LECTURE 2 単純なキャラクタデバイスドライバ

著者:りょう (Ryo Ichinose)(rio_i@nifty.com)
Last update : 2006/06/09
since 2005/03/13

前のページへ 次のページへ

○ はじめに

Linux® (UNIX) でのデバイスの形式は、大きく以下に分けることが出来ます。

- キャラクタデバイス
- ブロックデバイス
- ネットワークインターフェース

キャラクタデバイスとはデータをバイト単位で扱うデバイスで、/dev/ttyS0 やコンソール等は代表的なキャラクタデバイスです。ブロックデバイスはデータをブロックと呼ばれる一定サイズの塊単位で扱うデバイスで、フロッピーディスクやハードディスクが代表的なブロックデバイスです。ネットワークインタフェースは、名前の通りネットワーク関連の処理を行うデバイスです。

これら以外にも SCSI 等いろいろなデバイスがありますが、それらは構造が異なるため上記の様にまとめることが出来ない様です。

これからしばらくは、比較的全体像が掴みやすいキャラクタデバイスを作っていきたいと思います。

- サンプルプログラム
- サンプルプログラムの説明

○ サンプルプログラム

open / close / read / write システムコールに対応したサンプルプログラム chardev.c です。はじめに のページから、全てのサンプルプログラムを 固めた tar ボールをダウンロードすることもできます。

```
gcc -DMODULE -D__KERNEL__ -O -c chardev.c
```

前回と違い、_KERNEL_ をコンパイルオプションとして指定しています。カーネルヘッダの中には、カーネルでしか参照しない様なデータ構造等を_KERNEL_ マクロで保護していますが、それらにアクセスする必要があるからです。これ以降のサンプルでは、必ず指定します。なお、ソース中に記述する際にはどのカーネルヘッダをインクルードするよりも先に定義して下さい。

また、最適化オプション -O はインライン関数を展開するために指定しています。これは、カーネル内の多くの関数がインライン関数として定義されており、最適化オプションを指定しないとそれらが展開されないため、コンパイルに失敗するからです。

chardev.c はユーザプロセスからの open / close / read / write に対応していますが、それぞれをユーザプロセスから実行するためにデバイスファイルを作成しなければなりません。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つけられます(デバイスファイル名称は重ならなければ何でも良い)。メジャー番号でカーネルがドライバを区別し、マイナー番号でドライバがデバイスを区別します。chardev.c ではマイナー番号を使用しない(同じドライバで複数のデバイスを扱うわけではない)ので、メジャー番号のみが大事です。chardev.c の 13 行目で使用するメジャー番号を 77 と定義しています。もし、各自のシステム上で使用済みの番号である場合は、使われていない別の番号に置き換えて下さい。なお、メジャー番号は動的に割り当てることもできますが、ここでは触れません。

ここではデバイスファイル名 /dev/chardev を使うことにします。マイナー番号は何でも良いので 0 にします。以下の様にしてデバイスファイルを作成して下さい (root で行って下さい)。

```
mknod /dev/chardev c 77 0
chmod 0666 /dev/chardev
```

insmod でモジュールを組み込んだ後は、

```
echo "HELLO WORLD" > /dev/chardev
```

の様にして書き込みを行い

```
cat /dev/chardev
HELLO WORLD
```

の様にして書き込んだ内容を読み込むことが出来ます。通算 32 バイト以上を書き込もうとすると、カーネルログに

```
<6>module [chardev.o] : no space left
```

