# libcとは何ぞや？

たなか としひさ
tosihisa@netfort.gr.jp

# お題目

- libc とは何ぞや？
- システムコールと標準ライブラリは何が違う？
- ユーザプログラムと、カーネルの分かれ目
- システムコールと思っても、実は違うもの
- Ｃ言語で書いてコンパイルしたプログラムは main() から始まらない。
- crt0.o の役割
- 動的リンク？静的リンク？なにそれ？
- Linux thread 遍歴
- さらば libc
- ようこそ探訪の入り口へ

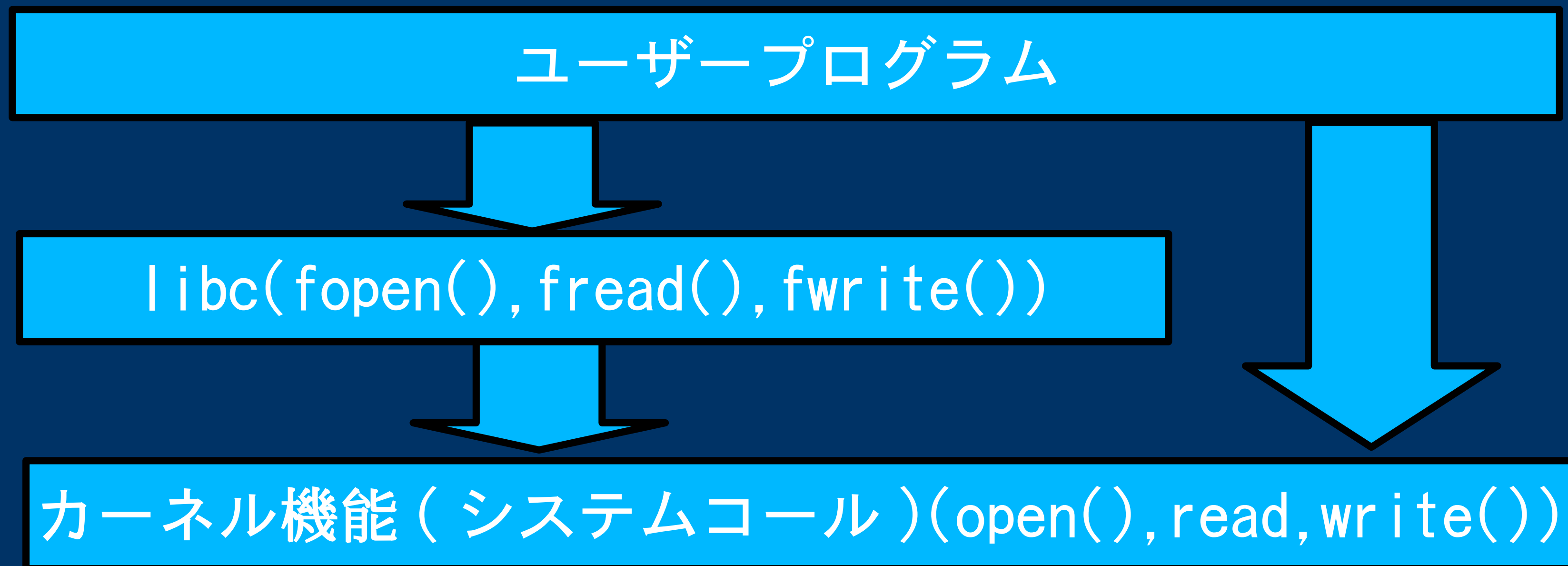
libc と書いてますが、glibc 2.x.x が基本です。

# libcとは何ぞや？

- 単純な話、「よー使われるＣ言語関数」がライブラリとなったものです。
- printf(),fopen(),fread()... とかとか
- 大きな枠組みとしては...
  1. 基本ライブラリ
  2. Ｃスタートアップルーチン(crt0.o)
  3. スレッドライブラリ(pthread)
  4. 国際化対応(locale)
  5. 暗号化(crypt)
  6. ネットワーク関係
  65535. ダイナミックリンクローダ(ld.so)
  ... 等が入っています。
- ちなみに、 open(),read(),write() は、「システムコール」と呼ばれます。

