システムプログラミング実験 (OS演習) 第10-1回

品川 高廣
TA　藤原 祐二、松尾 勇気
os-enshu-ta@il.is.s.u-tokyo.ac.jp

前回の補足

- ユーザープロセスを信用しないこと
  - アドレスは不正な値が入っているかもしれない
    - NULLポインタ
    - カーネル内のアドレス
      - PAGE_OFFSET(0xc0000000)より上
  - 安易に memcpy() などするとセキュリティホールになる
    - copy_from_user() などの関数はアドレスの妥当性チェックも行う
- エラー処理も適切におこなうこと
  - エラー時には、負の値を返す
    - -EINVAL等
  - システムコールがエラーを返した場合
    - syscall関数は-1を返し、errnoにそのエラー番号をセットする

本日の内容

- 続々・Linuxカーネル
  - カーネルモジュール
  - キャラクタデバイス
  - コンテキスト
  - 同期処理

- 課題10 (1)

今後の予定

- 7月9日
  - カーネルモジュール
  - キャラクタデバイス
  - 同期機構
  - 課題10(1)

- 7月23日
  - Ethernetデバイスドライバ
  - 課題10(2)

カーネルモジュール

- 動的にカーネルに組み込むことのできるコード
  - メモリの無駄を削減できる
    - 必要なデバイスのドライバのみを組み込むなど
  - 後からカーネルの機能を拡張できる
    - 動的に接続したデバイスのドライバなど
  - Linux では拡張子は .ko
    - 実態は ELF ファイル
  - カーネル本体から何らかのイベントを契機として呼ばれる
    - 「デバイスを開く」「デバイスに対して何か操作する」など
    - モジュールの種類によって定義するものは異なる

カーネルモジュールの例

- helloworld.ko
  - ロードするとカーネルメッセージとしてHello, worldと表示する

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

static int helloworld_init(void)
{
        printk("Hello, world!¥n");

        return 0;
}

static void helloworld_exit(void)
{
}

module_init(helloworld_init);
module_exit(helloworld_exit);
```

カーネルモジュールの組み込み

- 初期化・終了処理
  - `module_init()`, `module_exit()`マクロを使う
    - モジュールのロード・アンロード時に呼ばれる関数名を指定する
    - 関数自体の名前は何でもよいし、static 関数でよい
- ロード・アンロード
  - insmod(8), rmmod(8)コマンドを用いる

```
# insmod helloworld.ko
# rmmod helloworld
```

- ドライバの一覧
  - lsmod(8) コマンド

カーネルモジュールのビルド

- Makefile の例

```
KDIR = (カーネル本体のビルドディレクトリ)
obj-m = helloworld.o

.PHONY: clean install

modules:
	$(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
	$(RM) .*.cmd *.mod.c *.o *.ko Module.symvers modules.order
	$(RM) -r .tmp*
```

- makeすると、helloworld.koが生成される

デバイスドライバ

- 「デバイス」を動作させるためのソフトウェア
  - HDD，グラフィックス，USBなどのハードウェアを制御する
  - 仮想コンソールなどの疑似的なデバイスも含む
  - カーネルモジュールとして作成されることが多い
    - 必要なデバイスのデバイスドライバだけ組み込み得る
- Linuxでは大きく3種類のデバイスに分かれる
  - キャラクタ，ブロック，ネットワーク
  - 演習では「キャラクタデバイス」を作成する
- 「デバイスファイル」を通じてアクセスする
  - 通常 /dev 以下に作られる
  - read/write や ioctl でデバイスを操作する

デバイスファイル（1）

- メジャー番号とマイナー番号の2つの番号を持つ

```
$ ls -l /dev/null /dev/zero
crw-rw-rw- 1 root root 1, 3 2010-06-17 12:16 /dev/null
crw-rw-rw- 1 root root 1, 5 2010-06-17 12:16 /dev/zero
```

デバイスの種類

メジャー番号

マイナー番号

- 番号に応じてカーネル内の処理する関数が決まる
  - 上の例では、マイナー番号に基づいて処理を変える
  - /drivers/char/mem.c ll.893-902（抜粋）

