ARM gcc バッドノウハウ集

Kunihiko IMAI <bak @ d2.dion.ne.jp>

2009 年 1 月 11 日

ARM gcc にまつわるいやらしい現象とその対策をまとめてみました.

目 次

1 はじめに

XScale などの ARM CPU アーキテクチャ用の gcc や binutils などを使うに当たっての注意点をまとめてみました.

2 責任の放棄

この文書に関する一切の責任は放棄します.

3 想定している読者と環境

読者としては，x86 で一通りの C プログラムが書ける人を想定しています.

話題としては ARM toolchain に関することで，OS には依存しないはずです．が，説明の都合上，ARM Linux を仮定します.

4 アラインメント

まずは肩慣らし．アラインメントについては RISC アーキテクチャでは一般的な話で，ARM に限ったことではありません.

x86 以外の アーキテクチャでは，２バイト整数型や４バイト整数型の変数を配置できるメモリアドレスに制限があります.

- ２バイト整数型は，偶数アドレスに
- ４バイト整数型は，４で割り切れるアドレスに

配置しなくてはいけません.

4.1 アラインメント条件を満たさないと

アラインメント条件を満たさないメモリアクセスが発生した場合，ARM では例外が発生します．が，ARM Linux 上ではこの例外をカーネルで捕捉し,「うまく処理する」ようになっています.

ARM Linux では `/proc/cpu/alignment` で，この例外の捕捉状況を表示したり，処理の変更を指示することができます.

```
$ cat /proc/cpu/alignment
User:           0
System:         0
Skipped:        0
Half:           0
Word:           0
Multi:          0
User faults:    0 (ignored)
```

試しに以下のようなプログラムを実行し，x86 上の挙動と比較してみましょう.

```
#include <stdio.h>

main ( void )
{
        char *str = "\x01\x23\x45\x67\x89\xab\xcd\xef";
```

```
        unsigned *u = (unsigned *)(str + 1);

        printf ( "%08x\n", *u );
}
```

x86 上で実行すると以下のような結果が得られます.

```
$ ./a.out
89674523
```

ARM 上で実行すると以下のような結果が得られます.

```
$ ./a.out
01674523
```

このとき，/proc/cpu/alignment の内容は以下のように変化しました．カーネルで例外を捕捉し，User の項目がカウントアップしていることがわかります.

```
$ cat /proc/cpu/alignment
User:           1
System:         0
Skipped:        0
Half:           0
Word:           0
Multi:          0
User faults:    0 (ignored)
```

出力結果から考えられるメモリアクセスの様子を，以下に図示します．どうしてこんな結果が得られるのか，なんとなく想像できますね.

```
+3 +2 +1 +0
67 45 23 01      str のダンプ
ef cd ab 89

+3 +2 +1 +0
02 01 00         x86 でのアクセス
         03      （値は unsigned アクセスでのバイト位置）

+3 +2 +1 +0
02 01 00 03      ARM でのアクセス（同上）
```

4.2 由緒正しい対策

結論から言うと

「アラインメントをまたぐアクセスをするプログラムを書かない」

これに尽きます.

アラインメントをまたがないプログラムは，x86 上でも何の問題もなく動作します．また，x86 上であってもアラインメントをまたぐメモリアクセスは余計なメモリアクセスが入るため，パフォーマンス上，悪影響があります.

既存のプログラムで x86 以外のアーキテクチャで動作実績の無いプログラムの場合,「アラインメントの問題があるかもしれない」と疑ってかかる必要があるでしょう.

4.3 バッドノウハウな対策

4.1 (ARM Linux ではうまく処理する) と先に書きましたが，その内容をもう少し掘り下げてみます.

/proc/cpu/alignment に数値を書き込むことにより，例外ハンドラの振る舞いを変更することができます.

