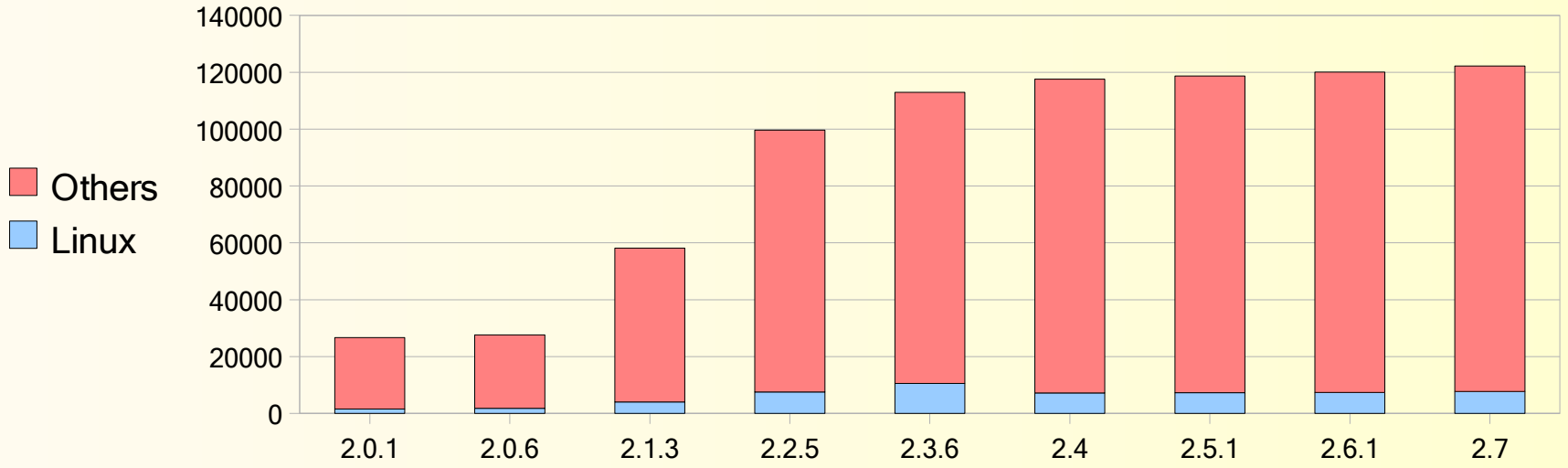
Linux kernel の glibc 依存症

しらい たかし

OSC2008 関西
2008/07/19

いきなりですが du で数えてみた

Glibc	Version	2.0.1	2.0.6	2.1.3	2.2.5	2.3.6	2.4	2.5.1	2.6.1	2.7
	Date	97/02/04	97/12/29	00/02/25	02/01/22	05/11/04	06/03/06	07/08/01	07/08/01	07/10/20
All	KBytes	26,720	27,640	58,164	99,616	112,888	117,552	118,616	120,040	122,200
Linux	linuxthreads	544	694	812	1,368	2,356	0	0	0	0
	linuxthreads_db	0	0	208	212	228	0	0	0	0
	sysdeps/ unix/ sysv/ linux	1,004	1,156	2,996	5,976	6,420	5,548	5,576	5,668	5,892
	nptl/ sysdeps/ unix/ sysv/ linux	0	0	0	0	1,532	1,656	1,696	1,716	1,796
	KBytes	1,548	1,850	4,016	7,556	10,536	7,204	7,272	7,384	7,688
	Ratio	5.79%	6.69%	6.90%	7.59%	9.33%	6.13%	6.13%	6.15%	6.29%



調査結果の考察

- 2.3.6 までは順調に Linux 固有コードが増加。
- 2.4 で linuxthread 辺りが別パッケージから統合。
  - Linux 固有コード率が減少。
- 統合されたと言うよりは巧妙に隠蔽した？
  - Linux で培った技術を他にも展開、と考えよう。
- 最近の Linux で使う 2.8 は ftp.gnu.org に無い。
  - glibc project の snapshot にある。
  - release 版が間に合わない程ペースが速い？