```
static int memory_open(struct inode *inode, struct file *filp)
{
        for (i = 0; i < ARRAY_SIZE(devlist); i++) {
                if (devlist[i].minor == iminor(inode)) {
                        filp->f_op = devlist[i].fops;
```

デバイスファイル（２）

- 登録デバイス一覧は /proc/devices で見れる

```
$ cat /proc/devices
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  4 ttyS
  5 /dev/tty
  5 /dev/console
…
```

- デバイスファイルの作成は mknod (1) コマンドで行う
  - （例）メジャー番号240・マイナー番号 0のキャラクタデバイス

```
# mknod /dev/os-enshu c 240 0
```

キャラクタデバイスドライバ（1）

- デバイスドライバの初期化
  - メジャー番号と対応する関数ポインタの構造体を登録する

```
ret = register_chrdev(OS_ENSHU_MAJOR, "os-enshu", &os_enshu_ops);
```

    - 第1引数はメジャー番号
    - 第2引数は struct file_operations 型の構造体へのポインタ
  - struct file_operations型の構造体を用意しておく

```
static struct file_operations os_enshu_ops = {
          .owner = THIS_MODULE,
          .open = os_enshu_open,
          .release = os_enshu_release,
          .read = os_enshu_read,
          .write = os_enshu_write,
          .ioctl = os_enshu_ioctl,
};
```

    - このデバイスファイルをreadするとos_enshu_read()関数が呼ばれる
    - include/linux/fs.h に完全な定義がある

キャラクタデバイスドライバ(2)

- 各システムコールに対応する関数の実装
  - open, release

```
int (*open) (struct inode *, struct file *);
int (*release) (struct inode *, struct file *);
```

    - それぞれ、open・closeシステムコールに対応すると考えてよい
  - read, write

```
ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
```

    - 第4引数はファイル内の現在の位置
      - 位置の概念が存在する場合は、ドライバが適切に進める
    - バッファ(第2引数)のポインタはユーザー空間なので注意する
  - ioctl