の様に表示されます。なお、/dev/chardev は、同時に一つのユーザプロセスしか使用することができません。

--- chardev.c ---

```
  1  #include <asm/uaccess.h>    /* copy_from_user, copy_to_user */
  2  #include <linux/errno.h>
  3  #include <linux/fs.h>       /* inode, file, file_operations */
  4  #include <linux/kernel.h>   /* printk */
  5  #include <linux/module.h>
  6  #include <linux/spinlock.h>
  7
  8
  9  MODULE_AUTHOR( "Ryo" );
 10  MODULE_DESCRIPTION( "Sample character device driver." );
 11  MODULE_LICENSE( "GPL" );
 12
 13
 14  static int   devmajor = 77;
 15  static char* devname  = "chardev";
 16  static char* msg      = "module [chardev.o]";
 17
 18
 19  #define MAXBUF 32
 20  static unsigned char devbuf[ MAXBUF ];
 21  static int buf_pos;
 22
 23  static int access_num;
 24  static spinlock_t chardev_spin_lock;
 25
 26
 27  /*
 28   * open()
 29   * 1ユーザプロセスのみがデバイスファイルの open に成功する。
 30   */
 31  static int
 32  chardev_open( struct inode* inode, struct file* filp )
 33  {
 34      printk( KERN_INFO "%s : open()  called¥n", msg );
 35
 36      spin_lock( &chardev_spin_lock );
 37
 38      if ( access_num ) {
 39          spin_unlock( &chardev_spin_lock );
 40          return -EBUSY;
 41      }
 42
 43      access_num ++;
 44      spin_unlock( &chardev_spin_lock );
 45
 46      return 0;
 47  }
 48
 49
 50  /*
 51   * release()
 52   * 使用ユーザプロセス数をクリアする。
 53   */
 54  static int
 55  chardev_release( struct inode* inode, struct file* filp )
 56  {
 57      printk( KERN_INFO "%s : close() called¥n", msg );
 58
 59      spin_lock( &chardev_spin_lock );
 60      access_num --;
 61      spin_unlock( &chardev_spin_lock );
 62
 63      return 0;
 64  }
 65
 66
 67  /*
 68   * read()
 69   * ユーザプロセスに対して内部バッファの内容を転送する。
 70   * 転送した内容は内部バッファから消える。
 71   */
 72  static ssize_t
 73  chardev_read( struct file* filp, char* buf, size_t count, loff_t* pos )
 74  {
 75      int copy_len;
 76      int i;
 77
 78      printk( KERN_INFO "%s : read()  called¥n", msg );
 79
 80      if ( count > buf_pos )
 81          copy_len = buf_pos;
 82      else
```

```
 83          copy_len = count;
 84
 85      if ( copy_to_user( buf, devbuf, copy_len ) ) {
 86          printk( KERN_INFO "%s : copy_to_user failed¥n", msg );
 87          return -EFAULT;
 88      }
 89
 90      *pos += copy_len;
 91
 92      for ( i = copy_len; i < buf_pos; i ++ )
 93          devbuf[ i - copy_len ] = devbuf[i];
 94
 95      buf_pos -= copy_len;
 96      printk( KERN_INFO "%s : buf_pos = %d¥n", msg, buf_pos );
 97
 98      return copy_len;
 99  }
100
101
102  /*
103   * write()
104   * 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む。
105   */
106  static ssize_t
107  chardev_write(
108      struct file* filp, const char* buf, size_t count, loff_t* pos )
109  {
110      int copy_len;
111
112      printk( KERN_INFO "%s : write() called¥n", msg );
113
114      if ( buf_pos == MAXBUF ) {
115          printk( KERN_INFO "%s : no space left¥n", msg );
116          return -ENOSPC;
117      }
118
119      if ( count > ( MAXBUF - buf_pos ) )
120          copy_len = MAXBUF - buf_pos;
121      else
122          copy_len = count;
123
124      if ( copy_from_user( devbuf + buf_pos, buf, copy_len ) ) {
125          printk( KERN_INFO "%s : copy_from_user failed¥n", msg );
126          return -EFAULT;
127      }
128
129      *pos    += copy_len;
130      buf_pos += copy_len;
131
132      printk( KERN_INFO "%s : buf_pos = %d¥n", msg, buf_pos );
133      return copy_len;
134  }
135
136
137  static struct file_operations chardev_fops =
138  {
139      owner   : THIS_MODULE,
140      read    : chardev_read,
141      write   : chardev_write,
142      open    : chardev_open,
143      release : chardev_release,
144  };
145
146
147  /*
148   * モジュールの初期処理
149   * insmod 時に呼ばれる
150   */
151  int
152  init_module( void )
153  {
154      if ( register_chrdev( devmajor, devname, &chardev_fops ) ) {
155          printk( KERN_INFO "%s : register_chrdev failed¥n" );
156          return -EBUSY;
157      }
158
159      spin_lock_init( &chardev_spin_lock );
160      printk( KERN_INFO "%s : loaded  into kernel¥n", msg );
161
162      return 0;
163  }
164
165
166  /*
167   * モジュールの解放処理
168   * モジュールの参照数が 0 であれば、rmmod 時に呼ばれる
169   */
170  void
171  cleanup_module( void )
172  {
173      if ( unregister_chrdev( devmajor, devname ) ) {
174          printk( KERN_INFO "%s : unregister_chrdev failed¥n" );
175          /* 対策無し ... */
```

```
176         }
177
178         printk( KERN_INFO "%s : removed from kernel¥n", msg );
179     }
180
181
182     /* End of chardev.c */
```

