unix実験

テーマ：Unix Kernel

チーム名「春の雪」

メンバー
兼城　賢仁 (015717B)
比嘉　武　 (025747J)
大城　丈剛 (035712A)
国吉　貴文 (035718K)
任　家林　 (035739B)
宮平　大輔 (035757A)
饒平名隆一 (035763E)

1　目的

この実験では、Unix Kernel に関するプログラムを行う。
C 言語に関する深い知識と、OS の既往の理解、そして、大量のソースを短い時間で理解する能力が必要である。

2　実験方法

ここでは、Linux の kernel を使用する。本実験では、pw を使用した。

3　level1:Linux kernel を構築する

3.1　Linux Kernel を構築するには

Linux kernel を構築するためには、まず kernel の設定を行います。
kernel のソースがある/usr/src/linux/に移動し、そこで設定をします。
Linux では、以下の 3 つの方法があります。

- make config(make oldconfig)
  make config は最もオーソドックスな設定コマンドで、どんな環境でも利用出来ます。
  カーネルの設定について 1 つ 1 つ [y/n] で回答する形式の問答が続いて、カーネルの設定を行います。
  カーネルの隅々までよく知っている場合は手軽ですが、普通の人には、つらいものがあります。

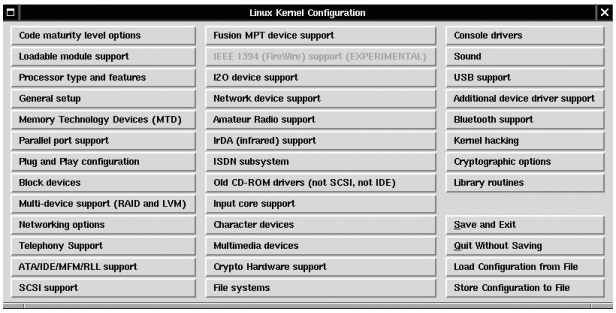
  以前のバージョンの.config ファイルを雛形として利用していて、
  以前のバージョンと今のバージョンの差分の所だけを取り出して設定したい場合には、makeoldconfig を代わりに利用することができます。
  ユーザインターフェースは make config と全く同じですが、
  以前の バージョンで設定されているところは Skip されて、
  新しい項目だけ回答を求められます。

- make menuconfig
  通常は、make menuconfig を使うほうが良いです。
  ncurses というパッケージを導入してある環境ならば、どんな環境でも利用出来ます。
  インターフェースは、メニューが表示されて選択する形式です。

カスタマイズの必要がないときは、これで一旦開いて、
何も変更せずにセーブすれば、.config ファイルが作成されます。

- make xconfig X Window System が使える環境で、tcl/tk が導入されていれば、利用できます。
  pw では、3 つの方法全て実行できましたが、ここでは、make menuconfig を使って説明します。
  root になり、/usr/src/linux でコマンドを打つと、

```
[root@pw019 ~]% make xconfig
```



  このような画面が出てきて、設定したい項目を選択し、設定していきます。
  今回は、実験 Level3 で使うリモートデバッグを on に変更して save します。
  すると、/usr/src/linux/に.config が出来上がります。

カスタマイズが終わったら、その変更をカーネルツリー全体に反映させます。
以下のコマンドを続けて入力します。

```
[root@pw019 ~]% make dep
[root@pw019 ~]% make clean
```

make dep は、カーネルソース中に相互依存ツリーを構築します。
これらの相互依存は設定時に選んだオプションに影響されます。
make clian は、以前のカーネルの構築時から残っていて、現在は必要でないファイルを取り除きます。

次に、カーネル本体を make します。

```
[root@pw019 ~]% make bzImage
[root@pw019 ~]% /sbin/installkernel 2.4.27.new arch/i386/boot/bzImageSystem.map
```

make bzImage を実行すると、vmlinux と System.map と arch/i386/boot/bzImage が作成されます。
vmlinux と arch/i386/boot/bzImage は、カーネル本体 (カーネルイメージ) です。
vmlinux は非圧縮のもので、bzImage の方が圧縮されたものです。
通常は、bzImage の方が使われます。
System.map は、vmlinux から symbol 情報を取り出したものです。
symbol とは、c 言語のオブジェクトファイルの symbol です。
つまり、System.map は、カーネルが利用できるモジュールについての情報が格納されたファイルと考えられます。

/sbin/installkernel を実行すると、arch/i386/boot/bzImage を vmlinux-2.4.27.new に改名し/boot/に置きます。
リブート時にはこれを読み込みます。