たとえば，以下のように '2' を設定すると

```
# echo 2 >/proc/cpu/alignment
# cat /proc/cpu/alignment
User:           1
System:         0
Skipped:        0
Half:           0
Word:           0
Multi:          0
User faults:    2 (fixup)
```

先のプログラムの実行結果はは以下のように，x86 と同様になります．つまり，例外ハンドラで「x86 と同様のメモリアクセス」をエミュレートしていることになります.

```
$ ./a.out
89674523
```

/proc/cpu/alignment に書き込む値とハンドラの動作は以下の通りになります．また，各値の bit OR で，複数の動作を指示することもできます．例えば '3' を書き込むと warn と fixup の両方の処理を行います.

0

ignore．例外を単に無視する

1

warn．例外発生を dmesg に出力

2

fixup．例外発生の命令を解釈し，メモリアクセスのつじつまを合わせる

4

signal．バスエラーを発生させる．ユーザプログラムは core を吐いて終了

これで一応 x86 互換のメモリアクセスは実現できました．が，あくまでも「例外ハンドラによる補正」であることを忘れないように．

aligned

read →…

x86 unaligned

read → read →…

ARM unaligned(fixup)

read →（トラップ）→エミュレート→（復帰）→…

aligned access の場合は単なる read サイクルです．が，fixup による補正処理が入った場合，ユーザ空間 → カーネル空間 → ユーザ空間と，コンテキストスイッチが入ります．このため，単なるメモリアクセスに比べ，非常に重い処理となっています．この方法でアラインメント問題を回避しようとする場合，注意が必要です．

5 パディング

パディングとは，アラインメント条件を満たすために構造体に勝手に入ってしまう「詰めもの」のことです．これもアラインメントと同様，RISC アーキテクチャのプロセッサでは一般的な話題です．

5.1 例

パディング自体は x86 でも生じます．が，以下のプログラムは，x86 と ARM でパディングの挙動が異なる例です．

```
#include <stdio.h>

struct s0 {
        unsigned char a, b, c;
};

struct s1 {
        char d;
        struct s0 e;
};


main ( void )
{
        struct s1 s;
```

```
        void *s_a = (void *)&s,
                *d_a = (void *)&s.d,
                *e_a = (void *)&s.e;

        printf ( "d = %d\n", d_a - s_a );
        printf ( "e = %d\n", e_a - s_a );
}
```

このプログラムの x86 での実行結果を以下に示します.

```
$ ./a.out
d = 0
e = 1
```

ARM での実行結果は以下のようになります.

```
$ ./a.out
d = 0
e = 4
```

5.2 由緒正しい対策

やはり

「パディングに依存しないプログラムを書く」

というのが正しい対策でしょう.

特に，データストリームに対して構造体のポインタを設定し，直接値を読み取る，というようなプログラムは書くべきではないでしょう.

5.3 バッドノウハウな対策

構造体の宣言に gcc 拡張の `__attribute__ ((packed))` の属性を付けることによって，パディングを抑止することができます.

```
#include <stdio.h>

struct s0 {
        char a, b, c;
} __attribute__ ((packed));

struct s1 {
        char d;
```

```
        struct s0 e;
} __attribute__ ((packed));


main ( void )
{
        struct s1 s;
        void *s_a = (void *)&s,
                *d_a = (void *)&s.d,
                *e_a = (void *)&s.e;

        printf ( "d = %d\n", d_a - s_a );
        printf ( "e = %d\n", e_a - s_a );
}
```

このプログラムを ARM 上で実行すると

```
$ ./a.out
d = 0
e = 1
```

と，x86 と同様の結果が得られます.

ただし，x86 で付くはずのパディングまで削除してしまうので，使用には注意が必要です．が,「x86 環境とと同一の実行結果を得る」という目的は達することができるでしょう.

6 char 型

さて，ここらから ARM toolchain 特有の話題に入ります.

ARM gcc では，char 型で宣言した変数は unsinged char として扱われます．x86 などの他の gcc では signed char として扱われます．普通はこちらの方ですね.

char が符号あり・なし，ということ自体は C の規約では「実装依存」であるため，C としては間違っていません．が，世の中のほとんどの C プログラムは char が符号つきであるという前提で書かれていますし，読者の認識もそうだと思います.

というわけで，既存の C のプログラムをコンパイルして使用しようとする場合，これが問題となる場合があります.

