デバイスドライバのデバッグ手法と使い分け

Yoichi Yuasa

OSAKA NDS Embedded Linux Cross Forum #9

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
- デバイスドライバデバッグ手法の使い分け

デバイスドライバデバッグ手法

- printk
- ftrace
- Kprobe
- 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
  - 動作に影響がある可能性
  - 追加すると不具合現象が発生しなくなるか頻度が下がる可能性
- 種類がいくつかあるが同じもの
  - printk(KERN_INFO “Some information\n”);
  - pr_info(“Some information\n”);
  - dev_info(dev, “Some information\n”);

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 Tracer

- trace内容
  - 実行プロセス(タスク)
  - 実行CPU番号
  - 割込み禁止/許可
  - スケジュール要求
  - ハードIRQ/ソフトIRQ
  - プリエンプト深さ
  - タイムスタンプ
  - 関数および呼び出し関係

Kernel Function Trace表示

```
# tracer: function
#
# entries-in-buffer/entries-written: 44736/44736   #P:4
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
 rcar3_create_ga-2030  [000] ...1   249.796155: mutex_unlock <-rb_simple_write
 rcar3_create_ga-2030  [000] ...1   249.796164: __fsnotify_parent <-vfs_write
 rcar3_create_ga-2030  [000] ...1   249.796165: fsnotify <-vfs_write
 rcar3_create_ga-2030  [000] ...1   249.796167: __sb_end_write <-vfs_write
 rcar3_create_ga-2030  [000] ...1   249.796167: percpu_up_read <-__sb_end_write
 rcar3_create_ga-2030  [000] ...1   249.796168: update_fast_ctr <-percpu_up_read
```

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    |  }
```

Kprobe

- 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

Kprobesではできないこと

- Kprobes機能自体へのKprobes
  - 内部で利用しているページフォルト処理とnotifier_call_chain　処理も含む
- インライン展開関数へのシンボル名でのプローブ設定
- 複数プローブ設定間での競合解決
  - プローブ処理中に別プローブ処理が起きる場合など
- 内部でのmutexとメモリ確保処理(register時は除く)
- CPU yield処理(内部はpreemption disable)

Kprobes動作

- 指定アドレス or シンボルにブレイク命令を挿入
- ブレイク例外が発生
- 例外ハンドラにてレジスタを保存
- 例外ハンドラからの通知でKprobes処理を実行
- pre_handlerを実行
- シングルステップ実行で指定位置を実行
- post_handlerを実行

debugfs

- kernel開発者がユーザースペースへ大きなデバッグデータを渡すための方法を提供
- フルのファイルアクセス提供
  - read/write以外にseekなど利用可能
- ftraceもインターフェースに利用
- 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

debugfs作成

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


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

debugfs operations

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

debugfsのmount

- mount -t debugfs none /sys/kernel/debug
- 通常はmountされていないことが多い
- 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できるようになる

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
- メモリ上に記録する場合はデバイスドライバ側の処理は軽い
- debugfsのread処理にてローカルバッファ上のデバッグデータをユーザーランドに転送して解析

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

開発初期のデバッグ

- printk一択
- デバイスドライバの動作が不安定でも利用できる
- シリアルコンソールを利用すればprintk出力ログは別のマシンに保存できるのでデバッグ中のシステムがどんなに不安定でも問題ない
- JTAG対応も無い初期のkernel移植もprintkデバッグで行う
  - 起動の初期からシリアルコンソールに出力できるearlyconの仕組みがある

全体的な動作を解析するデバッグ

- 昔、関数の最初と終わりにprintkを入れておく
- 今、まずftrace
- ftraceでデバッグ箇所を絞り込めたら
  - 関数の引数や戻り値を見たい場合はKprobe
  - 関数内部の細かな処理を見たい場合はprintk
    - kprobeだとアセンブラを見る必要があるのでprintkが容易

長期的に動作を解析するデバッグ

- デバッグデータの出力先をdebugfsにする
- printkと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やすい
- 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
  - ローカルバッファは固定的に確保するなど大きな領域を割り当てることも可能

長期的に動作を解析するデバッグ

- まれに処理時間がかかる場合があるように見える場合
  - 時間計測処理を入れてローカルバッファに計測データなどを保存しておきdebugfsから読み出して検証
  - 時間計測処理は関数処理時間であれば Kprobe(Return probe)で入れることも可能

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/kprobe.txt
  - Documentation/trace/ftrace.rst