS
$ ./scripts/config –disable MODULE_SIG_KEY

これらの設定はビルド中の認証を回避するためのものである．

ビルドに必要な設定がすべて完了したのでビルドを行う．今回ホストは以下に示すように ubuntu なので，

$ uname -a Linux kh-ubuntu 5.15.0-41-generic #44 20.04.1-Ubuntu SMP Fri Jun 24 13:27:29 UTC 2022 x86_64 x86_64 x86_64 GNU/Linux

debian パッケージ経由でインストールできるように bindev-pkg を引数に渡して実行する．また，パッケージ名から内容を区別できるように KDEB_PKGVERSION という変数名を make に渡して命名を行う．これは複数回施行したときにバックアップを取った際の混乱を避けるためである．

[1] 設定は必要なものを足すより不要そうなものを削るほうが簡単だからである．

$ make -j8 bindeb-pkg KDEB_PKGVERSION=5.17.1-DisableRandomizeBase

make に-j8 というオプションをつけることで 8 ジョブで並列に処理ができ更にビルドを高速化することができる．ビルドが終わったら VMware でゲスト OS を立ち上げインストールを行う．

$ cd kernel_hack/ $ dpkg –install *.deb $ uname -a Linux ubuntu 5.17.1 #2 SMP PREEMPT Thu Apr 21 19:27:13 JST 2022 x86_64 x86_64 x86_64 GNU/Linux

インストールしたら再起動して uname でカーネルのバージョンを表示する．期待していたカーネルをインストールして起動できることが確認できた．

第 2 章

カーネルのリモート・デバッグ

この章では課題 2 で行った作業について説明する．課題 2 では gdb のリモート・デバッグ機能を用いて，カーネルのプログラムをデバッグ可能な状態にしステップ実行でシステムコールを追う．

2.1 デバッグの準備

まず，カーネルを VMware 経由でデバッグするための準備を行う．1 章で行ったアドレスのランダム化の無効化やデバッグ情報の付加もデバッグのための準備の 1 つであった．ここでは VMware の設定を中心に説明する．

まず，vim を使って VMware の.vmx ファイルを直接編集する．

$ cd $ cd vmware/Ubuntu 64-bit/ $ vim Ubuntu 64-bit.vmx debugStub.listen.guest64 = ”TRUE”

debugStub.listen.guest64 という項目を TRUE にする．これによりポート 8864 番経由で gdb からリモートデバッグができるようになる．

2.2 デバッグ対象の準備

今回は rmdir というシステムコールの流れを gdb を使って追った．rmdir はディレクトリを削除するときに呼ばれるシステムコールで，ユーザが意図的にシステムコールを呼ぶような操作を行わない限り実行されないだろうという推測のもと選んだ．

デバッグ対象となる，rmdir を呼ぶアセンブリをリスト 2.1 に示す．

Listing 1: rmdir.s

```
.intel_syntax noprefix
.global main

.LC1:
    .string "target_dir"

main:
    push rbp
    mov rbp, rsp

    mov rax, 84 # rmdir
    lea rdi, .LC1[rip]

    syscall

    mov rsp, rbp
    pop rbp
```

```
18      ret
```

リスト 2.1: rmdir を呼ぶアセンブリ

rmdir に割り当てられている番号は 84 番 [1] なので rax に 84 を入れて syscall を呼んでいる．第一引数となる rdi レジスタには消去したいディレクトリ名を入れている．このプログラムを実行するとシステムコール rmdir が呼ばれカレントディレクトリにある”target_dir”というディレクトリが消される．

2.3　gdb を用いたデバッグ

準備ができたのでデバッグを行う．gdb を起動して以下のコマンドを入力し，rmdir のエントリ関数に breakpoint を仕掛ける．

> $ gdb ¿¿¿ file vmlinux ¿¿¿ target remote localhost:8864 ¿¿¿ lx-symbols ¿¿¿ b __x64_sys_rmdir ¿¿¿ c ¿¿¿ b do_rmdir

$がターミナルのプロンプト, ¿¿¿は gdb 内のプロンプトである．まず gdb を起動し, vmlinux を読み込む．vmlinux にはデバッグに必要なデバッグシンボルやシンボルテーブルが内包されていて，カーネルをビルドしたディレクトリに置かれている．