6.1 例

以下のようなプログラムをコンパイルし，実行してみます.

```
#include <stdio.h>

main ( void )
{
        char c1 = 0xff, c2 = 0x01;

        if ( c1 > c2 ) {
                printf ( "c1 > c2\n" );
        } else {
                printf ( "c1 <= c2\n" );
        }
}
```

x86 では signed char として扱われるので，c1 = -1，c2 = 1 で，実行結果は

```
$ ./a.out
c1 <= c2
```

となりますが，ARM では unsigned char として扱われるので c1 = 255，c2 = 1 となり，実行結果は

```
$ ./a.out
c1 > c2
```

となります.

6.2 由緒正しい対策

char 型変数を符号つきの文脈で使用する場合は，明示的に signed char と宣言することでこの問題を解決することができます.

もちろん，C として移植性をまじめに考えるとこのようにプログラムすべきなのですが… 慣習としてはちょっと受け入れがたいものがあるかもしれません.

6.3 バッドノウハウな対策

ARM gcc のコンパイルオプションで -fsigned-char と指定することで，char 型を signed char と扱わせることができます.

もちろん，先のプログラムでも以下のように x86 と同様の結果が得られます.

```
$ gcc -fsigned-char char.c
$ ./a.out
c1 <= c2
```

7 soft fp と hard fp

ここで言う fp とは浮動小数点演算のことです.

7.1 一般的な問題

かつては x86 も，FPU （Floating Poing Unit; 浮動小数点演算ユニット）は別付けのオプションでした．SoC ではその用途上，浮動小数点演算を駆使する機会はあまりありません．このため，現在でも浮動小数点演算ユニットを搭載していない石が多く存在します.

が,「たまには」浮動小数点演算が必要となる機会もあります．この場合，浮動小数点演算を処理させるにはいくつかの方法があります.

7.1.1 FPU エミュレーション

浮動小数点演算ユニットの無い CPU で浮動小数点演算命令を実行しようとすると，未定義命令等の例外が発生します．この例外を捕捉し，例外ハンドラ中で浮動小数点演算命令をエミュレートし，例外発生元に復帰します.

この場合の長所は，浮動小数点演算ユニット用の実行プログラムがそのまま実行できることです．ただし，浮動小数点演算ユニットがある場合に比べ，実行速度は格段に落ちます.

短所は,「オーバヘッドが大きい」ことです．浮動小数点演算命令ごとに例外が発生し，コンテキストスイッチが発生します．そして，例外ハンドラ中でエミュレーションを行い，元のコンテキストに復帰します.

7.1.2 浮動小数点演算ライブラリとのリンク

ここでは，プログラムのコンパイル時に浮動小数点演算ライブラリとリンクするという方法を検討します．このライブラリ中で浮動小数点演算を整数演算で処理しよう，というものです．gcc では `--msoft-float` オプションで使用することができます.

この場合は実行プログラムのレベルで浮動小数点演算ライブラリを呼び出します．このため，浮動小数点演算を行っても例外の発生は無く，コンテキストスイッチも発生しません.

が，この実行プログラムを FPU がある環境に持っていっても，実行速度は上がりません．せっかく持ってる FPU を使わずに演算ライブラリの整数演算で浮動小数点演算を行うためです.

7.1.3 実行速度

これまで出てきた３つの浮動小数点演算処理の実行速度は

（遅） FPU エミュレーション < 浮動小数点演算ライブラリ < 実 FPU での演算 （速）

となります.