さてここでライブラリに関するおさらい

- ライブラリって何？
  - OS が用意したプログラマ向けサブルーチン群。
- システムコールとはどう違うの？
  - システムコール : kernel 中にあってメモリに常駐。
  - ライブラリ : 起動されるまでは読込まれない。
- ライブラリの実体はどこから読込まれるの？
  - スタティックリンク : バイナリに埋込み。
  - ダイナミックリンク : 実行時に共有ライブラリから。

共有ライブラリの仕組みもおさらいしよう

- 共有ライブラリはランタイムリンカ (ld.so) が制御。
- どの共有ライブラリを使うかは ldconfig(8) が管理。
  - /etc/ld.so.conf にルールを記述しておく。
- 共有ライブラリの判別には soname が使われる。
  - ライブラリバージョンを含めた識別名称。
- もっと詳しく知りたい人は JF(↓) を読もう。
  http://www.linux.or.jp/JF/JFdocs/Program-Library-HOWTO/

今更ですが glibc とは？

- 「 GNU C library 」の略。(「 C library 」→ libc)
- GNU システム (GNU/Linux を含む) 用の libc 。
  - 「 GNU/Linux って言え！」by RMS
- UNIX 系 OS は C ベースなので libc は必須。
- 一応 FSF 管轄の汎用品 (*BSD や Solaris でも利用可) だが、実態としては殆ど Linux 専用かも。
  - glibc のメイン開発陣は Linux 畑。
  - glibc project の運用は RedHat 。

glibc が標準 libc なのは Linux だけ

- 一般の OS では OS ベンダが libc を提供。
- フリーの *BSD もコアチームが独自 libc を開発。
  - 基本的には glibc 不要。
- Linux は自前で libc を用意出来ていない。
  - kernel と libc の開発が別々。
- kernel の機能は libc に実装されるまで使えない。
  - システムコールも一応 libc にラッパ関数がある。
- kernel と libc のバージョンを気にしておく必要あり。

では libc からシステムコールを呼ぼう

- syscall(2) で kernel 内のシステムコールを呼出す。
  - i386 ではソフトウェア割込みを利用。
- 例えば stat(2) の実体はこんな感じ。

```
int stat(const char *path, struct stat *buf)
{
        return syscall(SYS_stat, path, buf);
}
```

SYS_stat: kernel で割振ったシステムコール ID 。

- libc 内にラッパ関数がシステムコールの数だけ必要。
  - BSD だとアセンブリを Makefile で自動生成。

ところが kernel は進化する

- stat(2) には複数の実装がある。(Linux: 3 種類)
  - oldstat st_dev や st_ino が 16 ビット幅。
  - stat st_size や st_rdev が 32 ビット幅。
  - stat64 st_size や st_rdev が 64 ビット幅。
- それぞれの実装には別々のシステムコール ID が割振られている。
- kernel 側に合わせた libc の実装が必要となる。
  - libc が kernel の実装を知らないとどうなる？

実は glibc はこんなことしてます

- glibc は OS 標準じゃないので kernel を選べない。
  - libc 側でバージョン管理を行なう。
- 例えば stat(2) の実体はこんな感じ。

```
int stat(const char *path, struct stat *buf)
{
        return xstat(_STAT_VER, path, buf);
}
```

  _STAT_VER: stat(2) の仕様を示すバージョン番号。
- -O オプション付でコンパイルすると stat(2) は xstat() にインライン展開されて stat(2) が全く残らない。

もう少し詳しく見てみよう

- xstat() の実体はこんな感じ。

```
int xstat(int vers, const char *path, struct stat *buf)
{
        struct kernel_stat kbuf;
        int result;

        if (vers == _STAT_VER_KERNEL)
                return syscall(SYS_stat, path, buf);
        result = syscall(SYS_stat, path, kbuf);
        if (result == 0)
                result = xstat_conv(vers, &kbuf, buf);

        return result
}
```

要求される stat(2) の仕様が kernel の最新版と異なる場合はコンバートして返す。

旧バイナリを新 kernel 上で使う場合

- スタティックなバイナリの場合。
  - 旧システムコール ID で新 kernel が呼ばれる。
- ダイナミックなバイナリの場合。
  - xstat() があれば、旧 stat(2) 仕様を新 kernel 用 libc の xstat() が処理する。
  - xstat() が無ければ、stat(2) 仕様の違い毎に別の libc バージョンを割振る必要がある。
    →対応 libc が無ければランタイムリンカが弾く。

