レコーディング&リプレイデバッグツール rr

Yoichi Yuasa

OSAKA NDS Embedded Linux
Cross Online Forum #13

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にkernelを移植

本日の内容

- rr概要
- rrを利用する上でのポイント
- rrの実装(一部)

アプリケーションデバッグ手法

- printf
- gdb
  - ブレークポイント設定
  - ステップ実行
  - core解析
- trace
  - strace
  - ltrace

rr

- アプリケーション実行レコーディング(記録)/リプレイ(再生)ツール
- C/C++アプリケーションに対応
- IDEからの利用も可能
  - Visual Studio Codeなど
- Intel Nehalem(初代Core iシリーズ)以降とAMD Zen以降をサポート
- AArch64(Arm 64bit)を実験的にサポート

rrの特徴

- 実行記録/再生型のデバッグツールで一度実行を記録すると同じ現象を繰り返し調査可能
- 同等のマシンであれば記録と再生を別のマシンで行うことも可能

rrが記録するデータ

- メモリにマップされたデータ
  - 実行ファイル
  - ライブラリ
- アクセスしたファイルデータ
- 実行トレースデータ(イベント、システムコールの結果)

デバッグにおけるrr利用方法

1.調査したい現象を発生させ記録する

2.記録した動作を再生して現象を再現

3.gdbを利用して調査

rrメイン機能

- rr record <app> <app args>
  - <app>の実行を記録
- rr replay
  - 記録された<app>の実行を再生
- rr pack
  - 記録を移動できる形式にまとめる

rr record

```
$ rr record ./testapp
rr: Saving execution to trace directory `/home/foo/.local/share/rr/testapp-0'.
$ ls ~/.local/share/rr/testapp-0/
cloned_data_791_1  data  events  mmap_hardlink_3_testapp  mmaps  tasks  version
```

rr replay

```
$ rr replay
GNU gdb (Ubuntu 8.1.1-0ubuntu1) 8.1.1
…
0x00007f2ee5441090 in _start () from /lib64/ld-linux-x86-64.so.2
(rr) break main
Breakpoint 1 at 0x5604c9bc579e: file testapp.c, line 30.
(rr) c
Continuing.
Breakpoint 1, main (argc=1, argv=0x7ffec227b978) at testapp.c:30
30  {
(rr)
```

rr pack

```
$ rr pack
rr: Packed trace directory `/home/foo/.local/share/rr/testapp-0'.
$ ls ~/.local/share/rr/testapp-0
cloned_data_791_1 mmap_pack_0_libdl-2.27.so mmap_pack_3_libpthread-2.27.so mmaps
data              mmap_pack_1_libc-2.27.so  mmap_pack_4_ld-2.27.so         tasks
events            mmap_pack_2_testapp       mmap_pack_5_librrpreload.so   version
```

rrの欠点

- 複数スレッド同時実行の記録はできない
  - 同時実行の状況まで記録/再生するのは難しいため
  - 複数スレッドのアプリケーションもすべて実行を1CPUに制限して記録(2スレッド同時動作はできない)
- 記録していないプロセス(別のアプリケーション)との共有メモリなどによる競合は回避する必要がある
  - PulseAudio(socket転送に切り替えて対応)
  - X(socket転送に切り替えて対応)
  - vDSO(virtual dynamic shared object)(無効にして対応)
- 記録することで実行に時間がかかる
  - 2倍弱〜(アプリケーションの動作やファイルシステムの状況による)
  - 再生は記録より実行時間はずっと短い(システムコールを実行しないので普通の実行より早いこともある)

rrの実装(システムコール部分)