7.2 ARM 固有の問題

と，ここまでは一般的な浮動小数点演算の話でした．ARM の場合は更に一癖あります．

7.2.1 まずは現象から

説明の前に現象から入ってみましょう．

以下の 2 つの C のソースプログラムを用意します．

```
/* foo.c */
int foo ( int a, int b )
{
  return a + b;
}
```

```
/* main.c */
#include <stdio.h>

extern int foo ( int a, int b );

main ( void )
{
  printf ( "%d\n", foo ( 1, 2 ) );
}
```

関数を別ソースファイルに記述した，何の変哲もない 2 本の C プログラムです．

これらを以下の手順でコンパイルしてみます．

```
$ gcc foo.c -msoft-float -c
$ gcc main.c -c
$ gcc foo.o main.o
```

foo.c のほうは -msoft-float オプション付きで，main.c のほうはオプション無しでコンパイルし，オブジェクトファイルをリンクします．これを x86 や PowerPC 上で実行すると，何の問題もなく a.out 実行ファイルが作成されます．

ところが，ARM toolchain 上ではリンクの段階で

```
$ gcc main.o foo.o
/usr/bin/ld: ERROR: /usr/lib/gcc/arm-linux-gnu/4.0.4/libgcc_s.so uses hardware
FP, whereas a.out uses software FP
/usr/bin/ld: failed to merge target specific data of file /usr/lib/gcc/arm-linu
x-gnu/4.0.4/libgcc_s.so
/usr/bin/ld: ERROR: /lib/libc.so.6 uses hardware FP, whereas a.out uses softwar
```

```
e FP
(略)
collect2: ld returned 1 exit status
```

というエラーでリンクに失敗します.

7.2.2 原因

大量のエラーメッセージが出力されますが，ほとんどは以下のようなエラーです.

```
/usr/bin/ld: ERROR: /usr/lib/gcc/arm-linux-gnu/4.0.4/libgcc_s.so uses hardware
FP, whereas a.out uses software FP
```

ということは，このエラーがポイントのようです.

メッセージを素直に解釈すると「一方は hardware FP で他方は software FP である」と読めます．が，このエラーメッセージで google 検索をすると,「それは toolchain のバグだ．最新版にアップグレードせよ」など,意味不明のアドバイスが得られます．しかし，それがどういうバグなのか言及しているページは見付からないようです[1].

というわけで，目の前の現象を見てみることにします.

オブジェクトファイルの形式を調べてみましょう．それには readelf コマンドを使用します.

以下は，x86 のオブジェクトファイルのヘッダ情報を出力した例です.

```
$ readelf -h foo.o
ELF ヘッダ:
  マジック:  7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  クラス:                            ELF32
  データ:                            2 の補数、リトルエンディアン
  バージョン:                        1 (current)
  OS/ABI:                            UNIX - System V
  ABI バージョン:                    0
  タイプ:                            REL (再配置可能ファイル)
  マシン:                            Intel 80386
  バージョン:                        0x1
  エントリポイントアドレス:          0x0
  プログラムの開始ヘッダ:            0 (バイト)
  セクションヘッダ始点:              192 (バイト)
  フラグ:                            0x0
  このヘッダのサイズ:                52 (バイト)
  プログラムヘッダサイズ:            0 (バイト)
  プログラムヘッダ数:                0
  セクションヘッダ:                  40 (バイト)
  Number of section headers:         9
  Section header string table index: 6
```

で，x86 オブジェクトファイルをこのコマンドで調べてみると，以下のように違いが見付かりません.

[1]まるで「牛の首」のような…

```
$ readelf -h foo.o main.o

ファイル: foo.o
ELF ヘッダ:
  マジック:  7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  …（略）…

ファイル: main.o
ELF ヘッダ:
  マジック:  7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  …（略）…
```

が，ARM オブジェクトファイルでは

```
$ readelf -h foo.o main.o

ファイル: foo.o
ELF ヘッダ:
  マジック:  7f 45 4c 46 01 01 01 61 00 00 00 00 00 00 00 00
  …（略）…
  フラグ:                            0x200, GNU EABI, software FP
  …（略）…

ファイル: main.o
ELF ヘッダ:
  マジック:  7f 45 4c 46 01 01 01 61 00 00 00 00 00 00 00 00
  …（略）…
  フラグ:                            0x0
  …（略）…
```

と，2つのオブジェクトファイル間で相違が見られ，

> ELF ヘッダ上に soft FP か hard FP かの情報が埋め込まれている

ということがわかります．リンカステージではこれを見て「種類か違うのでリンクしてはいけない」と判断し，エラーメッセージを出力してるわけです．

7.2.3 由緒正しい対策

そもそも，soft FP と hard FP では浮動小数点の内部表現が一致しているとは限らないので，これらのオブジェクトファイルを混ぜてリンクすべきではありません．