○ サンプルプログラムの説明

では、サンプルプログラムについて説明していきます。

- 1 ～ 6 行目

  必要なヘッダファイルをインクルードしています。これらの中には、_KERNEL_ マクロで内容を保護している箇所がありますので、先頭に _KERNEL_ と記述するかコンパイルスイッチで定義する必要があります。

- 9 ～ 11 行目

  MODULE_AUTHOR と MODULE_DESCRIPTION は、それぞれモジュール作成者 およびモジュールについての説明をカーネルモジュール内に埋め込むための マクロです。動作上必須というわけではありません。MODULE_LICENSE マクロについては、前回の説明通りです。

- 14 行目

  chardev.c で使用するメジャー番号を 77 番としています。他の番号を 使いたい時には、ここを書き換えた上で同じ番号を使って mknod でデバイス ファイルを作成して下さい。

- 19 ～ 21 行目

  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。

- 23 ～ 24 行目

  同時アクセス可能プロセス数を管理するための変数です。chardev.c では、同時に 1 ユーザプロセスからのアクセスしか許可しません。open が実行された際に access_num が 0 でなければ、それ以降の操作を行えない様にしています。また、SMP システム用に access_num の操作を排他制御するためのロック 変数 chardev_spin_lock を用意しています。スピンロックについては後述します。

- 27 ～ 47 行目 (chardev_open)

  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ます。ここでは、access_num が 0 でない(既に使用されている)場合には EBUSY を返して終了しています。

  access_num の操作を行う前後に、spin_lock / spin_unlock という処理を実行しています。SMP システムの場合は、access_num の 0 チェックと実際の増分処理の間に別 CPU が割り込む可能性があります。それでは問題があるため、access_num のチェックおよび増分処理の前に 何らかのロックを取得する必要があります。ロックにもいろいろありますが、ごく短時間の間だけロックできれば良い様な場合にはスピンロックを使います。

```
#include <linux/spinlock_h>
spin_lock_init( spinlock_t* lock );
spin_lock( spinlock_t* lock );
spin_unlock( spinlock_t* lock );
```

  スピンロックは、ロックを取得するまでビジーループで待ちます。なお、_SMP_ を定義せずにコンパイルした場合、スピンロックは何も処理しません。また、スピンロックを使う際には あらかじめ spin_lock_init で初期化を行う必要があります。chardev.c では init_module 内で行っています。

- 50 ～ 64 行目 (chardev_release)

  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。linux/fs.h 内の struct file_operations のメンバである release 関数ポインタの形式で定義する必要があります。返却値については chardev_open と同じです。ここでは access_num をデクリメントしているだけなので、常に 0 を返します。

  access_num 操作の前後でスピンロックの処理をしていますが、chardev.c の場合は実際にはスピンロックを取得する必要はありません。

- 67 ～ 99 行目 (chardev_read)

  ユーザプロセスからの read に対応する関数です。linux/fs.h 内の struct file_operations のメンバである read 関数ポインタの形式で定義する 必要があります。エラー終了時の返却値に対しては chardev_open と同じです。正常終了時は、ユーザプロセスに返すデータのバイト数を返します。