カーネル本体を make したら、モジュールを make します。

```
[root@pw019 ~]% make modules
```

ここまでできたら、後はインストールするだけです。

```
[root@pw019 ~]% make modules_install
[root@pw019 ~]% make install
```

make modules_install を実行すると、/lib/modules/ ’ リリース’ 以下にモジュールが展開されます。

最後に、ブート時の変更をします。/etc/の lilo.conf を書き換えます。
lilo.conf は以下の様になっています。

```
[root@pw019 /etc]%cat lilo.conf
prompt
timeout=50
default=linux
boot=/dev/hda
map=/boot/map
```

```
install=menu
message=/boot/message

image=/boot/vmlinuz-2.4.26-0vl15
        label=linux
        initrd=/boot/initrd-2.4.26-0vl15.img
        read-only
        root=/dev/hda1
        append=" resume2=swap:/dev/hda2"
```

これを以下の様に変更します。

```
[root@pw019 /etc]% cat lilo.conf
prompt
timeout=50
default=new
boot=/dev/hda
map=/boot/map
install=menu
message=/boot/message

image=/boot/vmlinuz-2.4.27.new
        label=new
        read-only
        root=/dev/hda1
        append=" resume2=swap:/dev/hda2"
```

変更点は、default と label の名前を変更しています。
起動時に変更されているかこれで確認できます。
次に、image を新しく作ったものに変更し、これを読み込ませます。
ここでは、vmlinuz-2.4.26-0vl15 から vmlinux-2.4.27.new に変更し、
読み込ませています。
そして、initrd の行を削除しています。
これは、initrd で読みこまれるファイルを見つけることが出来なかったから
です。
これでブート時の変更は終わりです。後は、この変更を反映されるように、
`[root@pw019 /etc]% /sbin/lilo -v`
として、エラーが無ければ終わりです。

これで、変更は終了しましたので、リブートしてみます。
`[root@pw019 ~]% reboot`

これで正しく起動されれば kernel の再構築は終了です。

kernel が正しく動いている証拠として、linux のバージョンを調べてみると、

```
[root@pw019 ~]% uname -r
2.4.27-0vl7
```

となり、他の pw のバージョンが 2.4.26 から、正しく実行されたことが分かります。

3.2 考察

この Level1 の実験は相当苦労しました。kernel を構築すると、
pw が壊れる可能性があったので、他の Level から実験することにしましたが、他の実験 (Level2 以外) 全て、kernel を再構築しなければ出来ない事が分かり、とても時間をロスしました。
逆に、Level1 が出来ると、他の Level はスムーズに出来ました。
最初は恐ろしく思える実験ですが、実際に実行してみると 案外簡単な実験であることが分かりました。

4 level2:open system call を処理する部分を見つけよ

システムコールとは、オペレーティングシステムの機能を呼び出すために使用される機構のこと。
実際のプログラミングにおいては OS の機能は、関数（ＡＰＩ）呼び出しによって実現されるので、OS の備える関数（ＡＰＩ）のことを指すこともあります。
システムコールは、多くの場合、ソフトウェア割り込みによって実行されます。
ソフトウェア割り込みを用いて CPU の動作モードを遷移させることによって、通常のアプリケーションプログラムからはアクセスできない保護されたメモリ領域にアクセスすることや、保護されたレジスタを操作すること、また、自ら CPU の動作モードを変更することなどが可能になります。

4.1 アプリケーションが呼び出すライブラリ

アプリケーションがライブラリを呼び出す際、つまり、プログラムの make 時には、リンカ (ld) はまず/usr/lib/libc.so にアクセスします。このファイルは、以下のような ld のスクリプトになっています。

```
/* GNU ld script
    Use the shared librarys but some functions are only in
    the staticc library. so try that secondarily.        */
OUTPUT_FORMAT(elf32-i386)
GROUP(/lib/libc.so.6 /usr/lib/libc_nonshared.a )
```

これは、動作時の共有ライブラリとして探しに行くべきファイルの名前が示されています。よって、アプリケーションが呼び出すライブラリは、/lib/libc.so.6にあります。

4.2 ライブラリが呼び出す trap

Linux のシステムコールは、レジスタに引数を設定して int 0x80 によるソフトウェア割り込みで呼び出します。
/usr/src/linux/init/main.c では、カーネルの起動部分で割り込みテーブルの設定をしています。

```
369 printK("kernel command line : %\n".saved_command_line);
370 parse_options(command line);
371 trap_init();
372 init_IRQ();
373 sched_init();
```

371 行目にある trap_init() でシステムコールの入り口が設定されます。それを、実際に設定している部分は、/usr/src/linux/arch/i386/kernel/traps.cにあり、以下のようになっています。

```
957 void __init trap_init(void)
958 {
              ...
989 set_system_gate(SYSCALL_VECTOR,&system_call);
```