```
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
```

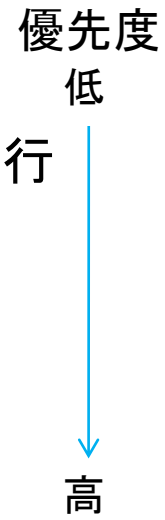
    - read,write以外に、ドライバを直接操作したい場合
    - リクエストコード(第3引数)と引数を一つ(第4引数)とる

キャラクタデバイス・参考

- メジャー番号
  - 空いている番号を適当に使う
  - 動的に確保することもできる
    - alloc_chrdev_region関数
- デバイスファイルの作成
  - mknodで、メジャー番号・マイナー番号を指定して作成
  - udevという仕組みを通じて、モジュールの初期化時に、自動的に作ってもらうこともできる
    - device_create関数

コンテキスト

- カーネルが実行された文脈
  - プロセスコンテキスト（in_interrupt）
    - システムコールなどユーザプロセスで起きたイベントが契機で実行
  - ソフト割り込みコンテキスト（in_softirq）
    - 割り込みの処理のうち時間のかかる部分を後から実行
    - ハードウェア割り込みは有効
  - ハード割り込みコンテキスト（in_irq）
    - 外部ハードウェアからの割り込みが契機で実行
    - ある割り込みハンドラの実行中には同種の割り込みは来ない
      - なるべく処理は少なくする
- プロセスコンテキスト以外では休眠する関数は呼べない
  - schedule関数（後述）を呼び出しうる関数は呼べない



同期

- 割り込みハンドラとの競合
  - プロセスコンテキストで実行中に割り込まれて他コードが実行される
    - シグナルハンドラの場合と同様
  - 競合がおきうる部分では割り込みを禁止して防ぐ
    - ハードウェア割り込み
      - `local_irq_disable, local_irq_enable`関数
    - ソフトウェア割り込み
      - `local_bh_disable, local_bh_enable`関数
- 他プロセッサとの競合
  - 一つのカーネルデータが複数プロセッサから同時にアクセスされる
    - マルチスレッドの場合と同様
  - 競合が起きうる部分では排他制御の仕組みを用いる
    - spinlock
    - mutex
    - RCUなど

排他制御 (1)

- spinlock
  - ロックが取れるまで休眠せずに繰り返し再試行する
  - 割り込みコンテキストでも使用できる
  - include/spinlock.h
    - DECLARE_SPINLOCK(…)
      - 変数の宣言と初期化を同時に行う
    - spin_lock_init
      - 初期化
    - spin_lock, spin_unlock
      - ロックを確保する
  - ロックを確保している間では休眠できない
    - 他のスレッド・割り込みハンドラを長時間待たせてしまう

排他制御 (2)

- spinlock + 割り込み禁止
  - ハードウェア割り込み
    - 当然、ソフトウェア割り込みも禁止される
    - spin_lock_irq, spin_lock_irqsave
    - spin_unlock_irq, spin_lock_irqrestore
  - ソフトウェア割り込み
    - spin_lock_bh / spin_unlock_bh

排他制御 (3)

- mutex
  - ロックを取れなかったら休眠して他のスレッドを待つ
  - プロセスコンテキストのみで使用できる
  - include/linux/mutex.h
    - DEFINE_MUTEX(…)
    - mutex_init
      - 初期化
    - mutex_lock, mutex_trylock, mutex_unlock
    - mutex_lock_interrutible
      - プロセスの状態をINTERRUPTIBLEにする（後述）
      - デフォルトではUNINTERRUPTIBLE
      - mutexを取るまえにシグナルが来た場合、-EINTRを返す

プロセスのスケジューリング

- スケジューラ
  - マルチプログラミングを実現する機能
    - 1つのプロセッサを複数のプロセス（スレッド）で共有
  - 定期的に実行され、次に動作するプロセス（スレッド）を選択する
- schedule関数 (kernel/sched.c) で行う
  - 次に実行するスレッドを選択しコンテキストスイッチを行う
    - 同じスレッドが選択されれば，schedule関数からすぐに戻ってくる
    - 別のスレッドが選択されれば，schedule関数内で別スレッドに切り替わる
      - schedule関数を呼んだスレッドは次にスケジュールされるまで待つことになる
  - システムコールやタイマ割り込み等などのタイミングでも呼ばれる
    - プリエンプションの実現

プロセスの状態

- include/linux/sched.hで定義
  - TASK_RUNNING
    - 実行可能状態
      - 実行中のプロセスや待ち状態のプロセス
    - スケジューラはこの状態のプロセスからのみ選択
  - TASK_INTERRUPTIBLE
    - シグナルによって割り込み可能な待ち状態
      - 比較的長い時間の休眠に用いられる
  - TASK_UNINTERRUPTIBLE
    - シグナルによって割り込めない待ち状態
      - ディスクI/O待ちなど比較的短い時間の休眠

休眠プロセスの管理

- wait queue（include/linux/wait.h）
  - wait queue の生成
    - DECLARE_WAIT_QUEUE_HEAD(wq);
      - wqは変数名
  - wait queue によるイベント待ち
    - wait_event_interruptible (wq, condition)
    - conditionが偽の間、呼び出したプロセスをTASK_INTERRUPTIBLEにしてschedue関数を呼ぶ
  - wait queue のプロセスの起動
    - wake_up_interruptible(&wq)
  - その他
    - wait_event / wake_up (UNINTERRUPTIBLEにする)などがある

waitの実装

- wait_event_interruptibleの実装
  - include/linux/wait.h ll.249-265

