KprobesによるEmbedded Linux kernel 動的解析手法

Yoichi Yuasa

OSAKA NDS Embedded Linux Cross Forum #3

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

Kprobesとは

- Linux kernelデバッグ機能の一つ
  - 他にもKGDB(リモートデバッグ)やKernel Function Tracerなど多くのデバッグ機能がある
- ブレイクポイント命令などを利用してkernelを動的に変更
  - ARMは未定義命令空間を利用
  - MIPSはbreak命令を利用など
- 指定位置に処理(プローブ)を追加
  - 複数追加も可能
- ほとんどの位置にプローブを追加可能(制限あり)
  - 割込み処理でも追加可能

Kprobes特徴

- 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
- 複数のアーキテクチャで動作
  - Intel x86(x86_64も含む)
  - PowerPC(64も含む)
  - ARM
  - MIPSなど
  - ARM64は開発中
- Kprobes, Jprobes, Return Probesの3種類がある

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

Kprobes利用準備

- Linux kernelコンフィグレーションにて
  - Kprobesをオンにしておく
    - メニューではGeneral setupの下にKprobesがある
  - loadable module supportをオンにしておく
    - Kprobes処理追加のモジュール(ドライバ)をロードするため
- 準備しておくだけでは動作に影響なし

Kprobes利用方法

- プローブ登録モジュールを作成
  - サンプルはsamples/krobes/kprobe_example.cなど
- insmod or modprobeでモジュールをロード

kprobe構造体抜粋

include/linux/kprobes.h内に定義

```
struct kprobe {
    kprobe_opcode_t addr; アドレス指定
    const char *symbol_name; シンボル名指定
    unsigned int offset; 上記からのオフセット指定
    kprobe_pre_handler_t pre_handler; 指定位置実行前のプローブ
    kprobe_post_handler_t post_handler; 指定位置実行後のプローブ
    kprobe_fault_handler_t fault_handler;
                                pre/postのプローブで例外が発生した時のプローブ
}
```

kprobe登録

```
static struct kprobe kp;
static int __init kprobe_init(void)
{
    kp.pre_handler = handler_pre;
    kp.post_handler = handler_post;
    kp.fault_handler = handler_fault;
    return register_kprobe(&kp);
}
module_init(kprobe_init)
```

kprobe位置登録

- アドレス指定
  - struct kprobeのaddrに設定

```
static struct kprobe kp = {
    .addr = 0xffffffff8107f150,
};
```

- シンボル名指定
  - 関数名をstruct kprobeのsymbol_nameに設定

```
static struct kprobe kp = {
    .symbol_name = “_do_fork”,
};
```

pre_handler

- 指定したプローブ位置を実行する前に呼ばれる

```
int handler_pre(struct kprobe *p,
                struct pt_regs *regs)
{
    レジスタ内容の表示や変更など
    return 0;
}
```

struct pt_regs

- CPUが持つ標準レジスタを格納する構造体
- 例外発生時はこの構造体にレジスタの値が保存される
- ARMの場合(arch/arm/include/uapi/asm/ptrace.h)

```
struct pt_regs {
        long uregs[18];
};
#define ARM_cpsr        uregs[16] カレントプログラムステータスレジスタ
#define ARM_pc          uregs[15]
#define ARM_lr          uregs[14]
#define ARM_sp          uregs[13]
…
#define ARM_r0          uregs[0]
#define ARM_ORIG_r0  uregs[17]
```

post_handler

- 指定したプローブ位置を実行後に呼ばれる

```
void handler_post(struct kprobe *p,
                  struct pt_regs *regs,
                  unsigned long flags)
{
    レジスタ内容の表示やチェックなど
    flagsは多くのアーキテクチャで0固定
}
```

fault_handler

- pre_handlerとpost_handlerで例外が発生したときに実行される

```
int handler_fault(struct kprobe *p,
                  struct pt_regs *regs,
                  int trapnr)
{
    例外原因を調べるためのレジスタ内容の表示など
    return 0;
}
```