新バイナリを旧 kernel 上で使う場合

- スタティックなバイナリの場合。
  - 新システムコール ID を知らない kernel が弾く。
- ダイナミックなバイナリの場合。
  - xstat() があれば、新 stat(2) 仕様を旧 kernel 用 libc の xstat() が対応出来ずに弾く。
  - xstat() が無ければ、ランタイムリンカが弾けるように、stat(2) 仕様の違い毎に別の libc バージョンを割振る必要がある。

Linux での glibc を考えてみる

- OS 標準 libc なら元々 kernel と一対一じゃん。
  - コンバートは kernel 側でもやってるくれてるよね。
- stat(2) を使ってるつもりのユーザ (プログラマ) から xstat() の存在を隠してるのでは？
  - 便宜を図ってるつもりが混乱を招くかも。
  - stat(2) がライブラリ関数だとは普通気づかない。
- UNIX 的な思想からはちと遠いかも。
  - 単純性とか移植性とか。

(おまけ) マイ libc を作ってみよう

- マイ libc の作り方。
  - コンパイル時に -fPIC オプションを付ける。
  - リンク時に -shared オプションを付ける。
- マイ libc 内でのライブラリ関数の使用には要注意！
  - 無限に再起呼出しが発生してしまわないように。
- ファイル I/O 関数を置換えちゃう応用例。
  - Samba の smbsh(1) 。
  - 複数 NAS を束ねて単一ディレクトリに見せる。

(続き) マイ libc で unlink を置換えよう

- 環境変数 LD_PRELOAD で共有ライブラリを上書き。

```
$ cat unlink.c
#include <stdio.h>
#include <syscall.h>
int unlink(const char *path)
{
        fprintf(stderr, "Unlink %s.¥n", path);
        return syscall(SYS_unlink, path);
}
$ gcc -fPIC -O -c -o unlink.o unlink.c
$ gcc -shared -Wl,-soname,libunlink.so.1 -o libunlink.so unlink.o
$ LD_PRELOAD=`pwd`/libunlink.so rm foo/bar
Unlink foo/bar.
```

-soname は soname を埋込む ld(1) 用オプション。
→ soname は ld.so に名前とバージョンを教える。

- LD_PRELOAD 用には実は不要。

(おまけ 2) 意外なライブラリ関数

- システムコールは非常に基本的な関数なので、ライブラリ関数で置換えることが出来ないことが多い。
- 自分で実装手段を思いつかない関数は、システムコールだと勘違いしてしまうことがままある。
  - opendir(3), readdir(3), closedir(3)
    →open(2) で開いて getdents(2) で読む。
  - getwd(3), getcwd(3)
    →「 .. 」の readdir(3) 結果から i ノードを探す。

(うんちく 1)getdents(2) の補足

- open(2) 時に O_DIRECTORY フラグを付けるとディレクトリも普通に開ける。
- 4.2BSD や SystemV の頃は、ディレクトリファイルの構造がそのまま struct dirent の構造になっていたので、struct dirent 型変数に read(2) 出来た。
- 最近は最適化とかジャーナリングとか色々やってるので、getdents(2) でラップして読む。
- BSD 方面だと getdirentries(2) になる。長い…。

(うんちく 2)getcwd(3) の補足

- まず親を特定しそのまた親を特定し、これを繰返すことで最終的にカレントディレクトリが調べられる。
- そもそも getcwd(3) が辿れるために、わざわざディレクトリエントリ上に「 . 」や「 .. 」が用意されている。
  - 「 ... 」みたいに仮想的な実装も可能だよね。
- 「 / 」は「 . 」と「 .. 」が同じ i ノードを持つ特殊なディレクトリなので判別可。
  - 「 / 」に辿り着くことが繰返しの終了条件。
- 最近の Linux には getcwd(2) がある。

おわりに

- GNU/Linux に於いては glibc の重要性は甚大。
  - glibc は大切に。
  - kernel 同様にバージョン管理に気を配ろう。
- この資料は下記 URL に置いときます。
  http://www.unixusers.net/~shirai/presentation/glibc080719/img0.html
- ご清聴ありがとうございました。