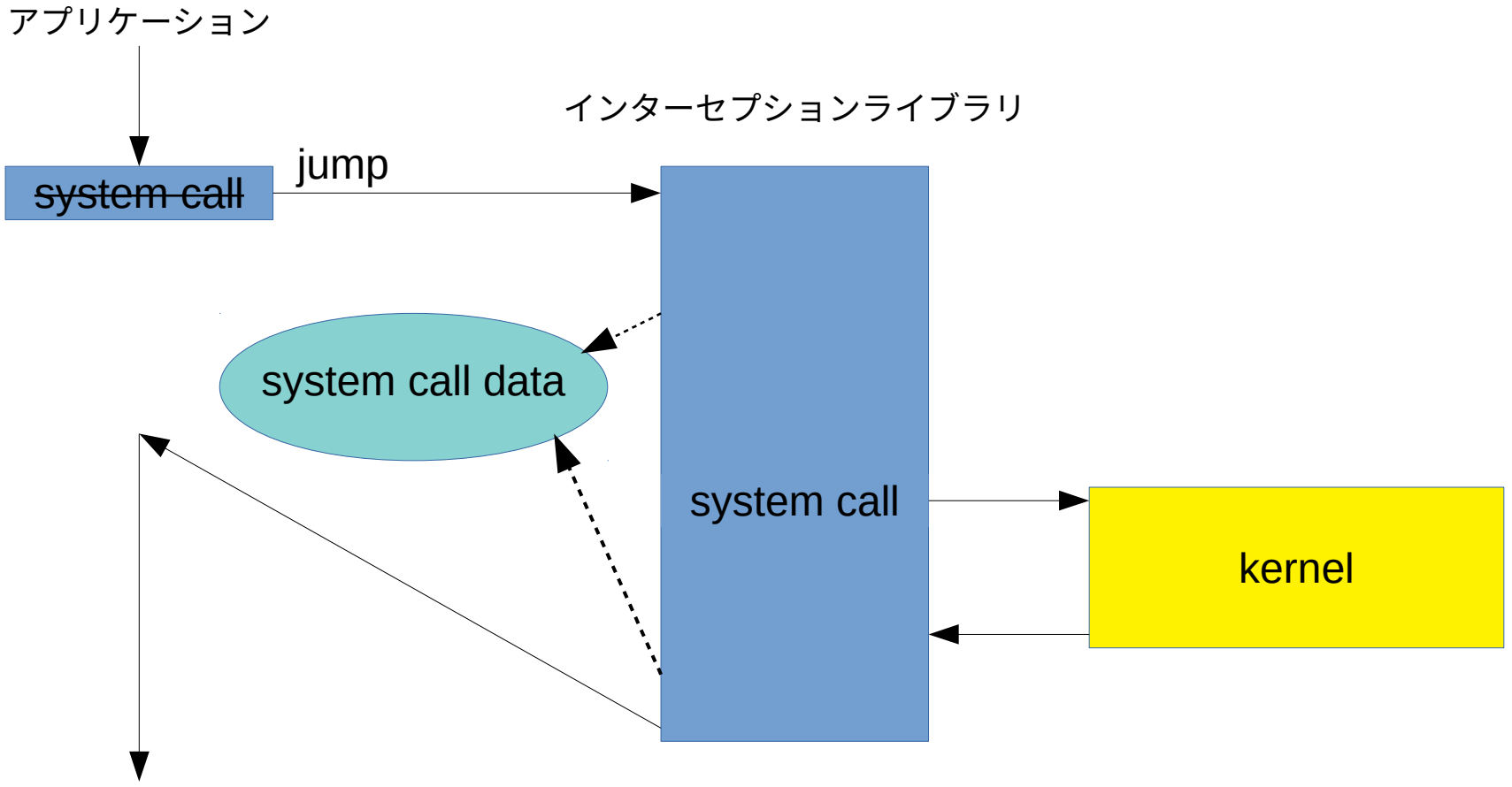
- システムコールを記録するために主にkernelの3つの機能を利用
  - ptrace
  - seccomp-bpf
  - perf event

ptraceの利用方法

- 補助的なシステムコールの検出
  - 始めてのアドレスからのシステムコールの検出
  - 静的解析では見つけられないシステムコールを検出するために利用(動的な実行コード変更など)
  - ptraceを利用することによるコンテキストスイッチのコストが大きいため利用は最小限に留める

ptraceの代替え

- 独自インターセプションライブラリの利用
  - システムコール命令を置き換えて独自ライブラリにジャンプ
  - システムコール呼び出し前のデータを保存してからシステムコールを呼び出し
  - システムコールから戻ってきたら結果データを保存して元のコードに戻る


アプリケーション
インターセプションライブラリ
system call
jump
system call data
system call
kernel

seccomp-bpf

- Secure Computing Berkeley Packet Filter
  - システムコールの実行を選択的に制御
- インターセプションライブラリとそれ以外でptraceの制御を切り替え
  - インターセプションライブラリのシステムコールはトラップせずスルー
  - それ以外のシステムコールはトラップ
  - トラップしたシステムコールは内容に応じて処理を追加

perf event

- kernel内で発生したイベントを通知してくれる機能
- rrは記録中のアプリケーションがシステムコールにブロックされたかを知りたい
  - トレースしていないアプリケーションとのシステムコールを介したやり取りでデッドロックを回避するため
  - システムコールがブロックされたら必要に応じて別のスレッドを実行する

まとめ

- rrの実装は複雑だが利用は簡単
- 問題の現象を再現させて記録できれば何度でも再生して調査できる
- AArch64も実験的にサポート

参考

- https://rr-project.org/
- https://developers.redhat.com/blog/2021/05/03/instant-replay-debugging-c-and-c-programs-with-rr
- https://arxiv.org/pdf/1705.05937.pdf

  実装の詳細やパフォーマンス測定など