  引数の buf が結果を格納するためのユーザ空間のバッファで、count が要求された読みだしバイト数です。ここでは、count が現在の格納バイト数 より大きい場合は内部バッファサイズの、それ以外は count バイトのデータを ユーザプロセスに返しています。カーネル空間とユーザプロセス空間では アドレス体系が異なるため、単に buf にコピーするわけにはいきません。この場合、copy_to_user という関数を使います。

```
#include <asm/uaccess.h>
unsigned long copy_to_user( void* to, const void* from, unsigned long count );
```

to にコピー先アドレス、from にコピー元アドレスを指定します。また、count にコピーするバイト数を指定します。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 が返ります。

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 先頭に詰めています。なお、chardev.c は read 実行時に返すデータが無い場合でも ブロックすることはありません。ブロック I/O については今後触れる予定 です。

なお、最初の引数である struct file* filp は、オープンされている ファイル情報が格納されています。例えば、ファイルの読み書きモードや現在のポインタ位置 といった内容が格納されています。また、最後の引数である loff_t* pos は、現在アクセスしているファイルの位置です。コピー直後に、コピー長だけ進めています。位置が問題となるデータを扱うドライバの場合は、この値が示す場所からのデータを 返す必要があります。

- 102 ～ 134 行目 (chardev_write)

ユーザプロセスからの write に対応する関数です。linux/fs.h 内の struct file_operations のメンバである write 関数ポインタの形式で定義する 必要があります。エラー終了時の返却値に対しては chardev_open と同じです。正常終了時は、ユーザプロセスから書き込まれたデータのバイト数を返します。

引数の buf が内部バッファに格納するためのデータを表すユーザ空間のバッファで、count が要求された書き込みバイト数です。ここでは、内部バッファの 空きバイト数だけ buf からデータを読み込み、内部バッファに書き込んでいます。内部バッファに空きが無い場合は、ENOSPC を返しています。

ここでも単に buf からコピーするわけにはいきません。この場合、copy_from_user という関数を使います。

```
#include <asm/uaccess.h>
unsigned long copy_from_user( void* to, const void* from, unsigned long count );
```

to にコピー先アドレス、from にコピー元アドレスを指定します。また、count にコピーするバイト数を指定します。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 が返ります。

filp、pos については chardev_read と同様です。

- 137 ～ 144 行目

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付けるためには、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 2.4.19 の linux/fs.h では、以下の様に定義されています。

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*readv) (
        struct file *, const struct iovec *, unsigned long, loff_t *);
    ssize_t (*writev) (
        struct file *, const struct iovec *, unsigned long, loff_t *);
    ssize_t (*sendpage) (
        struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *,
        unsigned long, unsigned long, unsigned long, unsigned long);
};
```

この中で、関連付けを行いたいメンバに関数ポインタを設定し、他のメンバには NULL を設定します。ただし上記のメンバの中で struct module * owner メンバは、ファイル操作との関連付けを行うための関数ポインタではなく、カーネルモジュール自身の情報を格納するためのメンバです。そのため、別途初期化する必要があります。カーネル 2.4 では、THIS_MODULE という値で初期化することが出来ます。

chardev.c では、全メンバに値を設定するのではなく、gcc の拡張機能 である構造体のタグ付き初期化を使用しています。"メンバ名 :" というラ ベルに続けて値を記述していきます。この様にすると、必要な箇所だけ設 定することが出来ます。

- 147 ～ 163 行目 (init_module)

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。