SYSCALL_VECTOR は、別のファイルで 0x80 と設定されています。

4.3 trap を受け取り振り分ける部分

ライブラリから trap を受け取り、システムコールを振り分ける部分は、/usr/src/linux/arch/i386/kernel/の entry.S にあります。
entry.S の中は、以下のように実装されています。

```
222 ENTRY(system_call)
223 push %eax
```

```
224 SAVE_ALL
225 GET_CURRENT(%ebx)
226 testb $0x02.tsk_ptrace(%ebx)
227 jne tracesys
228 cmpl $(NR_syscalls) .%eas
229 jae badsys
230 call * SYMBOL_NAME(sys_call_table)(.%eax.4)
231 movl %eax.EAX(%esp)
```

230 行目の `call * SYMBOL_NAME(sys_call_table)(.%eax.4)` では、eax に指定されたシステムコール番号を入れることで、処理を振り分けます。

4.4 実際に kernel 内部で処理する部分

システムコール open のプログラムの場所は、/usr/src/linux/fs/open.c にあります。他のシステムコールのプログラムも、/usr/src/linux-2.4.27/fs/ や/usr/src/linux/kernel/等にあります。

4.5 考察

この Level 2 の実験で、システムコールやライブラリ関数について色々と調べてみた結果、ライブラリ関数が内部でシステムコールを呼び出していることが分かりました。
そこで、システムコールを使わずにライブラリ関数を使うメリットについて調べてみたら、以下のようなことが挙げられました。

- 移植性の向上
- バッファリングによる速度向上
- 利便性向上（システムコールを使うよりライブラリを使った方が簡単かつ短く書ける）

つまり、プログラムを書く際は、できるだけライブラリ関数を使用した方が良いプログラムになると思います。

5 level3:gdb のリモートデバッグに関して調べ、実際に動作させてみよ

5.1 GDB とは

GDB は GNU デバッガーとも呼ばれる。GNU とは、FSF(Free Software Foundation) が進めている UNIX 互換ソフトウェア群の開発プロジェクトの総称 であり、フリーソフトウェアの理念に従った修正・再配布自由な UNIX 互換シス テムの構築を目的としている。

GDB プログラムを、コマンド・ラインまたは DDD(Data Display Debugger) な どのグラフィカル・ツールから実行することによりプログラム中のエラーを探 り出すことができる。

GDB は、通常ユーザー・プログラムのデバッグに使われ るが、リモートデバッグ機能を用いれば OS のカーネルのデバッグにも使用する ことができる。

5.2 リモートデバッグとは

リモートデバッグは通常の方法で GDB を実行させることができないマシン上の プログラムをデバッグする場合に用いられる機能である。

具体的には、OS のカーネルのデバッグをする場合や、自身で GDB を十分に動かせるだけの機能を持 たない OS を使用した小規模なシステム上のプログラムのデバッグを行う場合に 用いられる。

リモートデバッグを行う際は、デバッグをしたいマシンとは別に GDB を実行で きる環境を持ったマシンを用意して接続し、GDB を実行できるマシン側から GDB を実行できないマシン側のデバッグをする。

この際の通信はそれぞれのマシン のシリアルポートをシリアルケーブルで接続して行われる。

5.3 リモートデバッグを行う際の条件

リモートデバッグを実行するためには、次に示す条件を満たす必要がある。

1. GDB を実行するマシンにデバッグ対象のプログラムを実行する環境が整っていること。

2. 相手のマシンとシリアルポートを通じて通信できる準備をする必要がある。GDB は通信プロトコルを標準装備しているのでホスト側は特に準備することは ないが、ターゲット側ではデバッグ対象のプログラムに通信用プロトコルを実 装したサブルーチンをリンクする必要がある。このサブルーチンを含むファイ ルをデバッグスタブと呼ぶ。デバッグスタブはターゲット側マシンのアーキテ クチャに固有なのでいくつか種類があり、GDB とともに配布されている。

5.4 デバッグスタブの機能

デバッグスタブは次の 3 つのサブルーチンを提供する。デバッグ対象のプログ ラムの先頭付近には次のサブルーチンを記述した行を追加してやる必要がある。

- `breakpoint`
  デバッグ対象のプログラムにブレイクポイントを持たせるルーチン。
- `handle_exception`
  デバッグ対象のプログラムがブレイクポイントなどで停止すると呼び出される。デバッグ対象のプログラムを制御し、ホスト側の GDB と通信を行う。GDB が必要とする情報を送り、ターゲット側のマシンで GDB の代理の働きをする。
- `set_debug_traps`
  デバッグ対象のプログラムが停止したときに `handle_exception` が実行されるように設定するルーチン。