次にリモートの対象を指定してアタッチする．今回の場合は前述の通り localhost の 8864 番である．アタッチできたら lx-symbols でシンボルを読み込む．

ここまでで全ての準備が整ったのでシステムコールの流れを追っていく．まずシステムコールの流れを追うためにはその処理を行っている入口にブレークポイントを張る必要がある．対応表を見ると__x64_sys_rmdir にブレークポイントを張れば良いことが分かる．[1] ブレークポイントを張ったら一度 c(continue) で飛ぶ．ると rmdir の処理を行う do_rmdir が見つかるのでそこにブレークポイントを仕掛ける．

第 3 章

システムコールの追加

本章では課題 3 のシステムコールの追加について説明する．今回は配列の中央値を返すシステムコールを実装した．

3.1　システムコールの実装

まず，今回実装したシステムコールの概要について述べる．今回実装したのは median という与えられた int 型配列の中央値を求めるシステムコールである．配列のポインタと配列のサイズ，結果を格納する変数のポインタを受け取り結果を変数のポインタに格納して終了する．

詳しい仕様を man の形式に倣って以下に示す．

Listing 2

```
\$ man 2 median
NAME
       median - get a median

SYNOPSIS
       \#include <unistd.h>

       int median(int *array, const int size, int *result);

DESCRIPTION
       median() は配列の中央値を取得する．配列は int 型でなくてはならない．
       配列の要素数が奇数の場合，ソートされた配列のちょうど真ん中の値が result にセットされる．
       配列の要素数が偶数の場合，ソートされた配列の真ん中 2 つの値のうち大きなものが取られる．

RETURN VALUE
       成功した場合，0 が返される．
       失敗した場合は下に示す ERRORS の値が返される．

ERRORS
       EINVAL 配列のサイズが 0 以下の場合，引数が不正なので中央値を求めずに終了する．
              result には何もセットされない．
       EFAULT カーネル内で配列のコピーに失敗した場合，処理が行えないため異常終了する．
              配列のサイズが大きすぎる場合などに発生する．
              result には何もセットされない．
```

リスト 3.2: median の仕様

注意点として配列の要素数が偶数の場合，ソートされた配列の真ん中 2 つの値のうち大きなものが取られる．通常は中央に近い 2 つの値の平均を取るが，カーネル空間では浮動小数点が使えないので今回はこのような仕様にした．

上記の仕様に沿って作成したシステムコールをリスト 3.3 に示す．

Listing 3: my_syscall.c

```c
#include <linux/syscalls.h>

#define         EFAULT 14         /* Bad address */
#define         EINVAL 22         /* Invalid argument */
#define BUFFERSIZE 100

void myswap (int *x, int *y) {
    int tmp;

    tmp = *x;
    *x = *y;
    *y = tmp;
}

int partition (int *array, int left, int right) {
    int i = left;
    int j = right + 1;
    int pivot = left;

    do {
        do { i++; } while (array[i] < array[pivot]);
        do { j--; } while (array[pivot] < array[j]);
        if (i < j) { myswap(&array[i], &array[j]); }
    } while (i < j);

    myswap(&array[pivot], &array[j]);

    return j;
}

void quick_sort (int *array, int left, int right) {
    int pivot;

    if (left < right) {
        pivot = partition(array, left, right);
        quick_sort(array, left, pivot-1);
        quick_sort(array, pivot+1, right);
    }
}

int do_median(int __user *_array, const int size, int __user *_result) {
    int array[BUFFERSIZE] = {0};
    int result = 0;

    printk("syscall median start.");
    if (size <= 0) return EINVAL;
    if (size > BUFFERSIZE) return EINVAL;
    if (copy_from_user(array, _array, sizeof(int) * size) > 0) return EFAULT;
    if (copy_from_user(&result, _result, sizeof(int)) > 0) return EFAULT;

    quick_sort(array, 0, size-1);

    result = array[size/2];
```

```
54
55      if (copy_to_user(_result, &result, sizeof(int)) > 0) return EFAULT;
56
57      printk("syscall median done.");
58      return 0;
59  }
60
61  // https://elixir.bootlin.com/linux/v5.17.1/source/fs/namei.c#L4097
62  SYSCALL_DEFINE3(median,
63          int __user *, _array,
64          const int, size,
65          int __user *, _result) {
66
67      return do_median(_array, size, _result);
68  }
```