```
#include <linux/fs.h>
int register_chrdev(
    unsigned int major, const char* name, struct file_operations* fops );
```

major はメジャー番号、name は任意のドライバ名称、fops はファイル操作と関数の関連テーブルのアドレスです。返り値が負の場合はエラー終了です。major に 0 以外を指定して正常に終了 した場合は 0 が返ります。また major に 0 を指定した場合は メジャー番号の動的割り当てが行われ、実際に割り当てられたメジャー番号が返ります。

また、spin_lock_init によってスピンロックの初期化を行っています。

- 166 ～ 179 行目 (cleanup_module)

前のページへ 次のページへ

○ はじめに

Linux® (UNIX) でのデバイスの形式は、大きく以下に分けることが出来ます。

- キャラクタデバイス
- ブロックデバイス
- ネットワークインターフェース

キャラクタデバイスとはデータをバイト単位で扱うデバイスで、/dev/ttyS0 やコンソール等は代表的なキャラクタデバイスです。ブロックデバイスはデータをブロックと呼ばれる一定サイズの塊単位で扱うデバイスで、フロッピーディスクやハードディスクが代表的なブロックデバイスです。ネットワークインタフェースは、名前の通りネットワーク関連の処理を行うデバイスです。

これら以外にも SCSI 等いろいろなデバイスがありますが、それらは構造が異なるため上記の様にまとめることが出来ない様です。

これからしばらくは、比較的全体像が掴みやすいキャラクタデバイスを作っていきたいと思います。

- サンプルプログラム
- サンプルプログラムの説明

○ サンプルプログラム

open / close / read / write システムコールに対応したサンプルプログラム chardev.c です。はじめに のページから、全てのサンプルプログラムを 固めた tar ボールをダウンロードすることもできます。

```
gcc -DMODULE -D__KERNEL__ -O -c chardev.c
```

前回と違い、_KERNEL_ をコンパイルオプションとして指定しています。カーネルヘッダの中には、カーネルでしか参照しない様なデータ構造等を_KERNEL_ マクロで保護していますが、それらにアクセスする必要があるからです。これ以降のサンプルでは、必ず指定します。なお、ソース中に記述する際にはどのカーネルヘッダをインクルードするよりも先に定義して下さい。

また、最適化オプション -O はインライン関数を展開するために指定しています。これは、カーネル内の多くの関数がインライン関数として定義されており、最適化オプションを指定しないとそれらが展開されないため、コンパイルに失敗するからです。

chardev.c はユーザプロセスからの open / close / read / write に対応していますが、それぞれをユーザプロセスから実行するためにデバイスファイルを作成しなければなりません。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つけられます(デバイスファイル名称は重ならなければ何でも良い)。メジャー番号でカーネルがドライバを区別し、マイナー番号でドライバがデバイスを区別します。chardev.c ではマイナー番号を使用しない(同じドライバで複数のデバイスを扱うわけではない)ので、メジャー番号のみが大事です。chardev.c の 13 行目で使用するメジャー番号を 77 と定義しています。もし、各自のシステム上で使用済みの番号である場合は、使われていない別の番号に置き換えて下さい。なお、メジャー番号は動的に割り当てることもできますが、ここでは触れません。

ここではデバイスファイル名 /dev/chardev を使うことにします。マイナー番号は何でも良いので 0 にします。以下の様にしてデバイスファイルを作成して下さい (root で行って下さい)。

```
mknod /dev/chardev c 77 0
chmod 0666 /dev/chardev
```

insmod でモジュールを組み込んだ後は、

```
echo "HELLO WORLD" > /dev/chardev
```

の様にして書き込みを行い

```
cat /dev/chardev
HELLO WORLD
```

の様にして書き込んだ内容を読み込むことが出来ます。通算 32 バイト以上を書き込もうとすると、カーネルログに

```
<6>module [chardev.o] : no space left
```

の様に表示されます。なお、/dev/chardev は、同時に一つのユーザプロセスしか使用することができません。

--- chardev.c ---

```
1    #include <asm/uaccess.h>    /* copy_from_user, copy_to_user */
2    #include <linux/errno.h>
3    #include <linux/fs.h>       /* inode, file, file_operations */
4    #include <linux/kernel.h>   /* printk */
```

