Linux カーネルへのシステムコール追加の手順書

2020/6/24

中川 雄介

1 概要

本手順書では，Linux カーネルへシステムコールを追加する方法を記す．カーネルへシステムコールを追加するには，Linux のソースコードを取得してシステムコールのソースコードを追加した後，カーネルの再構築を行う．また，本手順書で想定する読者はコンソールの基本的な操作を習得しているものとする．

2 実装環境

システムコールを追加した環境を表 1 に示す．導入済みパッケージのうち git は linux の取得に，gcc， make， bc はカーネルの再構築に用いる．

表 1 実装環境

項目	環境
OS	Debian GNU/Linux 10 (buster)
カーネル	Linux 4.19.0+
CPU	Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
メモリ	16GB

3 追加したシステムコールの概要

今回は，カーネルのメッセージバッファに文字列を出力するシステムコールを追加した．

システムコールの関数名は my_syscall とした．システムコールの概要を以下に示す．

【形式】 asmlinkage long my_syscall(char __user *buf, int count)

【引数】 char *buf :出力したい文字列，int count :出力したい文字列の文字数

【戻り値】 long err :エラー処理用

【機能】カーネルのメッセージバッファに文字列を出力する．

4 システムコール追加の手順

4.1 概要

システムコール追加手順の構成を以下に示す.

(1) Linux カーネルの取得
  （A）linux のソースコードの取得
  （B）ブランチの作成と切り替え
(2) ソースコードの編集
  （A）システムコールの作成
  （B）システムコール番号の設定
  （C）システムコールのプロトタイプ宣言
(3) カーネルの再構築
  （A）.config ファイルの作成
  （B）カーネルのコンパイル
  （C）カーネルのインストール
  （D）カーネルモジュールのコンパイル
  （E）カーネルモジュールのインストール
  （F）初期 RAM ディスクの作成
  （G）ブートローダーの設定
  （H）再起動

以降では上記の手順について述べる．(1) については 4.2 節，(2) については 4.3 節，(3) については 4.4 節で述べる.

4.2 Linux カーネルの取得

(1) Linux のソースコードの取得
  Linux のソースコードを取得する．Linux のソースコードは Git で管理されている．Git とはオープンソースの分散型バージョン管理システムである．下記の Git リポジトリからクローンすることで，Linux のソースコードを取得する．リポジトリとはファイルやディレクトリの状態を記録する場所のことであり，クローンとはリポジトリの内容を指定のディレクトリに複製すことである.

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git
```

  本手順書では，/home/nakagawa/git 以下のディレクトリでソースコードを管理す

る．/home/nakagawa で以下のコマンドを実行する．

```
$ mkdir git
$ cd git
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-st
  able.git
```

実行すると，mkdir コマンドにより git ディレクトリが作成され，cd コマンドにより git ディレクトリに移動する．git clone コマンドにより/home/nakagawa/git 以下に linux-stable ディレクトリが作成される．この linux-stable ディレクトリ以下に Linux のソースコードが格納されている．

(2) ブランチの作成と切り替え
Linux カーネルのソースコードのバージョンを切り替えるため，ブランチの作成と切り替えを行う．ブランチとは開発の履歴を管理するため分岐である．/home/nakagawa/git/linux-stable で以下のコマンドを実行する．

```
$ git checkout -b 4.19 v4.19
```

実行後，v4.19 というタグが示すコミットからブランチ 4.19 が作成され，ブランチ 4.19 に切り替わる．コミットとはある時点における開発の状態を記録したものである．タグとはコミットを識別するためにつける印である．

4.3 ソースコードの編集

ソースコードの左端に書かれている数字は行数を表し，追加した行には + をつけている．

(1) システムコールの作成
/home/nakagawa/git/linux-stable/kernel/sys.c に 3 章で示したシステムコールを追加する．ソースコードは 6.1 節に添付してある．

(2) システムコール番号の設定
/home/nakagawa/git/linux-stable/arch/x86/entry/syscalls/syscall_64.tbl を編集して，追加するシステムコール番号を定義する．追加する際には既存のシステムコール番号と重複しないようにする．今回は作成したシステムコールのシステムコール番号を 400 とした．このシステムコール番号は，システムコールを呼び出す際に必要になる．編集例を以下に示す．

```
    346     334     common rseq             __x64_sys_rseq
+   347     400     common my_syscall       __x64_sys_my_syscall
```

(3) システムコールのプロトタイプ宣言
/home/nakagawa/git/linux-stable/include/linux/syscalls.h を編集して，追加するシステム

コールのプロトタイプ宣言を記述する．編集例を以下に示す．