リスト 3.3: median の実装

まず，SYSCALL_DEFINE3 というマクロを使ってシステムコールの定義を登録する．今回は引数が 3 つあるので SYSCALL_DEFINE3 を用いている．このマクロでは引数と引数の型をカンマで区切る必要があり，またユーザ空間のポインタであることを示すために__user をつけてポインタの型を表現している．

SYSCALL_DEFINE3 の return の部分で本体に当たる do_median を呼び出す．これは 2 章で rmdir の流れを追ったときと同じ流れである．do_median ではまず引数のチェックを行い実行が不可能な場合はエラーを返して終了する．そうでない場合はまずユーザ空間のデータをカーネル空間のデータに copy_from_user を使って安全にコピーする．カーネル空間からユーザ空間のポインタを触るのは危険なので一度カーネル空間のメモリにコピーして操作をしてから書き戻す．これに失敗した場合もエラーを返して終了する．

次に中央値を得るためにソートする．この操作はユーザ空間のプログラムと変わらない普通の関数である．そして size を 2 で割った要素にアクセスして中央値を得る．これを copy_to_user を使って安全にユーザ空間のアドレスに書き戻す．成功した場合には 0 を返して終了する．以上が処理の流れである．

3.2 システムコールの呼び出し

システムコールはカーネルのソースコードを改変する形で組み込むので 1 度カーネルをビルドし直す必要がある．以下の操作でビルドを行った．（カッコ内は修正内容）

> $ cd /kernel_hack/linux-5.17.1/arch/x86/kernel $ cp /kernel_hack/myrepo/src/CH3/my_syscall.c ./median.c $ vim /kernel_hack/linux-5.17.1/arch/x86/kernel/Makefile (add ”obj-y += median.o” to line 59) $ vim ../../../include/linux/syscalls.h (add ”asmlinkage long sys_median(int __user *array, const int size, float __user *result);” to line 1000) $ vim ../../../arch/x86/entry/syscalls/syscall_64.tbl (add ”336 common median sys_median”) $ cd /kernel_hack/linux-5.17.1/ $ make -j8 bindeb-pkg KDEB_PKGVERSION=5.17.1-AddMedian

まず，リスト 3.2 で示したプログラムを/arch/x86/kernel にコピーする．次にリンクできるように対象となる median.o をターゲットに追加する．また syscalls.h を編集して関数定義を登録し，syscall_64.tbl にシステムコールの番号と名前，関数を登録する．準備ができたら 1 章と同様の手順でビルドする．

次にゲスト OS に移動し 1 章と同様の手順でインストールを行う．

> $ cd kernel_hack/ $ sudo dpkg -r linux-image-5.17.1 $ sudo dpkg -r linux-image-5.17.1-dbg $ sudo dpkg –install *AddMedian*.deb $ uname -a Linux ubuntu 5.17.1 #AddMedian SMP PREEMPT Sat May 28 00:35:01 JST 2022 x86_64 x86_64 x86_64 GNU/Linux

インストールして OS が起動することを確認できたら実際にシステムコールを呼び出して動作を検証する．システムコールを呼ぶプログラムをリスト??,??に示す．

Listing 4: call_median.c

```c
#include <stdio.h>
#include "call_median.h"

int main(void) {
    int array[7] = {6, 2, 7, 1, 3, 5, 4};
    int result = 0;

    syscall(__NR_median, array, 7, &result);
    printf("syscall_median(array, 7, &result) -> %d\n", result);
    return 0;
}
```

リスト 3.4: median を呼ぶプログラム

Listing 5: call_median.h

```c
#ifndef __LINUX_NEW_SYSCALL_H
#define __LINUX_NEW_SYSCALL_H
#include "unistd_64.h"
#include <unistd.h>
#include <asm/unistd.h>
#endif
```