```
 5  #include <linux/module.h>
 6  #include <linux/spinlock.h>
 7
 8
 9  MODULE_AUTHOR( "Ryo" );
10  MODULE_DESCRIPTION( "Sample character device driver." );
11  MODULE_LICENSE( "GPL" );
12
13
14  static int   devmajor = 77;
15  static char* devname  = "chardev";
16  static char* msg      = "module [chardev.o]";
17
18
19  #define MAXBUF 32
20  static unsigned char devbuf[ MAXBUF ];
21  static int buf_pos;
22
23  static int access_num;
24  static spinlock_t chardev_spin_lock;
25
26
27  /*
28   * open()
29   * 1ユーザプロセスのみがデバイスファイルの open に成功する。
30   */
31  static int
32  chardev_open( struct inode* inode, struct file* filp )
33  {
34      printk( KERN_INFO "%s : open()  called¥n", msg );
35
36      spin_lock( &chardev_spin_lock );
37
38      if ( access_num ) {
39          spin_unlock( &chardev_spin_lock );
40          return -EBUSY;
41      }
42
43      access_num ++;
44      spin_unlock( &chardev_spin_lock );
45
46      return 0;
47  }
48
49
50  /*
51   * release()
52   * 使用ユーザプロセス数をクリアする。
53   */
54  static int
55  chardev_release( struct inode* inode, struct file* filp )
56  {
57      printk( KERN_INFO "%s : close() called¥n", msg );
58
59      spin_lock( &chardev_spin_lock );
60      access_num --;
61      spin_unlock( &chardev_spin_lock );
62
63      return 0;
64  }
65
66
67  /*
68   * read()
69   * ユーザプロセスに対して内部バッファの内容を転送する。
70   * 転送した内容は内部バッファから消える。
71   */
72  static ssize_t
73  chardev_read( struct file* filp, char* buf, size_t count, loff_t* pos )
74  {
75      int copy_len;
76      int i;
77
78      printk( KERN_INFO "%s : read()  called¥n", msg );
79
80      if ( count > buf_pos )
81          copy_len = buf_pos;
82      else
83          copy_len = count;
84
85      if ( copy_to_user( buf, devbuf, copy_len ) ) {
86          printk( KERN_INFO "%s : copy_to_user failed¥n", msg );
87          return -EFAULT;
88      }
89
90      *pos += copy_len;
91
92      for ( i = copy_len; i < buf_pos; i ++ )
93          devbuf[ i - copy_len ] = devbuf[i];
94
95      buf_pos -= copy_len;
96      printk( KERN_INFO "%s : buf_pos = %d¥n", msg, buf_pos );
97
```