5.5 実際に動かしてみる

はじめに、http://kgdb.linsyssoft.com/quickstart.htm の手順に従いリモートデバッグを試みたが、項目「Compiling the kernel on the development machine」の項目 3 でカーネルコンフィギュレーションを行うために xconfig、oldconfig を実行したが、コンフィグ画面が表示されず失敗した。

参考ページでは、カーネルのソースをダウンロードしてきてカーネルのイメージを作っていたが、カーネルハッキングの項目を有効にしたカーネルのイメージを作成できればページ通りの手順を踏まなくてもよいだろうと思われる。

そこで、現在使用しているマシンのカーネルハッキングの設定を変更して再構築したカーネルのイメージを使用することにした。
その結果、カーネルのイメージは作成できたものの、使用しているマシンで

はカーネルの再構築の際に grub ではなく lilo を用いるため、項目 7 の grub に関する設定の箇所を飛ばしてカーネルを再構築した。
その結果、成功すれば、ターゲットマシンは接続待ちのメッセージを出力して停止するはずなのだが、通常通りに起動して失敗した。

5.6 課題

カーネルの再構築の際に lilo でなく grub を用いて再起動できるように grub について調べる必要がある。

6 level4:新しいシステムコールを提案し、実装してみよ。

system call 追加方法

6.1 linux/include/asm(-i386)/unistd.h の修正

```
#define __NR_exit                 1
#define __NR_fork                 2
...
```

となっているところがあるので、それの末尾に

```
#define __NR_(システムコール名)（空き番号）
```

を挿入。この時の空き番号がシステムコール番号になるので憶えておくこと。

6.2 linux/arch/i386/kernel/entry.S の修正

```
.data
ENTRY(sys_call_table)
.long SYMBOL_NAME(sys_ni_call)          /* 0 */
.long SYMBOL_NAME(sys_exit)
.long SYMBOL_NAME(sys_fork)
...
```

次のようなテーブルがあるので、既にあるシステムコールのコメントを元に先程のシステムコール番号と一致する場所に

```
.long SYMBOL_NAME(sys_(システムコール名))
```

を挿入。
また、Linux2.6.XX の場合、entry.S にはシステムテーブルに関するテーブルが無く、その代わりに linux/arch/i386/kernel/syscall_teble.S にそれに近いことが記述されているのでシステムコール番号と一致する場所に

.long sys_(システムコール名)

を挿入することで設定出来る。

6.3 linux/arch/i386/kernel/Makefile の修正

```
obj-y   := process.o semaphore.o signal.o entry.o traps.o irq.o vm86.o
                ptrace.o i8259.o ioport.o ldt.o setup.o time.o sys_i386.o
                pci-dma.o i386_ksyms.o i387.o bluesmoke.o dmi_scan.o
```

のようになっている箇所があるのでここに (システムコール名).o を追記。

6.4 ヘッダファイルの作成

“(システムコール名).h” というファイルを次の書式で linux/include/linux ディ レクトリに記述。

```
#ifndef __LINUX_(システムコール名を大文字で)_H
#define __LINUX_(システムコール名を大文字で)_H

#include #include #include
(※)

#endif
```

※ここには_syscallX (X はシステムコールのとる引数の数が入る、0 の場合は省略) という関数が入る。
例えば X=2 の場合の書式はこうなる。

_syscall2((システムコールの関数型),(システムコールの名前),(引数 1 の変数型),(引数 1 の名前),(引数 2 の変数型),(引数 2 の名前));

これは、システムコールを呼び出すための関数 (＝ライブラリ関数) を作るためのマクロである。

6.5　システムコール本体の作成

“(システムコール名).c” というファイルを次の書式で linux/arch/i386/kernel ディレクトリに記述。

```
#include
asmlinkage (関数型) sys_(システムコール名)(引数) {
/*システムコール実行部*/
}
```

ここまで完了したらカーネルを再構築することで追加したシステムコールが使用可能になる。

ちなみに今回の実験で使用するシステムコールとそれの検証用のプログラムは以下の通り。

＃与えられた int 数値の三倍の値を返すシステムコール sys_arg_triple()

○ arg_triple.h

```
#ifndef __LINUX_ARG_TRIPLE_H
#define __LINUX_ARG_TRIPLE_H

#include #include
_syscall1(int, arg_triple, int, arg1);

#endif
```

○ arg_triple.c

```
#include
asmlinkage int sys_arg_triple(int arg1) {
  int j = 3;
  j = j * arg1;
  return j;
}
```