リスト 3.5: call_median.h

syscall 関数を使って任意のシステムコールを呼び出すことができるのでそれを利用する．第 1 引数の__NR_median はマクロで実際には long 型の番号が割り当てられている．[2] リスト 3.5 で示した call_median.h ではシステムコールに必要なマクロなどを参照している．実行の流れを以下に示す．

$ cd /tmp/CH3 $ cp /kernel_hack/linux-5.17.1/arch/x86/include/generated/uapi/asm/unistd_64.h . $ cp /kernel_hack/my_repo/src/CH3/call_median* . $ gcc call_median.c -o CallMedian $ ./CallMedian

call_median.h で参照する unistd_64.h 事前にをコピーしている．

3.3 システムコールの動作確認

第 4 章

デバイス・ドライバの作成

4 章ではデバイス・ドライバを作成する．今回は open, write, read, close, ioctl などの関数を用意して読み書きや特別な操作のできるキャラクタデバイスを作成した．

4.1　キャラクタデバイスの作成

今回作成するデバイス・ドライバは読み書きと ioctl による 2 つの値の設定・取得・交換のできるキャラクタデバイスである．

作成したデバイス・ドライバのソースコードとヘッダをリスト 4.6,4.7 に示す．

Listing 6: my_chdev.c

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/types.h>
#include <linux/cdev.h>
#include <linux/slab.h>
#include "my_chdev.h"
#define DRIVER_NAME "MyDevice"
#define BUF_SIZE 256

static const unsigned int MINOR_BASE = 0;
static const unsigned int MINOR_NUM  = 3;
static unsigned int major_num;
static struct cdev my_cdev;

MODULE_LICENSE("Dual BSD/GPL");

struct _mydevice_file_data {
    unsigned char buffer[BUF_SIZE];
    struct parameter param;
};

static long mydevice_ioctl(struct file *fp, unsigned int cmd, unsigned long arg) {
    printk("called ioctl\n");

    int tmp = 0;
    switch (cmd) {
        case MYDEVICE_SET_VALUES:
            if (copy_from_user(&(((struct _mydevice_file_data *)fp->private_data)->param),
 ↪  (struct parameter __user *)arg, sizeof(struct parameter))) {
```

```
30                  return -EFAULT;
31              }
32              break;
33          case MYDEVICE_GET_VALUES:
34              if (copy_to_user((struct parameter __user *)arg, &(((struct _mydevice_file_data
    ↪   *)fp->private_data)->param), sizeof(struct parameter))) {
35                  return -EFAULT;
36              }
37              break;
38          case MYDEVICE_SWAP_VALUES:
39              tmp = ((struct _mydevice_file_data *)fp->private_data)->param.value1;
40              ((struct _mydevice_file_data *)fp->private_data)->param.value1 = ((struct
    ↪   _mydevice_file_data *)fp->private_data)->param.value2;
41              ((struct _mydevice_file_data *)fp->private_data)->param.value2 = tmp;
42              break;
43          default:
44              printk(KERN_WARNING "unsupported command %d\n", cmd);
45              return -EFAULT;
46      }
47
48      return 0;
49  }
50
51  static int mydevice_open(struct inode *inode, struct file *fp) {
52      printk("open my device\n");
53
54      struct _mydevice_file_data *p = kmalloc(sizeof(struct _mydevice_file_data), GFP_KERNEL);
55      if (p == NULL) {
56          printk(KERN_ERR  "kmalloc\n");
57          return -ENOMEM;
58      }
59
60      strlcat(p->buffer, "init", 4);
61      p->param.value1 = 0;
62      p->param.value2 = 0;
63      fp->private_data = p;
64
65      return 0;
66  }
67
68  static int mydevice_close(struct inode *inode, struct file *fp) {
69      printk("close my device\n");
70
71      if (fp->private_data) {
72          kfree(fp->private_data);
73      }
74
75      return 0;
76  }
77
78  static ssize_t mydevice_read(struct file *fp, char __user *_buf, size_t count, loff_t *f_pos)
    ↪   {
79      printk("read my device\n");
80
```