```
 98      return copy_len;
 99  }
100
101
102  /*
103   * write()
104   * 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む。
105   */
106  static ssize_t
107  chardev_write(
108      struct file* filp, const char* buf, size_t count, loff_t* pos )
109  {
110      int copy_len;
111
112      printk( KERN_INFO "%s : write() called¥n", msg );
113
114      if ( buf_pos == MAXBUF ) {
115          printk( KERN_INFO "%s : no space left¥n", msg );
116          return -ENOSPC;
117      }
118
119      if ( count > ( MAXBUF - buf_pos ) )
120          copy_len = MAXBUF - buf_pos;
121      else
122          copy_len = count;
123
124      if ( copy_from_user( devbuf + buf_pos, buf, copy_len ) ) {
125          printk( KERN_INFO "%s : copy_from_user failed¥n", msg );
126          return -EFAULT;
127      }
128
129      *pos    += copy_len;
130      buf_pos += copy_len;
131
132      printk( KERN_INFO "%s : buf_pos = %d¥n", msg, buf_pos );
133      return copy_len;
134  }
135
136
137  static struct file_operations chardev_fops =
138  {
139      owner   : THIS_MODULE,
140      read    : chardev_read,
141      write   : chardev_write,
142      open    : chardev_open,
143      release : chardev_release,
144  };
145
146
147  /*
148   * モジュールの初期処理
149   * insmod 時に呼ばれる
150   */
151  int
152  init_module( void )
153  {
154      if ( register_chrdev( devmajor, devname, &chardev_fops ) ) {
155          printk( KERN_INFO "%s : register_chrdev failed¥n" );
156          return -EBUSY;
157      }
158
159      spin_lock_init( &chardev_spin_lock );
160      printk( KERN_INFO "%s : loaded  into kernel¥n", msg );
161
162      return 0;
163  }
164
165
166  /*
167   * モジュールの解放処理
168   * モジュールの参照数が 0 であれば、rmmod 時に呼ばれる
169   */
170  void
171  cleanup_module( void )
172  {
173      if ( unregister_chrdev( devmajor, devname ) ) {
174          printk( KERN_INFO "%s : unregister_chrdev failed¥n" );
175          /* 対策無し ... */
176      }
177
178      printk( KERN_INFO "%s : removed from kernel¥n", msg );
179  }
180
181
182  /* End of chardev.c */
```

○ サンプルプログラムの説明

では、サンプルプログラムについて説明していきます。

- 1 ～ 6 行目

  必要なヘッダファイルをインクルードしています。これらの中には、_KERNEL_ マクロで内容を保護している箇所がありますので、先頭に _KERNEL_ と記述するかコンパイルスイッチで定義する必要があります。

- 9 ～ 11 行目

  MODULE_AUTHOR と MODULE_DESCRIPTION は、それぞれモジュール作成者 およびモジュールについての説明をカーネルモジュール内に埋め込むための マクロです。動作上必須というわけではありません。MODULE_LICENSE マクロについては、前回の説明通りです。

- 14 行目

  chardev.c で使用するメジャー番号を 77 番としています。他の番号を 使いたい時には、ここを書き換えた上で同じ番号を使って mknod でデバイス ファイルを作成して下さい。

- 19 ～ 21 行目

  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。

- 23 ～ 24 行目

  同時アクセス可能プロセス数を管理するための変数です。chardev.c では、同時に 1 ユーザプロセスからのアクセスしか許可しません。open が実行された際に access_num が 0 でなければ、それ以降の操作を行えない様にしています。また、SMP システム用に access_num の操作を排他制御するためのロック 変数 chardev_spin_lock を用意しています。スピンロックについては後述します。

- 27 ～ 47 行目 (chardev_open)

  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ます。ここでは、access_num が 0 でない(既に使用されている)場合には EBUSY を返して終了しています。

  access_num の操作を行う前後に、spin_lock / spin_unlock という処理を実行しています。SMP システムの場合は、access_num の 0 チェックと実際の増分処理の間に別 CPU が割り込む可能性があります。それでは問題があるため、access_num のチェックおよび増分処理の前に 何らかのロックを取得する必要があります。ロックにもいろいろありますが、ごく短時間の間だけロックできれば良い様な場合にはスピンロックを使います。

```
#include <linux/spinlock.h>
spin_lock_init( spinlock_t* lock );
spin_lock( spinlock_t* lock );
spin_unlock( spinlock_t* lock );
```

  スピンロックは、ロックを取得するまでビジーループで待ちます。なお、_SMP_ を定義せずにコンパイルした場合、スピンロックは何も処理しません。また、スピンロックを使う際には あらかじめ spin_lock_init で初期化を行う必要があります。chardev.c では init_module 内で行っています。

- 50 ～ 64 行目 (chardev_release)

  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。linux/fs.h 内の struct file_operations のメンバである release 関数ポインタの形式で定義する必要があります。返却値については chardev_open と同じです。ここでは access_num をデクリメントしているだけなので、常に 0 を返します。

  access_num 操作の前後でスピンロックの処理をしていますが、chardev.c の場合は実際にはスピンロックを取得する必要はありません。

- 67 ～ 99 行目 (chardev_read)

  ユーザプロセスからの read に対応する関数です。linux/fs.h 内の struct file_operations のメンバである read 関数ポインタの形式で定義する 必要があります。エラー終了時の返却値に対しては chardev_open と同じです。 正常終了時は、ユーザプロセスに返すデータのバイト数を返します。

  引数の buf が結果を格納するためのユーザ空間のバッファで、count が要求された読みだしバイト数です。ここでは、count が現在の格納バイト数 より大きい場合は内部バッファサイズの、それ以外は count バイトのデータを ユーザプロセスに返しています。カーネル空間とユーザプロセス空間では アドレス体系が異なるため、単に buf にコピーするわけにはいきません。この場合、copy_to_user という関数を使います。