`readelf` コマンドでオブジェクトファイルの形式を調べ，soft FP か hard FP のいずれかに形式を統一するよう，コンパイルオプションを調整しましょう．

7.2.4 バッドノウハウな対策

実は，リンカオプションに `--no-warn-mismatch`（gcc へは`-Wl,--no-warn-mismatch`）と追加すると，soft FP と hard FP のオブジェクトファイルを強引にリンクすることができます．

```
$ gcc -Wl,--no-warn-mismatch foo.o main.o
```

「プログラム内部では浮動小数点演算は全く使ってないのだけど，なぜかリンカステージで『浮動小数点がうんぬん』のエラーで跳ねられる」という場合は，この方法も有効でしょう.

8 EABI と legacy ABI

8.1 ABI とは

ここでいう ABI (Application Binary Interface) とは，C コンパイラにおいて

- 構造体への要素の詰め込みかた
- 関数呼出し時に
  - どのレジスタを関数の引数として利用し，どのレジスタに戻り値が入るか
  - どのレジスタを呼出元でスタックに退避し，どのレジスタは呼び出し先で破壊しても問題ないか
  - そもそもどのレジスタをスタックポインタとして利用するか

などを定義したものです.

8.2 ABI の種類

現在，ARM Linux 上で使われている ABI には以下の 2 つの種類があります.

8.2.1 legacy ABI

従来から ARM gcc で使用されている ABI．gcc でのコンパイルオプションは `-mabi=apcs-gnu` です.

8.2.2 ARM EABI

ARM holdings が定義[2]した ABI です．gcc 4.1 以降で使用可能で，コンパイルオプションは `-mabi=aapcs-linux` です.

[2] http://www.arm.com/products/DevTools/ABI.html

8.3 違い

これらの ABI の違いは，Linux カーネルの設定ファイル (linux-2.6.x/arch/arm/Kconfig) によると

> Since there are major incompatibilities between the legacy ABI and EABI, especially with regard to structure member alignment, this option also changes the kernel syscall calling convention to disambiguate both ABIs and allow for backward compatibility support (selected with CONFIG_OABI_COMPAT).

ということだそうです．構造体のアラインメントに違いがあるようです．

以下のプログラムで違いを見てみます．

```
/* abitest.c */
#include <stdio.h>

struct foo {
        char a;
};

main ( void )
{
        printf ( "sizeof(struct foo) = %d\n", sizeof(struct foo));
}
```

lagacy ABI のほうは

```
$ gcc -mabi=apcs-gnu abitest.c
$ ./a.out
sizeof(struct foo) = 4
$
```

ARM EABI のほうは

```
$ gcc -mabi=aapcs-linux abitest.c
$ ./a.out
sizeof(struct foo) = 1
$
```

という結果が得られます．構造体のアラインメントの違いが見られますね．

8.4 見分けかた

手元にあるプログラムが ARM EABI か legacy ABI か見分けるには `readelf` を使います．

```
$ gcc -mabi=aapcs-linux -c abitest.c     # ARM EABI でコンパイル
$ readelf -h abitest.o
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
...
（略）
...
  Flags:                             0x4000000, Version4 EABI
...
（略）
...
$ gcc -mabi=apcs-gnu abitest.c     # legacy ABI でコンパイル
$ readelf -h abitest.o
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 61 00 00 00 00 00 00 00 00
...
（略）
...
  Flags:                             0x400, GNU EABI, VFP
...
（略）
...
```

（注）この節は後から追加しており，前の `readelf` と出力フォーマットが違ってます.

つまり，Flags: の項が

- legacy ABI では GNU EABI
- ARM EABI では Version4 EABI

となっていることで見分けがつきます.

8.5 Debian での対応状況

Debian では lenny（執筆時点での test 版）で EABI 化されたユーザランドが用意されました．アーキテクチャ名は

arm

legacy ABI のユーザランド

armel

ARM EABI のユーザランド

となっています.