# システムコールと標準ライブラリは何が違う？

- ぶっちゃけた話、 man ｛ほげほげ｝ と入力した時に、 man 2 で出てくるか、 man 3 で出てくるかです。
- システムコールは、それが（ほぼダイレクトに）カーネルの機能を使う機能の事を指します。



# ユーザプログラムと、カーネルの分かれ目切れ目(1)

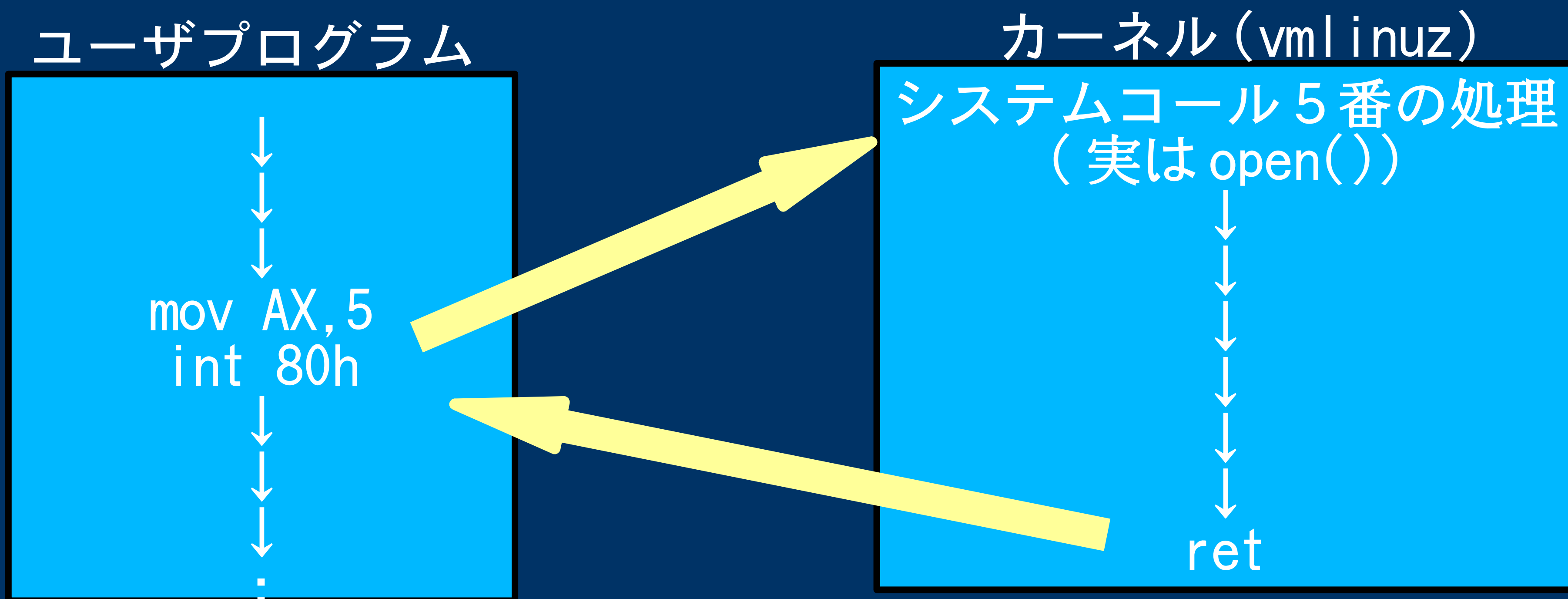
- open(),read(),write() は、そーゆー名前の関数がある訳ではありません。
- x86 系の場合、AX レジスタにシステムコール番号、他レジスタにパラメータを入れて、

  int 80h

  を実行します。ここがまさに、「ユーザプログラムとカーネルの境目」になります。
- int xxh - ソフトウェア割込み命令

# ユーザプログラムと、カーネルの分かれ目切れ目(2)

- 図で書くとこんな感じ



# ユーザプログラムと、カーネルの分かれ目切れ目(3)

- Zaurus 等ARM系のカーネルでは、 int 80h ではなく、

  swi {機能毎のとび先}

  になります。このへんは、 CPU依存な所。
- /usr/include/asm/unistd.hで、 _syscallX(...) とあるのが、インラインアセンブラを使ったシステムコールの呼出し。これがシステムコールの「正体」。