```
#include <asm/uaccess.h>
unsigned long copy_to_user( void* to, const void* from, unsigned long count );
```

  to にコピー先アドレス、from にコピー元アドレスを指定します。また、count にコピーするバイト数を指定します。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 が返ります。

  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 先頭に詰めています。なお、chardev.c は read 実行時に返すデータが無い場合でも ブロックすることはありません。ブロック I/O については今後触れる予定 です。

  なお、最初の引数である struct file* filp は、オープンされている ファイル情報が格納されています。例えば、ファイルの読み書きモードや現在のポインタ位置 といった内容が格納されています。また、最後の引数である loff_t* pos は、現在アクセスしているファイルの位置です。コピー直後に、コピー長だけ進めています。 位置が問題となるデータを扱うドライバの場合は、この値が示す場所からのデータを 返す必要があります。

- 102 ～ 134 行目 (chardev_write)

ユーザプロセスからの write に対応する関数です。linux/fs.h 内の struct file_operations のメンバである write 関数ポインタの形式で定義する 必要があります。エラー終了時の返却値に対しては chardev_open と同じです。正常終了時は、ユーザプロセスから書き込まれたデータのバイト数を返します。

引数の buf が内部バッファに格納するためのデータを表すユーザ空間のバッファで、count が要求された書き込みバイト数です。ここでは、内部バッファの 空きバイト数だけ buf からデータを読み込み、内部バッファに書き込んでいます。内部バッファに空きが無い場合は、ENOSPC を返しています。

ここでも単に buf からコピーするわけにはいきません。この場合、copy_from_user という関数を使います。

```
#include <asm/uaccess.h>
unsigned long copy_from_user( void* to, const void* from, unsigned long count );
```

to にコピー先アドレス、from にコピー元アドレスを指定します。また、count にコピーするバイト数を指定します。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 が返ります。

filp、pos については chardev_read と同様です。

- 137 ～ 144 行目

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付けるためには、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 2.4.19 の linux/fs.h では、以下の様に定義されています。

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*readv) (
        struct file *, const struct iovec *, unsigned long, loff_t *);
    ssize_t (*writev) (
        struct file *, const struct iovec *, unsigned long, loff_t *);
    ssize_t (*sendpage) (
        struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *,
        unsigned long, unsigned long, unsigned long, unsigned long);
};
```

この中で、関連付けを行いたいメンバに関数ポインタを設定し、他のメンバには NULL を設定します。ただし上記のメンバの中で struct module * owner メンバは、ファイル操作との関連付けを行うための関数ポインタではなく、カーネルモジュール自身の情報を格納するためのメンバです。そのため、別途初期化する必要があります。カーネル 2.4 では、THIS_MODULE という値で初期化することが出来ます。

chardev.c では、全メンバに値を設定するのではなく、gcc の拡張機能 である構造体のタグ付き初期化を使用しています。"メンバ名 :" というラ ベルに続けて値を記述していきます。この様にすると、必要な箇所だけ設 定することが出来ます。

- 147 ～ 163 行目 (init_module)

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。

```
#include <linux/fs.h>
int register_chrdev(
    unsigned int major, const char* name, struct file_operations* fops );
```

major はメジャー番号、name は任意のドライバ名称、fops はファイル操作と関数の関連テーブルのアドレスです。返り値が負の場合はエラー終了です。major に 0 以外を指定して正常に終了 した場合は 0 が返ります。また major に 0 を指定した場合は メジャー番号の動的割り当てが行われ、実際に割り当てられたメジャー番号が返ります。

また、spin_lock_init によってスピンロックの初期化を行っています。

- 166 ～ 179 行目 (cleanup_module)

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。

```
#include <linux/fs.h>
int unregister_chrdev( unsigned int major, const char* name );
```

major は登録時に指定したメジャー番号、name は登録時に指定した デバイス名称です。正常終了時は 0、それ以外はエラー終了です。

以上で chardev.c の解説は終わりです。open / close / read / write という基本的なファイル操作に対応してみましたが、それだけでも立派なデバイスドライバに見えてしまいます！(操作しているハードウェアはメモリだけですが。。。)

今後は、少しづつ各操作を拡張していきたいと思います。

前のページへ 次のページへ

2006/06/09 updated.
2005/03/13 created.