○ syscalltest.c

```
#include #include #include

int main() {
   int i = 3;
   printf("test arg_triple(%d) = %d ?n", i, arg_triple(i));
```

```
   return 0;
}
```

システムコールをテストするプログラムには必ずシステムコールのヘッダファイルを include させること。

6.6 syscalltest.c 実行結果

```
[j03063@pw039 ~/info4]% ls
syscalltest.c
[j03063@pw039 ~/info4]% gcc -o test.out syscalltest.c
[j03063@pw039 ~/info4]% ./test.out
Incorrectly built binary which accesses errno or h_errno directly. Needs
to be fixed.
test arg_triple(3) = 9
[j03063@pw039 ~/info4]%
```

設定や書式が悪かったのか、エラーが発生している。
時間が無かったので原因究明はしていない。
ただ、システムコール自体は正常に動いている模様。

6.7 考察

今回の実験は単純なプログラムを組むだけだったので難易度は低めだったものの、カーネルの内部をいろいろと見て回らないといけなかったため、 時間がかかってしまった。
そして、今回の実験で Linux カーネルというものが多くのプログラムの集合体であり、それを修正するのもカーネルに機能を追加するのもそれなりの技量があれば可能だということがよくわかった。

7 level5:何か、OS 内部の情報を取得するシステムコールを作成せよ。

7.1 システムコールを作成

OS の内部情報として、プロセス ID 情報を取得するシステムコールを作成する。

****

```
test.c
*****************
#include #include
int sys_test(char* comment)
{
  int pid;

  pid = current->pid;
  if(comment)
    printk("syscall_test: %s pid = %d?n", comment, pid);
  return 0;
}
```

システムコール番号の定義、エントリへの追加等を level4 と同様に行う。

7.2 AP へインターフェースの記述を行う

```
*************
test.h
************
#ifndef __TEST_H
#define __TEST_H

#include #include #include
#ifndef __KERNEL__
_syscall1(int, test, char*, comment)
#endif

#endif
```

AP へのインタフェースの記述を行う。
ここでは、AP から syscall を使ってシステムコールを行うのではなく, ”linux/unistd.h” で定義されている syscallX() マクロを使う方法をとる。
(X にはシステムコールの引数の数に応じて 0〜5 が入る)。
syscallX() の引数には「システムコールの戻り値の型、システムコール名、第 1 引数の型、第 1 引数の名前、...」を入れる。

7.3 テストプログラムの作成

カーネルのコンパイルを行った後、システムコールを追加したカーネルが動作確認の為のプログラムを作成する。

```
******************
test2.c
******************
#include #include
int main(int argc, char** argv)
{
  if(argc == 2)
    test(argv[1]);
  return 0;
}
```

7.4 テストプログラムの実行結果

作成したテストプログラムを実行すると、/var/log/messages において以下のようなメッセージが得られる。

```
[root@pw039 j03012]% ./a.out foo
[root@pw039 j03012]% tail -1 /var/log/messages
Jan 12 20:09:09 pw039 kernel: syscall_test: foo pid = 1754
```

これより、作成したシステムコールの動作を確認することができる。

7.5 考察

level4 と同様の手順を踏んで、システムコールを作成し動作の確認を行った。
test.c において、pid = current-¿pid でプロセス ID を取得する。

8 level6:ヌル・デバイスドライバ (何もしないデバイスドライバ) を作成せよ

8.1 デバイスドライバとは

デバイスドライバとは、ハードウェアコントローラを操作あるいは管理する ソフトウェアであります。
Linux カーネルのデバイスドライバは、本質的に特権を与えられ、メモリに常駐する、低レベルハードウェアの処理ルーチンから成る共有ライブラリであります。
担当するデバイスの特殊性に対処するのが、Linux のデバイスドライバの役割であります。

また、Linux ではデバイスはキャラクタ型とブロック型があります。
キャラクタ型が 1 バイト単位の細かい読み書きが可能なのに対して、ブロック型はブロックというデータの塊を単位に読み書きをします。
ブロック型のデバイスは mount するところでファイルシステムに組み込むことが可能で、またディスクキャッシュも働きます。