# システムコールと思っても、実はビミョ〜に違うもの(1)

- exit() はシステムコールとちゃいます。
- 動的メモリ管理機構(malloc(),calloc(), realloc(),free())もシステムコールとちゃいます。
  → libc は、内部で mmap() システムコールを使っています。
- shmget(),shmat(),semget()等(IPC)も、厳密にはシステムコールとちゃいます。
  この名のシステムコールは無く、全て ipc() に置き換えられます。
- time() は内部で gettimeofday() を呼んでるだけかも知れませんねぇ。

# システムコールと思っても、実はビミョ〜に違うもの(2)

- socket(),bind(),listen(),accept()も、やっぱりちゃいます。
  →この名のシステムコールは無い場合があり、その場合socketcall() と言うものがあります。
- pthread_hogehoge*() も、システムコールではありません。Linuxには、スレッド関係のシステムコールの中核はclone()です。

- man 2 に出てきても、それが「直で」カーネルにあるとは、期待しても良いけど信じちゃいけない。
  →システムコール番号指定でシステムコールを呼び出して、errno にENOSYS が返った場合、そのシステムコールは未実装です...。

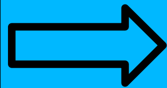
# Ｃ言語で書いてコンパイルしたプログラムは main() から始まらない。

- 始まりません :-)
- プログラムは、「指定した(指定された)番地」から始まります。

# crt0.o の役割 － C RunTime Zero －

- crt0 とは、Ｃ言語プログラムとリンクされ、「一番初めに実行される」処理があります。
- プロセス生成後、main()よりも前に走ります。
- 大体、"_start" とラベルがついてます。
- crt0 は、基本的に、以下の処理をします。
  1. int argc,char *argv[] の設定
  2. char *environ に、環境変数へのポインタを入れます。
  3. _exit(main(argc,argv));

_start:



int main(int argc,char *argv[]){...}

# 動的リンク？（Dynamic-Link）

- 動的リンクは、実行形式ファイルとしては不完全な状態（リンクが十分に出来ていない）。
- 実行時に、メモリ上で不足分をリンクし走る。
- ディスクの節約につながるのと、ライブラリの置き換えが容易。
- gcc と glibc で普通にコンパイルすると、動的リンクになる。
- 動的リンク対応ライブラリ（*.so）のバージョンの違いで、動かない場合がある。

# 静的リンク？(Static-Link)

- 静的リンクは、実行形式ファイルとしては完全な状態(リンクが十分に出来ている)。
- カーネルとマッチングさえすれば、ほぼ動く(正しく動くかは別)。
- ディスク消費量は増えるが、ライブラリ等のバージョンの違いに悩まされずに済む。
- gcc -static オプションでコンパイルすると静的リンクとなる。

# Linux thread 遍歴

- そんなに洗練されてない。正直使いたくない。
  遍歴としては↓に詳しい事が書かれている。

  http://www.linux.or.jp/JM/html/LDP_man-pages/man7/pthreads.7.html

# さらば libc

- libc を＊一切＊使わずに、”Hello World” を出してみる。

# ようこそ探訪の入り口へ

- libc は、ドキュメントが豊富にある様に見えて、その「内部構造」を書いたものは無い(本当に無いのかな...)。
- 「カーネルコメンタリ」みたいな本を出すなら、「libc コメンタリ」を出せ。マジで。
- カーネルをホゲる訳ではないけど、libc を「探訪」するのも、シンドイけど面白い。
- ある意味、「プログラムはなぜ動くか」を理解し、加えて、アセンブラの知識も必要なので、「腕試し」としては丁度良い。
- man は常に最新にしましょう :-)
  JF と JMAN はすごいデス...

# おしまい

- 質問などなど . . .