Jprobes特徴

- Kprobesで実装
- 指定関数の呼び出しを同じ引数の関数で置換える
- 置換えた関数で簡単に引数にアクセス

jprobe構造体

include/linux/kprobes.h内に定義

```
struct jprobe {
    struct kprobe kp;
    void *entry; 置換える関数
};
```

jprobe登録

```
static struct jprobe my_jprobe = {
    .kp = {
        .symbol_name = “_do_fork”,
    },
    .entry = j_do_fork,
};
```

jprobe登録

```
static int __init jprobe_init(void)
{
    return register_jprobe(&my_jprobe);
}
module_init(jprobe_init)
```

jprobe entry

- 引数は_do_fork関数とまったく同じにする
- 引数を簡単に表示できる

```
long j_do_fork(unsigned long clone_flags,
                    unsigned long stack_start, unsigned long stack_size,
                    int __user *parent_tidptr, int __user *child_tidptr,
                    unsigned long tls)
{
    pr_info(“jprobe: clone_flags = 0x%lx, stack_start = 0x%lx “
            “stack_size = 0x%lx¥n”, clone_flags, stack_start, stack_size);
    jprobe_return();
    return 0;
}
```

Return Probes特徴

- Kprobesで実装
- プローブする関数の呼び出し時と終了時にプローブが呼び出される
- 関数処理時間計測などに有効

Return Probes動作

- Return Probes用kprobeを設定
- kprobe pre_handlerでentry_handlerを実行
- 上実行後にpt_regsのリターンアドレスを差し替え
  - 差し替えたリターンアドレスにはkprobeを予め設定
- プローブされている関数(_do_forkなど)が普通に実行される
- 関数が終了してリターンすると差し替えた関数が呼び出される
- kprobe pre_handlerでhandlerを実行
- インストラクションポインタを本来のリターンアドレスに戻す
- 本来のリターン先へ戻る(例外からの復帰時)

kretprobe構造体抜粋

include/linux/kprobes.h内に定義

```
struct kretprobe {
    struct kprobe kp;
    kretprobe_handler_t handler; 関数終了時のプローブ
    kretprobe_handler_t entry_handler;
                                        関数呼び出し時のプローブ
    int maxactive; 並行してプローブする数
    size_t data_size;
                handler, entry_handlerで利用するデータ領域サイズ
}
```

kretprobe登録

```
static struct kretprobe_my_kretprobe = {
    .kp = { .symbol_name = “_do_fork”, },
    .handler = ret_handler,
    .entry_handler = entry_handler,
    .data_size = sizeof(struct my_data),
    .maxactive = 20,
};
Static int __init kretprobe_init(void)
{
    return register_kretprobe(&my_kretprobe);
}
module_init(kretprobe_init)
```

entry_handler

```
int entry_handler(struct kretprobe_instance *ri,
                  struct pt_regs *regs)
{
    struct my_data *data;
    data = (struct my_data *)ri->data;
                  register時にdata_size分確保してくれている
    data->entry_stamp = ktime_get(); 内部時間を取得
    return 0;
}
```

handler(関数終了時のプローブ)

```
int ret_handler(struct kretprobe_instance *ri,
                        struct pt_regs *regs)
{
    struct my_data *data = (struct my_data *)ri->data;
    s64 delta;
    ktime_t now = ktime_get(); 内部時間を取得
    delta = ktime_to_ns(ktime_sub(now, data->entry_stamp));
        entry_handler時データとの差分から実行時間を算出
    return 0;
}
```

Kprobesのオーバーヘッド

- i386 Pentium M 1495MHz
  - Kprobes: 0.57 μsec
  - Jprobes: 1.00 μsec
  - Return Probes: 0.92 μsec
  - Kprobes + Return Probes: 0.99 μsec
  - Jprobes + Return Probes: 1.40 μsec

参考

- Documentation/kprobes.txt
- kernel/krobes.c
- arch/mips/kernel/kprobes.c
- arch/arm/probes/kprobes/*
- include/linux/kprobes.h