8.6 影響と対策

「システム内部で ABI を統一せよ」

これが鉄則です.

8.6.1 実行プログラムとライブラリ

もちろん，ABI を統一しておかなくてはいけません．ライブラリ関数が仮定している構造体のデータ構造と呼出元が仮定している構造体のデータ構造が食い違っていては，マトモに動作することは期待できません.

8.6.2 カーネルとユーザランドプログラム

これらも ABI を統一しておく必要かあります．これも同様の理由です.

8.7 バッドノウハウな対策

最近の Linux カーネルでは CONFIG_AEABI と CONFIG_OABI_COMPAT オプションを有効にすると

- カーネル本体は EABI で
- システムコールは EABI と legacy ABI 両対応で

ビルドすることができます.

ただし，legacy ABI プログラムからのシステムコールは引数を EABI のデータ構造に変換するため，多少のオーバヘッドが生じることになります．また，以前のバージョンではこの変換に不具合があったらしく，apache などのプログラムが動作しないこともありました．というわけで，カーネルを自分でコンパイルできる場合はカーネルの ABI をユーザランド側に合わせてしまえば良いわけで，わざわざ使う機能ではないと思います.

9 memcpy()

9.1 現象

まずは以下のコードをコンパイルしてみましょう.

```
#include <string.h>

int foo ( char *to, char *from )
{
        char buf[1024];
        memcpy ( buf, "12345", 6 );
}
```

```
$ gcc -O0 -c foo.c
$ nm foo.o
00000000 T foo
$
```

あれ，memcpy() はどこに行ってしまったんでしょう．-O0 を指定したので最適化はしてないはずです．

こんどはこんなプログラム．

```
#include <string.h>

int foo ( char *to, char *from, int len )
{
        memcpy ( to, from, len );
}
```

同様にコンパイルしてみます．

```
$ gcc -O0 -c foo2.c
$ nm foo2.o
00000000 T foo
         U memcpy
```

今度は memcpy() はありますね．

9.2 インライン展開

この現象は，gcc により memcpy() がインライン展開されるために生じます．おそらくは x86 のストリング命令を有効に利用するためにこのような最適化が入っているのでしょう．

gcc の info によると，このようなインライン展開は

- memcpy
- mempcpy
- memmove
- memset
- strcpy
- stpcpy
- strncpy
- strcat
- strncat

で行われるそうです．

9.3 問題点

で，このようなインライン展開が，文脈として正しいところで正しいコードに展開されている限りは何の問題もありません．

が，gcc のバージョンによっては，誤ったインライン展開が行われることがあります．私が見たことがあるものでは,「コピー元とコピー先のアドレスがコンパイル時に決定できないにも関わらず，32bit word align を仮定したコードでインライン展開されている」というものがありました．そのときコンパイルしていたのはブートローダだったので，例外が発生した時点でプログラムは動作しなくなりました．

9.4 対策

誤ったインライン展開をしない gcc に入れ換える，というのが正しい対処法でしょう．

が，何らかの都合でバグ入りのバージョンを使わなければならない場合は

```
void *my_memcpy( void *dest, const void *src, size_t n )
{
        char *s = (char *)src, *d = (char *)dest;
        while ( n-- )
                *(d++) = *(f++);
        return dest;
}

int foo ( char *to, char *from )
{
        char buf[1024];
        my_memcpy ( buf, "12345", 6 );
}
```

と，私家版の memcpy を `my_memcpy()` として定義してやれば，誤ったインライン展開から逃れることができるでしょう．

10 まとめ

RISC CPU でのプログラミングが無い場合，アラインメントやパディングでつまづくのではないかと思います．他の RISC CPU でのプログラミング経験がある場合でも，char 型の扱い・soft fp / hard fp のオブジェクトファイルの不整合・ABI の違いという点はつまづきやすいポイントだと思います．場合によっては memcpy のインライン展開に悩まされることもあるでしょう．

ARM gcc 環境を利用する場合，これらのことを頭の片隅にでも覚えておくと良いでしょう．

（おしまい）