8.2 ヌル・デバイスドライバのプログラム作成

ヌル・デバイスドライバのプログラムは以下の様になっています。

```
/*必要なヘッダファイルをインクルード*/
#include #include #include
/*モジュールについての説明*/
MODULE_DESCRIPTION( "Sample null device driver." );

/*カーネルモジュールのライセンスを表す文字列を引数として指定*/
MODULE_LICENSE( "GPL" );

/*メジャー番号を指定*/
static int   devmajor = 241;
static char* devname  = "newnull2";
static char* msg      = "module [newnull2.o]";


/*モジュール内の各関数とユーザプロセスからの操作を関連づけるためには、

linux/fs.h 内の struct file_operations 型*/
/*構造体を作成し、カーネルに登録する必要があります。今回はヌルにしますの
で、何も設定していません*/

static struct file_operations newnulldev_fops =
{
        owner   : THIS_MODULE,        /*カーネルモジュール自身の情報を初
期化します。*/
};

/*自分自身をキャラクタデバイスドライバとしてカーネルに登録するために
```

```
register_chrdev を実行しています。*/

int
init_module( void )
{
        if ( register_chrdev( devmajor, devname, newnulldev_fops )
        ) {
                printk( KERN_INFO "%s : register_chrdev failed\n" );
                return -EBUSY;
        }

        return 0;
}

/*カーネルに登録されている情報を削除するために、unregister_chrdev を
実行
しています。*/

void
cleanup_module( void )
{
        if ( unregister_chrdev( devmajor, devname ) ) {
                printk( KERN_INFO "%s : unregister_chrdev failed\n" );
                /* 対策無し ... */
        }

        printk( KERN_INFO "%s : removed from kernel\n", msg );
}
```

8.3 ヌル・デバイスドライバの追加方法

プログラムを作り終えたら、ヌル・デバイスをカーネルに追加します。
まず、作成したプログラムを以下のようにコンパイルし、カーネルモジュー
ルを作成します。

```
[j03018@pw022 ~]% gcc -DMODULE  -D__KERNEL__ -O -c newnull2.c
```

コンパイルすると、カーネルモジュールの newnull2.o が作成されます。

次に、ユーザプロセスから実行する為には、デバイスファイルを作成しなけ

ればいけないので、デバイスファイルを作成します。
デバイスドライバとデバイスファイルは、メジャー番号とマイナー番号と呼ばれる数値で結び付けられています。
メジャー番号とマイナー番号については後で説明します。

ここからは、root で行います。

```
[j03018@pw022 ~]% mknod /dev/newnull2 c 241 0
```

```
[j03018@pw022 ~]% chmod 0666 /dev/newnull2
```

mknod で newnull のメジャー番号を 241、マイナー番号を 0 としています。
ローカルで使用可能な番号が 240〜254 なので 241 番にしました。

最後に、作成された newnull.o をカーネルへ組込みます。
組込みには、insmod コマンドを使います。

```
[j03018@pw022 ~]% /sbin/insmod nuwnull2.o
```

これで何も表示されなければ、カーネルへ無事組込まれています。

ヌル・ドライバデバイスなので、なにも出来ませんが、以下のコマンドを打つと

```
[j03018@pw022 ~]% cat /proc/devices
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 ttyS
  5 cua
  7 vcs
 10 misc
 14 sound
128 ptm
136 pts
162 raw
180 usb
226 drm
241 newnull2

Block devices:
```

```
  2 fd
  3 ide0
 22 ide1
```

となり、241 番に newnull2 が追加されていることが分かります。

ノード番号を指定するファイルは、/usr/include/linux/の major.h です。
もしくは、先ほどの/proc の devices にも番号が少し指定されています。

major.h は以下のようになっています。

```
#ifndef _LINUX_MAJOR_H
#define _LINUX_MAJOR_H

...
...

#define MAX_CHRDEV      255
#define MAX_BLKDEV      255

#define UNNAMED_MAJOR   0
#define MEM_MAJOR       1
#define RAMDISK_MAJOR   1
#define FLOPPY_MAJOR    2
#define PTY_MASTER_MAJOR 2
#define IDE0_MAJOR      3
...
...
#define MSR_MAJOR               202
#define CPUID_MAJOR             203

#define OSST_MAJOR      206     /* OnStream-SCx0 SCSI tape */

#define IBM_TTY3270_MAJOR       227     /* Official allocations now */
#define IBM_FS3270_MAJOR        228
...
...
static __inline__ int ide_blk_major(int m)
{
        return IDE_DISK_MAJOR(m);
}
```

```
#endif
```

ここで指定されているノード番号は、上で少し出てきましたが、メジャー番号と呼ばれています。
この他にもマイナー番号というのもあります。それぞれ、以下のような意味を持っています。

- メジャー番号は、デバイスドライバがそれぞれ固有に持っている番号です。
- マイナー番号は、そのデバイスドライバで利用可能なデバイスの内の、どの デバイスを使うのかを指定しています。これは一つのデバイスドライバで複数 のデバイスが利用可能な場合に識別のために用いられる番号です。