```
#define __wait_event_interruptible(wq, condition, ret)              ¥
do {                                                                ¥
        DEFINE_WAIT(__wait);                                        ¥
                                                                    ¥
        for (;;) {                                                  ¥
                prepare_to_wait(&wq, &__wait, TASK_INTERRUPTIBLE); ¥
                if (condition)                                      ¥
                        break;                                      ¥
                if (!signal_pending(current)) {                     ¥
                        schedule();                                 ¥
                        continue;                                   ¥
                }                                                   ¥
                ret = -ERESTARTSYS;                                 ¥
                break;                                              ¥
        }                                                           ¥
        finish_wait(&wq, &__wait);                                  ¥
} while (0)
```

補足

- Kernel Hacking config
  - カーネルハックに便利な設定がたくさんある
  - Kernel Hacking > Kernel Debugging
    - ONにするとデバッグに必要なオプションが出現する
  - Spinlock debugging: sleep-inside-spinlock checking
    - 休眠する関数を呼び出すと警告とスタックトレースを表示
  - ものによっては、動作が重くなる・コンパイル時間が増える・バイナリサイズが増える等の副作用もあるので注意

第10回課題 (1)

課題A (optional)

- カーネル内で次のような同期機構を作成せよ
  - 型 : ose_event_t
    - 内部にatomicに操作されるboolean値をもつ
    - 明示的にset/resetされるまでその値を保持する
  - 操作関数
    - 引数は任意
    - create_event
      - eventの初期化
    - wait_on_event
      - eventの値がtrueになるまで待つ
    - set_event
      - eventの値をtrueにする。待っているスレッドがいれば起こす
    - reset_event
      - eventの値をfalseにする
    - destroy_event
      - eventの使用を終了する
  - 注意
    - 割り込みハンドラでもset_event, reset_eventが利用できるようにせよ
    - 型名、関数名等は好きなようにつけてよい

課題A (optional) : 使用例

- 以下のような形で使う
  - 細かい仕様は任意

```
ose_event_t ev;

void interrupt_handler(void){
    /* put a packet to the buffer */
    ...
    set_event(&ev);
}

int ose_read(char *buf, int size){
    int ret;
    while (!packet_buffer_empty()){
        if((ret = wait_event(&ev)) < 0){
            return ret;
        }
        reset_event(&ev);
        if((ret = read_packet_buffer(buf, size)) > 0){
            return ret;
        }
    }
    ...
}
```

課題B (optional)

- 課題Aで作成した機構をユーザープロセスからも使えるようにせよ
  - キャラクタデバイスを作成する
    - readをwait_event, writeをset_event等に対応させる
  - マルチスレッド・マルチプロセスで扱えるようにすること
    - 名前の概念をどこかにつける必要がある
    - デバイスファイルの名前を変える（/dev/event0, event1….）
    - ioctlで指定する、など
  - メモリリークが生じないようにせよ
    - 使っているプロセスがすべていなくなったらメモリを解放する
    - open, releaseで参照カウントの操作をすればよい
  - （さらにoptional) select, pollに対応せよ

課題B (optional) : 使用例

- 以下のような形（ユーザープロセス）で用いる

```
int event_fd;

int main(void){
   ...
   event_fd = open(“/dev/ose_event”, O_RDWR);
   ...
   if(fork() == 0){
       read(event_fd, buf, 1);
       printf(“Parent’s work finished.¥n”);
   }else{
       calc();

       write(event_fd, buf, 1);
       waitpid(-1, NULL, WUNTRACED);
   }
   close(event_fd);
}
```

課題C~

- 次回につづく

第10回課題の締切

- 課題Cのみ必須（次回出題）
- 締切：2012年8月31日(金) 23:59 (JST)

- Webで提出
  - https://report.il.is.s.u-tokyo.ac.jp/os2012/

- レジュメ
  - https://report.il.is.s.u-tokyo.ac.jp/os2012/resume/

- 途中の場合でも**必ず**締め切りまでに提出すること
  - 提出は単位の**必要条件**

質問など

- 提出システム上の掲示板
- メール
  - os-enshu-ta@il.is.s.u-tokyo.ac.jp
- 石川研究室で直接
  - 理学部7号館5階の507号室