```
+   1296    asmlinkage long sys_my_syscall(char __user *buf, int count);
```

4.4 カーネルの再構築

(1) .config ファイルの生成
.config ファイルを作成する．.config ファイルとはカーネルの設定を記述したコンフィギュレーションファイルである．以下のコマンドを実行し，コンフィギュレーションファイルを作成する．エラーが発生した場合，エラー内容に応じて必要なパッケージをインストールする．

```
$ make defconfig
```

(2) カーネルのコンパイル
Linux カーネルをコンパイルする．以下のコマンドを実行する．

```
$ sudo make bzImage -j8
```

上記コマンドの「-j」オプションは，同時に実行できるジョブ数を指定の個数に指定する．ジョブ数が多すぎるとメモリ不足により実行速度が低下する場合があるため，適切なジョブ数を指定する必要がある．ジョブ数は CPU のコア数の 2 倍の数が効果的であるため，ジョブ数は 4 コアの 2 倍の 8 とした．コマンド実行後，/home/nakagawa/git/linux-stable/arch/x86/boot 以下に bzImage という名前の圧縮カーネルイメージが作成される．カーネルイメージとは実行可能形式のカーネルを含むファイルである．同時に，/home/nakagawa/git/linux-stable 以下に全てのカーネルシンボルのアドレスを記述した System.map が作成される．カーネルシンボルとはカーネルのプログラムが格納されたメモリアドレスと対応付けられた文字列のことである．

(3) カーネルのインストール
コンパイルしたカーネルをインストールする．/home/nakagawa/git/linux-stable で以下のコマンドを実行する．

```
$ sudo cp arch/x86/boot/bzImage /boot/vmlinuz-4.19.0-linux
$ sudo cp System.map /boot/System.map-4.19.0-linux
```

実行後，bzImage と System.map が，/boot 以下にそれぞれ vmlinuz-4.19.0-linux と System.map-4.19.0-linux という名前でコピーされる．

(4) カーネルモジュールのコンパイル
カーネルモジュールをコンパイルする．カーネルモジュールとは機能を拡張するためのバイナリファイルである．以下のコマンドを実行する．

```
$ make modules
```

(5) カーネルモジュールのインストール
コンパイルしたカーネルモジュールをインストールする．以下のコマンドを実行する．

```
$ sudo make modules_install
```

実行結果の最後の行は以下のように表示される．

```
DEPMOD 4.19.0
```

上記の例では，/lib/modules/4.19.0 ディレクトリにカーネルモジュールがインストールされている．このディレクトリ名は後の手順 で必要となるため，控えておく．

(6) 初期 RAM ディスクの作成
初期 RAM ディスクを作成する．初期 RAM ディスクとは初期ルートファイルシステムのことである．これは実際のルートファイルシステムが使用できるようになる前にマウントされる．以下のコマンドを実行する．

```
$ sudo update-initramfs -c -k 4.19.0
```

先ほど控えておいたディレクトリ名をコマンドの引数として与える．実行後，/boot 以下に初期 RAM ディスク initrd.img-4.19.0 が作成される．

(7) ブートローダーの設定
システムコールを実装したカーネルをブートローダから起動可能にするために，ブートローダを設定する．ブートローダの設定ファイルは/boot/grub/grub.cfg である．エントリを追加するためにはこのファイルを編集する必要がある．Debian10.3 で使用されているブートローダは GRUB2 である．GRUB2 でカーネルのエントリを追加する際，設定ファイルを直接編集しない．/etc/grub.d 以下にエントリ追加用のスクリプトを作成し，そのスクリプトを実行することでエントリを追加する．ブートローダを設定する手順を以下に示す．

（A）エントリ追加用のスクリプトの作成
カーネルのエントリを追加するため，エントリ追加用のスクリプトを作成する．本手順書では，既存のファイル名に倣い作成するスクリプトのファイル名は 11_linux-4.19.0 とする．スクリプトの記述例を以下に示す．

