debugfsを利用した実際のデバッグ

Yoichi Yuasa

OSAKA NDS Embedded Linux Cross Forum #11

自己紹介

- 湯浅陽一
- 1999年よりLinux kernel開発に参加
- MIPSアーキテクチャのいくつかのCPUへLinux kernelを移植

本日の内容

- デバイスドライバデバッグ手法概要
- debugfsを利用したデバッグ実装
- debugfsを利用するポイント

デバイスドライバデバッグ手法

- printk
- ftrace
- Kprobes
- debugfs

printk

- 利用が簡単
- 出力追加にはkernel/デバイスドライバモジュールのビルドが必要
- 出力レベルを設定可能(コンソール上に出力する/しないを選択可能)
  - KERN_DEBUG、KERN_INFO、KERN_NOTICE...
- シリアルコンソールと組み合わせると出力の保存を外部に依存できる

printk

- kernel全体で利用されるので排他制御などもありそれほど軽い処理ではない
  - 追加することにより動作に影響がある可能性
  - 不具合現象が発生しなくなるか頻度が下がるなど

ftrace

- OSAKA NDS Embedded Linux Cross Forum#4で詳細に紹介(資料がダウンロード可能)
- Kernel Function Tracer
  - Linux kernel標準搭載のKernel Tracerの一つ
  - Kernel Function Graph Traceという階層表示も可能
- コンパイル時プロファイリング用プローブ関数を各関数の先頭に挿入してトレースデータを取得
- 利用するにはKernel Function Tracer/Kernel Function Graph Tracerをオンにしてkernelをビルドしておく必要がある

Kernel Function Graph Trace表示

```
# tracer: function_graph
#
# CPU  DURATION                  FUNCTION CALLS
# |     |   |                     |   |   |   |
 1)   0.720 us    |  mutex_unlock();
 1)   0.120 us    |  __fsnotify_parent();
 1)   0.120 us    |  fsnotify();
 1)               |  __sb_end_write() {
 1)               |    percpu_up_read() {
 1)               |      update_fast_ctr() {
 1)   0.120 us    |        preempt_count_add();
 1)   0.120 us    |        preempt_count_sub();
 1)   2.640 us    |      }
 1)   4.320 us    |    }
 1)   5.520 us    |  }
```

Kprobes

- OSAKA NDS Embedded Linux Cross Forum#3で詳細に紹介
  (資料がダウンロード可能)
- ブレイクポイント命令などを利用してkernelを動的に変更
- 指定位置に処理(プローブ)を追加(複数可)
- ほとんどの位置にプローブを追加可能(制限はあり)
  - 割込み処理でも追加可能
- Loadable moduleとして後からプローブを追加可能
- プローブ内にてkernel内データを変更可能
- Kprobes, Jprobes, Return probesの3種類がある
- 利用するにはKprobesをオンにしてkernelをビルドしておく必要がある

debugfs

- kernel開発者がユーザースペースへ大きなデータを渡すための方法を提供
- フルのファイル形式アクセスを提供
  - read/write以外にseekなど利用可能
- 単体では単なるインターフェースなのでデバッグのための実装が別途必要
- 利用するにはDebug Filesystemをオンにしてkernelをビルドしておく必要がある

デバッグにおけるdebugfs利用方法

- デバイスドライバにてデバッグ用に記録したい内容をローカルバッファなどに保存しておく
- ローカルバッファ上に記録する場合printkと比較して処理は軽い
- debugfsのread処理にてローカルバッファ上のデータをユーザーランドに転送して解析

debugfsの利点

- 大量のデバッグ用データを収集可能
  - デバッグ初期は関係範囲が絞りきれずいろいろなデータを収集したい
  - 発生頻度が低いと1度の現象でなるべく多くのデータを収集しておきたい
- データ収集時はローカルメモリへのアクセスのみのため追加負荷も比較的小さい

debugfsの組込み

- デバイスドライバ初期化時
  - debugfsの作成
  - デバッグデータ用のメモリ確保

debugfs作成

- ディレクトリ作成

```
struct dentry *dir;
dir = debugfs_create_dir("foo", NULL);
if (!dir)
    return -ENOMEM;
```