```
81      if (count > BUF_SIZE) count = BUF_SIZE;
82
83      struct _mydevice_file_data *p = fp->private_data;
84      if (copy_to_user(_buf, p->buffer, count) != 0) {
85          return -EFAULT;
86      }
87      return count;
88  }
89
90  static ssize_t mydevice_write(struct file *fp, const char __user *_buf, size_t count, loff_t
    ↪  *f_pos) {
91      printk("write my device\n");
92
93      struct _mydevice_file_data *p = fp->private_data;
94      if (copy_from_user(p->buffer, _buf, count) != 0) {
95          return -EFAULT;
96      }
97      return count;
98  }
99
100 struct file_operations s_mydevice_fops = {
101     .open    = mydevice_open,
102     .release = mydevice_close,
103     .read    = mydevice_read,
104     .write   = mydevice_write,
105     .unlocked_ioctl = mydevice_ioctl,
106 };
107
108 static int mydevice_init(void) {
109     printk("init my device\n");
110
111     dev_t dev;
112     if (alloc_chrdev_region(&dev, MINOR_BASE, MINOR_NUM, DRIVER_NAME) != 0) {
113         printk(KERN_ERR  "alloc_chrdev_region = %d\n", major_num);
114         return -1;
115     }
116
117     major_num = MAJOR(dev);
118     dev = MKDEV(major_num, MINOR_BASE);
119
120     cdev_init(&my_cdev,&s_mydevice_fops);
121     my_cdev.owner = THIS_MODULE;
122     int cdev_err = cdev_add(&my_cdev, dev, MINOR_NUM);
123     if (cdev_err != 0) {
124         printk(KERN_ERR  "cdev_add = %d\n", cdev_err);
125         unregister_chrdev_region(dev, MINOR_NUM);
126         return -1;
127     }
128     return 0;
129 }
130
131 static void mydevice_exit(void) {
132     printk("exit my device\n");
133
```

```
134     dev_t dev = MKDEV(major_num, MINOR_BASE);
135     cdev_del(&my_cdev);
136     unregister_chrdev_region(dev, MINOR_NUM);
137 }
138
139 module_init(mydevice_init);
140 module_exit(mydevice_exit);
```

リスト 4.6: 作成したデバイス・ドライバ

Listing 7: my_chdev.h

```
1  #pragma once
2  #include <linux/ioctl.h>
3
4  struct parameter {
5      int value1;
6      int value2;
7  };
8
9  #define MYDEVICE_IOC_TYPE 'X'
10 #define MYDEVICE_SET_VALUES _IOW(MYDEVICE_IOC_TYPE, 1, struct parameter)
11 #define MYDEVICE_GET_VALUES _IOR(MYDEVICE_IOC_TYPE, 2, struct parameter)
12 #define MYDEVICE_SWAP_VALUES _IOR(MYDEVICE_IOC_TYPE, 3, struct parameter)
```

リスト 4.7: my_chdev.h

まず module_init マクロと module_exit マクロを使って load 時と unload 時に呼ばれる関数を登録する．次に file_operations 構造体を定義して open, release, read, write にそれぞれ呼ばれる関数のポインタを登録する．また，unlocked_ioctl に ioctl の操作を定義する関数のポインタを登録する．このようにすることで各動作時の振る舞いを簡単に定義することができる．

各関数の処理について説明する．まず，mydevice_init ではキャラクタデバイスの登録をしている．alloc_chrdev_region 関数を用いてメジャー番号とマイナー番号，ドライバの名前などを登録し，cdev_init 関数で初期化，cdev_add で登録を行う．mydevice_exit では逆に cdev_del で削除を行い，unregister_chrdev_region で登録も取り消す．

mydevice_open ではファイルポインタの private_data に ioctl のためのデータ領域を確保する．グローバル変数に領域を確保しても良いが，その場合同時に複数キャラクタデバイスにアクセスがあったときにデータが 1 つしか存在できないので競合が起きてしまう．ファイルポインタの領域に kmalloc を使って確保することで複数の接続でもそれぞれで専用のデータ領域を確保できる．mydevice_close では open で確保した領域を確保している．

mydevice_read ではファイルポインタに保存しているデータをユーザにコピーしてる．この際 copy_to_user を用いて安全にユーザ空間のデータをカーネル空間に持ち込んでいる．mydevice_write では read の逆で，ユーザ空間からのデータをファイルポインタに確保している空間にコピーしている．

mydevice_ioctl ではコマンドを受け取って実行する部分の処理を実装している．MYDEVIDE_SET_VALUES では構造体 parameter を受け取ってデータ領域に値をセットする．MYDEVICE_GET_VALUES では保存されていた値を arg に代入して返す．MYDEVICE_SWAP_VALUES は構造体 parameter に 2 つあるデータを交換する．

4.2 キャラクタデバイスの動作確認

上記の実装が正しく動作しているかを検証する．キャラクタデバイスをビルドするために用いた Makefile をリスト 4.8 に示す．

Listing 8: Makefile