8.4 考察

今回の実験を進めていて、カーネルモジュールをカーネルに組込もうとすると以下のエラーメッセジが表示されました。

```
newnull2.o: kernel-module version mismatch
        chardev.o was compiled for kernel version 2.4.27-0vl7
        while this kernel is version 2.4.26-0vl15.
```

これは、カーネルモジュールの対応カーネルバージョンと pw の Linux のカーネルバージョンが違っていて組込めないとのことでした。
それで色々調べてみると、pw のカーネルは、/usr/src/linux-2.4.27 以下に置かれていました。
しかし、コマンドで調べてみると

```
[j03018@pw022 ~]% uname -r
2.4.26-0vl15
```

と表示され実際は linux-2.4.26 が動いていることが分かりました。
それで、他の実験で設定し直した Linux のバージョンが 2.4.27 だったので、急遽、その Linux-2.4.27 で起動させた所、無事にカーネルに組込むことが出来ました.
pw の内部のバージョンは、合わせる必要があると思いました。

9　動くキャラクタデバイスドライバを作成せよ。

9.1　デバイスドライバを作成

　open/close/read/write システムコールに対応したキャラクタデバイス chardev.c を作成します。

chardev.c プログラム

```
/*必要なヘッダファイルをインクルードしています。*/
#include     /* copy_from_user, copy_to_user */
#include #include        /* inode, file, file_operations */
#include    /* printk */
#include #include
...

/*今回はメジャー番号を 242 番とします。*/
static int   devmajor = 242;

...

/*ユーザプロセスから書き込まれたデータを保存し、ユーザプロセスから読
み込
める*/
/*様にするための内部バッファ32 バイトを宣言しています。*/
#define MAXBUF 32
static unsigned char devbuf[ MAXBUF ];
static int buf_pos;

...

/*
 * open()
 * 1 ユーザプロセスのみがデバイスファイルの  open  に成功します。
 *
 * ユーザプロセスからの open に対応する関数です。linux/fs.h 内の
 * struct file_operations  のメンバである open 関数ポインタの形式で
 * 定義する必要があります。成功時には 0 を、何らかのエラー終了時
 * には linux/errno.h 内の適切なエラー番号を選び、負の数値に変換して
 * 返却しています。
 */
```

```
static int
chardev_open( struct inode* inode, struct file* filp )
{
        printk( KERN_INFO "%s : open()  called\n", msg );

        spin_lock( chardev_spin_lock );

        if ( access_num ) {
                spin_unlock( chardev_spin_lock );
                return -EBUSY;
        }

        access_num ++;
        spin_unlock( chardev_spin_lock );

        return 0;
}

/*
 * release()
 * 使用ユーザプロセス数をクリアします。
 *
 *ユーザプロセスとデバイスドライバの関連付けが解放される際に
 *に実行されます。一般的にはユーザプロセスが close を実行した際
 * に実行されますが、ユーザプロセスがエラー終了してしまった際
 * などにも実行されることで、資源が解放されます.
 */
static int
chardev_release( struct inode* inode, struct file* filp )
{
        printk( KERN_INFO "%s : close() called\n", msg );

        spin_lock( chardev_spin_lock );
        access_num --;
        spin_unlock( chardev_spin_lock );

        return 0;
}
```

```
/*
 * read()
 * ユーザプロセスに対して内部バッファの内容を転送します。
 * 転送した内容は内部バッファから消えます。
 */

static ssize_t
chardev_read( struct file* filp, char* buf, size_t count, loff_t* pos )
{
        int copy_len;
        int i;

        printk( KERN_INFO "%s : read()  called\n", msg );

        if ( count > buf_pos )
                copy_len = buf_pos;
        else
                copy_len = count;

/*buf にコピー先アドレス、devbuf にコピー元アドレスが指定されていま
す。*/
/*戻り値はコピーされていないバイト数であるため、正常に終了した場合は
0*/
/*が帰ります．*/
        if ( copy_to_user( buf, devbuf, copy_len ) ) {
                printk( KERN_INFO "%s : copy_to_user failed\n", msg );
                return -EFAULT;
        }

/*コピーした後は、コピーしたデータ長に応じて内部バッファ内の*/
/*データを先頭につめています。*/

        *pos += copy_len;

        for ( i = copy_len; i < buf_pos; i ++ )
                devbuf[ i - copy_len ] = devbuf[i];

        buf_pos -= copy_len;
        printk( KERN_INFO "%s : buf_pos = %d\n", msg, buf_pos );
```

```
        return copy_len;
}

/*
 * write()
 * ユーザプロセスから転送された内容を内部バッファに書き込みます。
 */
static ssize_t
chardev_write( struct file* filp, const char* buf, size_t count, loff_t*
pos )
{
        int copy_len;

        printk( KERN_INFO "%s : write() called\n", msg );

        if ( buf_pos == MAXBUF ) {
                printk( KERN_INFO "%s : no space left\n", msg );
                return -ENOSPC;
        }

        if ( count > ( MAXBUF - buf_pos ) )
                copy_len = MAXBUF - buf_pos;
        else
                copy_len = count;

/*ここでは、単にbufからコピーするわけには行きませんので、            */
/*copy_from_userという関数を使います。
*/
/*devbuf + buf_posにコピー先アドレス、bufにコピー元アドレスを     */
/*指定します。また、copy_lenにコピーするバイト数を指定します。*/
        if ( copy_from_user( devbuf + buf_pos, buf, copy_len ) ) {
                printk( KERN_INFO "%s : copy_from_user failed\n", msg );
                return -EFAULT;
        }

        *pos    += copy_len;
        buf_pos += copy_len;
```

```
        printk( KERN_INFO "%s : buf_pos = %d\n", msg, buf_pos );

        return copy_len;
}

/*モジュール内の各関数とユーザプロセスからの操作を関連づけています。*/
/*今回は、read と write、open、release の 4 つを使っていますので 4 つ
関連付け*/
/*をしています。*/

static struct file_operations chardev_fops =
{
        owner   : THIS_MODULE,
        read    : chardev_read,
        write   : chardev_write,
        open    : chardev_open,
        release : chardev_release,
};

...
...

/* End of chardev.c */
```