- ファイル作成

```
struct dentry *file;
file = debugfs_create_file("bar", 0644, dir, NULL, &bar_fops);
if (!file) {
    debugfs_remove(dir);
    return -ENOMEM;
}
```

- /sys/kernel/debug/foo/barファイルが作成されread/writeできる

debugデータ用メモリの確保

- 動的確保
  - __get_free_pages関数
    - 大きな連続領域を確保できるがサイズに限界がある
    - 連続しているので扱いやすい
  - 小さな領域を複数確保してリストやリング構造で独自に管理する
- 固定的な確保
  - Device treeで固定領域を設定
  - reserved-memoryリージョンでアドレスとサイズを指定

debugfs operations

```
static const struct file_operations bar_fops = {
    .open = bar_open,
    .read = bar_read,
    .write = bar_write,
    .release = bar_release,
    .llseek = no_llseek,
};
```

debugfs open

- static int bar_open(struct inode *inode, struct file *file)
- 行う処理
  - 排他制御(1つのアプリケーションからのみアクセス可能にしたい場合)
  - アクセス開始可能かのチェック(必要な場合)
  - read/write処理の前準備(read/writeに関連して使用するメモリの確保や変数の初期化など)
- read/writeで使うデータはポインタをfile->private_dataへ代入

openコード例

```
struct bar_data *bar_data;
bar_data = kzalloc(sizeof(*bar_data), GFP_KERNEL);
if (!bar_data)
    return -ENOMEM;
mutex_init(&bar_data->mutex);
file->private_data = bar_data;
return nonseekable_open(inode, file);
```

debugfs read

- static ssize_t bar_read(struct file *file, char __user *buf, size_t size, loff_t *ppos)
- 行う処理
  - デバッグデータをユーザーバッファー(buf)にコピー
  - 可読性が良いようにこの時点で文字列に変換してユーザーバッファー(buf)にコピーしてもよい

readコード例

```
struct bar_data *bar_data = file->private_data;
offset = *ppos;
if (size > 残りデータ量)
    size = 残りデータ量
if (copy_to_user(buf, bar_data->buf + offset, size))
    return -EFAULT;
*ppos += size;
return size;
```

debugfs release

- static int bar_release(struct inode *inode, struct file *file)
- 行う処理はopenで確保したリソースの解放
- releaseコード例

  kfree(file->private_data);

  return 0;

debugfs利用の前準備

- mount -t debugfs none /sys/kernel/debug
- デフォルトではmountされていないことがある
- mountするとデバイスドライバで作成したディレクトリとファイルにアクセス可能になる

デバッグ手法の使い分け

- 開発初期
- 全体的な動作を解析する場合
- 長期的に動作を解析する場合
  - debugfsはこちら向き

長期的に動作を解析するデバッグ

- printk+シリアルコンソールも可だがprintk出力を工夫しないと現象発生頻度が下がったり
  デバッグデータが膨大になり負荷になりやすい
- デバッグデータへのアクセスをdebugfsにする
- ローカルバッファにデバッグデータを保存してdebugfsから読み出し解析する

長期的に動作を解析するデバッグ例

- まれに処理時間がかかることがある場合
  - ftrace、Kprobesやperfなど利用すれば関数単位の処理時間であれば取得可能
  - さらに細かく計測するには時間計測処理を入れてローカルバッファに計測データや状況などを保存しておきdebugfsから読み出して検証

長期的に動作を解析するデバッグ例

- まれにおかしなデータが出力される
  - 処理途中の複数データをローカルリングバッファに保存
  - おかしなデータが出力で検出された時点で保存を停止
  - ローカルリングバッファを読み出してデータの変化を解析
  - データ変化点を絞り込んで問題点を見つける

参考

- https://www.nds-osk.co.jp/forum/onlcf3.html
- https://www.nds-osk.co.jp/forum/onlcf4.html
- Linux kernel source code
  - Documentation/filesystem/debugfs.txt
  - Documentation/kprobes.txt
  - Documentation/trace/ftrace.rst