```
obj-m := my_chdev.o

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

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) clean
	rm a.out

load:
	sudo insmod my_chdev.ko
	sudo mknod --mode=666 /dev/mydevice0 c `grep MyDevice /proc/devices | awk '{print
	↪ \$\$1;}'` 0
	sudo mknod --mode=666 /dev/mydevice1 c `grep MyDevice /proc/devices | awk '{print
	↪ \$\$1;}'` 1
	sudo mknod --mode=666 /dev/mydevice2 c `grep MyDevice /proc/devices | awk '{print
	↪ \$\$1;}'` 2

unload:
	sudo rmmod my_chdev.ko
	sudo rm /dev/mydevice0
	sudo rm /dev/mydevice1
	sudo rm /dev/mydevice2


.PHONY: load remove
```

リスト 4.8: キャラクタデバイスをビルドするために用いた Makefile

マイナー番号が機能しているか確認するために複数のデバイスを同時に作成している．実際に動作を確認するときは以下のようにビルドして動作を確認する．

$ make $ make load $ gcc call_mydevice.c $ ./a.out $ make unload

作成したキャラクタデバイスを開いて操作するプログラムをリスト 4.9 に示す．

Listing 9: call_mydevice.c

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <sys/ioctl.h>
#include "my_chdev.h"

int main() {
    int fd;
    struct parameter values_set;
    struct parameter values_get;
    values_set.value1 = 1;
    values_set.value2 = 2;

    if ((fd = open("/dev/mydevice0", O_RDWR)) < 0) perror("open");
```

```
17
18      if (ioctl(fd, MYDEVICE_SET_VALUES, &values_set) < 0) perror("ioctl_set");
19      if (ioctl(fd, MYDEVICE_GET_VALUES, &values_get) < 0) perror("ioctl_get");
20      if (ioctl(fd, MYDEVICE_SWAP_VALUES, &values_set) < 0) perror("ioctl_get");
21      if (ioctl(fd, MYDEVICE_GET_VALUES, &values_get) < 0) perror("ioctl_get");
22
23      // expected val1 = 2, val2 = 1
24      printf("value1 = %d, value2 = %d\n", values_get.value1, values_get.value2);
25
26      if (close(fd) != 0) perror("close");
27      return 0;
28  }
```

リスト 4.9: 作成したキャラクタデバイスを開いて操作するプログラム

第 5 章

/proc ファイル・システムの作成

本章では proc ファイルシステムを通じてアクセス可能なモジュールを作成する．今回は neofetch[3] のような複数のデバイスの情報をまとめて閲覧できる proc ファイルを作成した．

5.1 proc ファイルの実装

今回表示する情報として，

- カーネルのバージョン
- uptime
- CPU のベンダ
- CPU のモデル
- CPU の周波数
- キャッシュのサイズ
- CPU の core id
- CPU のコア数
- メモリの総使用量
- 使ってないメモリ
- 使用中のメモリ

を選んだ．

複数のデバイスの情報をまとめて表示するにあたって cpu やメモリの情報をカーネル内の構造体から直接取得した．実装をリスト 5.10 に示す．

Listing 10: my_procfs.c

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/types.h>
#include <linux/fs.h>
#include <linux/mm.h>
#include <linux/proc_fs.h>
#include <linux/slab.h>
#include <linux/cpufreq.h>
#include <linux/utsname.h>
#include <linux/kallsyms.h>
#include <linux/kprobes.h>
#include <linux/time_namespace.h>
#include <asm/page_types.h>
#include <asm/processor.h>
```