9.2 カーネルに登録

上で少し書きましたが、モジュール内の各関数とユーザプロセスからの操作を関連づけるためには、linux/fs.h 内の `struct file_operations` 型構造体を作成し、カーネルに登録する必要があります。
カーネル 2.4.27 の linux/fs.h では以下の様に定義されています。

```
884:struct file_operations {
885-    struct module *owner;
886-    loff_t (*llseek) (struct file *, loff_t, int);
887-    ssize_t (*read) (struct file *, char *, size_t, loff_t *);
888-    ssize_t (*write) (struct file *, const char *, size_t, loff_t
*);
889-    int (*readdir) (struct file *, void *, filldir_t);
890-    unsigned int (*poll) (struct file *, struct poll_table_struct
*);
```

```
891-    int (*ioctl) (struct inode *, struct file *, unsigned int,
unsigned long);
892-    int (*mmap) (struct file *, struct vm_area_struct *);
893-    int (*open) (struct inode *, struct file *);
894-    int (*flush) (struct file *);
895-    int (*release) (struct inode *, struct file *);
896-    int (*fsync) (struct file *, struct dentry *, int datasync);
897-    int (*fasync) (int, struct file *, int);
898-    int (*lock) (struct file *, int, struct file_lock *);
899-    ssize_t (*readv) (struct file *, const struct iovec *, unsigned
long, loff_t *);
900-    ssize_t (*writev) (struct file *, const struct iovec *, unsigned
long, loff_t *);
901-    ssize_t (*sendpage) (struct file *, struct page *, int, size_t,
loff_t *, int);
902-    unsigned long (*get_unmapped_area)(struct file *, unsigned long,
unsigned long, unsigned long, unsigned long);
903-};
```

追加方法は Level6 で書きましたので省略します。
追加された証に、

```
[root@pw022 j03018]% cat /proc/devices
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 ttyS
  5 cua
  7 vcs
 10 misc
 14 sound
128 ptm
136 pts
162 raw
180 usb
226 drm
241 newnull2
242 chardev

Block devices:
```

```
  2 fd
  3 ide0
 22 ide1
```

となっています。

9.3 実行結果

作成したキャラクタデバイスを動かしてみます。

```
[root@pw022 ~]% echo ‘‘HELLO WORLD’’ > /dev/chardev
```

として、次に chardev を見ると、

```
[root@pw022 ~]% cat /dev/chardev
HELLO WORLD
```

となります。なお、この/dev/chardev は、同時に一つのユーザプロセスしか使えない様になっています。

最後に、Level6 と LevelX は、新しく構築した Linux-2.7.27 でのみ動きます。よって、実験が終了次第元のバージョンを元に戻しましたので、新しく追加したキャラクタデバイスは今の pw022 では使えません。

9.4 考察

今回のキャラクタデバイスは、同時アクセス可能プロセス数を 1 にしています。
これを複数にすると、同時にアクセスされると内部バッファの内容に不整合が生じると思われます。
そうならないためには、内部バッファ操作の前後でロックを取得/解放する必要があると思われます。