```
1 #!/bin/sh -e
2 echo "Adding my custom Linux to GRUB2"
3 cat << EOF
4 menuentry "My custom Linux" {
5 set root=(hd0,1)
6 linux /vmlinuz-4.19.0-linux ro root=/dev/sda3 quiet
7 initrd /initrd.img-4.19.0
8 }
9 EOF
```

スクリプトに記述された各項目について以下に示す.

(a) menuentry <表示名>
<表示名>: カーネル選択画面に表示される名前

(b) set root=(<HDD 番号>，<パーティション番号>)
<HDD 番号>: カーネルが保存されている HDD の番号
<パーティション番号>: HDD の/boot が割り当てられたパーティション番号

(c) linux <カーネルイメージのファイル名>
<カーネルイメージのファイル名>: 起動するカーネルのカーネルイメージ

(d) ro <root デバイス>
<root デバイス>: 起動時に読み込み専用でマウントするデバイス

(e) root=<ルートファイルシステム><その他のブートオプション>
<ルートファイルシステム>: /root を割り当てたパーティション
<その他のブートオプション>: quiet はカーネルの起動時に出力するメッセージを省略する.

(f) initrd <初期 RAM ディスク名>
<初期 RAM ディスク名>: 起動時にマウントする初期 RAM ディスク名

(8) 実行権限の付与
/etc/grub.d で以下のコマンドを実行し，作成したスクリプトに実行権限を付与する.

```
$ sudo chmod +x 11_linux-4.19.0
```

(9) エントリ追加用のスクリプトの実行
以下のコマンドを実行し，作成したスクリプトを実行する.

```
$ sudo update-grub
```

エントリ追加用のスクリプトの実行後，/boot/grub/grub.cfg にシステムコールを実装したカーネルのエントリが追加される

(10) 再起動
任意のディレクトリで以下のコマンドを実行し，計算機を再起動させる.

```
$ sudo reboot
```

再起動後，GRUB2 のカーネル選択画面にエントリが追加されている．手順 7A のスクリプトを用いた場合，カーネル選択画面で My custom Linux を選択し，起動する.

5 テスト

5.1 概要

本手順書で追加したシステムコールが実装されているか確認する．まず，追加したシステムコールを実装したテストプログラムを作成する．テストプログラムを実行して，追加したシステムコールが正しく動作しているかを確認する．

5.2 テストプログラムの作成

システムコールを実行するテストプログラムを作成する．本手順書ではテストプログラムの名前を `call_my_syscall.c` とし/home/nakagawa/git/linux-stable 以下に作成する．テストプログラムの処理の流れは以下の通りである．

(1) 引数で文字列を受け取る
(2) `call_my_syscall.c` のシステムコール番号で syscall() を呼び出す
(3) エラーがなければ受け取った文字列を表示する

このプログラムに引数として文字列を入力して実行すると，追加したシステムコールを使用して，指定した文字列がカーネルのメッセージバッファに出力され，問題がなければ入力した文字列が表示される．作成したテストプログラムのソースコードを 6.2 節に添付してある．

5.3 テストプログラムの実行

5.2 節で作成したプログラムをコンパイルし実行する．以下のコマンドを実行する．

```
$ gcc call_my_syscall.c
$ ./a.out test
```

実行後，システムコールが追加できていれば以下のような結果が得られる．

```
$ print buf = test
```

5.4 システムコールの実行結果の確認

カーネルのメッセージバッファを確認する．以下のコマンドを実行する．

```
$ sudo dmesg | tail
```

実行後，システムコールが追加できていれば以下のような結果が得られる．なお，角括弧内の数字はカーネル起動開始からの経過時間を表す．

```
$ [ 533.131041] <MY_SYSCALL>test
```

6 作成したプログラムのソースコード

6.1 システムコールのソースコード

/home/nakagawa/git/linux-stable/kernel/sys.c に追加したソースコードを抜粋して以下に示す.

```
147 SYSCALL_DEFINE2(my_syscall, char __user *, buf, int, count){
148         long err;
149         char text[128]= {0};
150
151         if(count > sizeof(text)){
152                 return (-ENOMEM);
153         }
154
155         err = copy_from_user(text, buf, strlen(buf));
156
157         printk("<MY_SYSCALL>%s", text);
158         printk("");
159
160         return(err);
161 }
```

6.2 テストプログラムのソースコード

```
 1 #include <stdio.h>
 2 #include <unistd.h>
 3 #include <string.h>
 4
 5 #define SYS_my_syscall 400
 6
 7 int main(int argc, char *argv[]){
 8
 9         char buf[128] ="default_message";
10         long ret;
11
```

```
12          if(argc != 1){
13                  strncpy(buf, argv[1], strlen(argv[1]));
14                  buf[strlen(argv[1])] = '\0';
15          }
16
17          ret = syscall(SYS_my_syscall, buf, strlen(buf));
18
19          if(ret<0){
20                  fprintf(stderr, "err:%ld\n", ret);
21          }else{
22                  printf("print buf = %s\n",buf);
23          }
24
25          return (0);
26  }
```