```c
#include <asm/cpufeature.h>

#include "my_procfs.h"

#define pagenum_to_KB(x) ((x) << (PAGE_SHIFT - 10))
#define DRIVER_NAME "MyDevice"
#define PROC_NAME "MyProcFs"
#define BUF_SIZE 512

static char buffer[BUF_SIZE];
static unsigned int major_num;

MODULE_LICENSE("Dual BSD/GPL");


static int mydevice_open(struct inode *inode, struct file *fp) {
    printk("open my device\n");
    int *buf_pos = kmalloc(sizeof(int), GFP_KERNEL);
    *buf_pos = BUF_SIZE;
    fp->private_data = buf_pos;
    return 0;
}

static int mydevice_close(struct inode *inode, struct file *fp) {
    printk("close my device\n");
    return 0;
}

static ssize_t mydevice_read(struct file *fp, char __user *_buf, size_t count, loff_t *f_pos)
    {
    const int cpu_id = 0;
    int *buf_pos = fp->private_data;
    struct cpuinfo_x86 *c = &cpu_data(cpu_id);

    if (*buf_pos == BUF_SIZE) {
        unsigned int freq = 0;
        typedef unsigned long (*kallsyms_lookup_name_t)(const char *name);
        kallsyms_lookup_name_t kallsyms_lookup_name;
        static struct kprobe kp = {
            .symbol_name = "kallsyms_lookup_name"
        };
        register_kprobe(&kp);
        kallsyms_lookup_name = (kallsyms_lookup_name_t) kp.addr;
        unregister_kprobe(&kp);
        unsigned int (*aperfmperf_get_khz)(int) = kallsyms_lookup_name("aperfmperf_get_khz");

        if (cpu_has(c, X86_FEATURE_TSC)) {
            if (aperfmperf_get_khz != 0) freq = aperfmperf_get_khz(cpu_id);

            if (!freq) freq = cpufreq_quick_get(cpu_id);
            if (!freq) freq = cpu_khz;
        }

        struct sysinfo mem_info;
```

```
        si_meminfo(&mem_info);

        struct timespec64 uptime;
        ktime_get_boottime_ts64(&uptime);
        timens_add_boottime(&uptime);

        *buf_pos = snprintf(
            buffer,
            sizeof(buffer),
            "========= system =========\n"
            "kernel version: %s %s %s\n"
            "uptime: %lu.%02lu\n"
            "========== cpu ==========\n"
            "vender id: %s\n"
            "cpu family: %u\n"
            "model: %s\n"
            "cpu MHz: %u.%03u\n"
            "cache size: %u KB\n"
            "core id: %u\n"
            "cpu cores: %u\n"
            "========= memory =========\n"
            "MemTotal: %lu\n"
            "MemFree: %lu\n"
            "MemUsed: %lu\n"
            "",
            utsname()->sysname,
            utsname()->release,
            utsname()->version,
            (unsigned long) uptime.tv_sec,
            (uptime.tv_nsec / (NSEC_PER_SEC / 100)),
            c->x86_vendor_id,
            c->x86,
            c->x86_model_id,
            freq / 1000, (freq % 1000),
            c->x86_cache_size,
            c->cpu_core_id,
            c->x86_max_cores,
            pagenum_to_KB(mem_info.totalram),
            pagenum_to_KB(mem_info.freeram),
            pagenum_to_KB(mem_info.totalram - mem_info.freeram)
        ) + 1;
        printk("buf_pos = %d", *buf_pos);
    }

    int copy_size = count;
    if (count > *buf_pos) copy_size = *buf_pos;
    copy_to_user(_buf, buffer, copy_size);

    *f_pos += copy_size;
    int i;
    for (i = copy_size; i < *buf_pos; i++) {
        buffer[i - copy_size] = buffer[i];
    }
    *buf_pos -= copy_size;
```

```
122     printk( KERN_INFO "%s : buf_pos = %d\n", buffer, *buf_pos );
123
124     printk("read my device\n");
125
126     printk("buf_pos = %d", *buf_pos);
127     printk("count = %d", count);
128     printk("copy_size = %d", copy_size);
129
130     return copy_size;
131 }
132
133 static ssize_t mydevice_write(struct file *fp, const char __user *_buf, size_t count, loff_t
    ↪  *f_pos) {
134     printk("write my device\n");
135
136     return count;
137 }
138
139 static struct proc_ops s_mydevice_fops = {
140     .proc_open    = mydevice_open,
141     .proc_release = mydevice_close,
142     .proc_read    = mydevice_read,
143     .proc_write   = mydevice_write,
144 };
145
146 static int mydevice_init(void) {
147     printk("init my device\n");
148
149     struct proc_dir_entry *entry;
150     entry = proc_create(PROC_NAME, S_IRUGO | S_IWUGO, NULL, &s_mydevice_fops);
151     if (entry == NULL) {
152         printk(KERN_ERR, "proc_create\n");
153         return -ENOMEM;
154     }
155
156     return 0;
157 }
158
159 static void mydevice_exit(void) {
160     printk("exit my device\n");
161
162     remove_proc_entry(PROC_NAME, NULL);
163 }
164
165 module_init(mydevice_init);
166 module_exit(mydevice_exit);
```

リスト 5.10: proc ファイルの実装

基本的に module_init や module_exit でモジュールの初期化と終了時の関数を設定すること，構造体に関数ポインタを渡して open, release, read, write 時の動作を定義することは 4 章と同じである．5 章では読み込みのみ対応すれば良いとのことなので write 関数については実装を省略している．以下，それぞれのデバイスのデータの取得について説明する．

5.1.1 kernel version

カーネルのバージョンは utsname という関数から new_utsname という構造体を経由して簡単に参照することができる．[6] 定義自体も include 可能なヘッダに存在するので呼び出すだけで簡単に参照することができる．バージョン名は sysname と release と version に分かれている．

5.1.2 uptime

uptime は timespec64 という構造体の中に格納されるものであると分かった．[4] まず ktime_get_boottime_ts64[1] で uptime に起動時間を格納する．更に timens_add_boottime で boot にかかった時間を足している．[?]

5.1.3 memory

メモリに関する情報は構造体 sysinfo を通して si_meminfo から取得可能である．[7] 構造体 sysinfo にはメモリに関する情報がきれいに全部入っているのでこのまま使用可能である．

[1] マクロで ns_to_timespec64 に置き換わっている

第 6 章

参考文献

参考文献

[1] syscall_64.tbl - arch/x86/entry/syscalls/syscall_64.tbl - Linux source code (v5.17.1) - Bootlin, 2022 年 8 月 4 日閲覧. https://elixir.bootlin.com/linux/v5.17.1/source/arch/x86/entry/syscalls/syscall_64.tbl

[2] unistd32.h - arch/arm64/include/asm/unistd32.h - Linux source code (v5.17.1) - Bootlin 2022 年 8 月 4 日閲覧. https://elixir.bootlin.com/linux/v5.17.1/source/arch/x86/entry/syscalls/syscall_64.tbl

[3] dylanaraps/neofetch: A command-line system information tool written in bash 3.2+ 2022 年 8 月 4 日閲覧. https://github.com/dylanaraps/neofetch

[4] time64.h - include/linux/time64.h - Linux source code (v5.17.1) - Bootlink 2022 年 8 月 4 日閲覧. https://elixir.bootlin.com/linux/v5.17.1/source/include/linux/time64.h#L13

[5] time_namespace.h - include/linux/time_namespace.h - Linux source code (v5.17.1) - Bootlin 2022 年 8 月 4 日閲覧. https://elixir.bootlin.com/linux/v5.17.1/source/include/linux/time_namespace.h#L72

[6] utsname.h - include/linux/utsname.h - Linux source code (v5.17.1) - Bootlin 2022 年 8 月 4 日閲覧. https://elixir.bootlin.com/linux/v5.17.1/source/include/linux/utsname.h#L79

[7] page_alloc.c - mm/page_alloc.c - Linux source code (v5.17.1) - Bootlin 2022 年 8 月 4 日閲覧. https://elixir.bootlin.com/linux/v5.17.1/source/mm/page_alloc.c#L5793