公開コピー誌
Linux上でのアセンブラ

暗黒通信団

❶ 概論 －アセンブラの利点－

オブジェクト指向の時代にあって、なぜ移植性に乏しいアセンブラなのか？なにが嬉しいのか？ 確かにプログラムのすべてをアセンブラで書くのは現実的ではないし、そもそも最適化はコンパイラの仕事で、OS やコンパイラを書く人でもなければ必須というわけではない。が、部分的な高速化や、CPU の性能にあわせたコードを書くときには大事な知識であろう。Intel 製の 32bitCPU と Linux を使う民間人にとって、アセンブラは次の 4 点において有用だ。

1. C プログラムなどにおける部分的な高速化
2. 80bit レジスタの FPU を過不足なく使う
3. デバグ情報の理解
4. 効率よいアルゴリズム設計

本稿では Linux+IA32 環境におけるアセンブラの基本的な使い方を概説する。なお、詳細は Intel 社が提供する下記 3 点の純正ドキュメントを参照されたい。すべて日本語である。（また、古いものであるが、ASCII 社から出ている PC9801 シリーズ用のゲーム「インサイダーズ」は、この方面の知見を得るのにとても役だった）

- `http://download.intel.com`
  `/jp/developer/jpdoc/ia32.pdf`
- `http://download.intel.com`
  `/jp/developer/jpdoc`
  `/IA32_Arh_Dev_Man_Vol1_Online_i.pdf`
- `http://download.intel.com`
  `/jp/developer/jpdoc/24531704_j.pdf`

❷ 基礎 －コンパイルと書法－

Linux でアセンブラを使うには 3 つのパターンがある。

1. ソフトウェア全体を丸ごとアセンブラだけで書く。
2. 関数だけをアセンブラで書いて、C などの他の関数から呼び出す。
3. C の関数中で gcc のインラインアセンブラとして使う。

❶ 全体をアセンブラで作る場合

この方法は現実的ではないが、基礎的である。この場合、アセンブラのソースファイル *example.asm* を実行形式にするには、プロンプトから次のように入力する。

- コンパイル：`nasm -f elf example.asm`
- リンク　　：`ld -s -o example example.o`

以下は、画面に「世界征服！」と出力するだけの例文である。以下、命令等の意味や詳細は次の節を参照されたい。

exmaple.asm

```
section .text
global _start
msg             db    '世界征服！', 0x0A
msglen          equ  $ - msg

_start:
mov     ecx, msg        ; 文字列のアドレス
mov     edx, msglen     ; 文字列長
mov     eax, 4          ; 出力のシステムコール
mov     ebx, 1          ; 標準出力を指定
int     0x80            ; システムコール実行
mov     eax, 1          ; 終了のシステムコール
mov     ebx, 0          ; 正常終了
int     0x80            ; システムコール実行
```

コメントは「;」から行末までである。「`section .text`」と「`global _start`」、「`start:`」は決まり文句である。「`start`」は、実行ファイルのエントリになる。「`db`」は byte 型 (8bit) のデータであることを示す。「0x0A」は十進数の 10 であり、UNIX の改行コードである。「`msglen equ $ - msg`」は、現在のファイル位置`$`から `msg` ラベルのファイル位置までの差を `msglen` として定義している。これはすなわち `msg` 文字列の長さである。

正常終了する部分は、すべてのプログラムに共通なので、サブルーチンにまとめておくことができる。`Exit` サブルーチンは次のように書く。`Exit` を呼び出すには、「`call Exit`」と書けばよい。

stdio.inc

```
Exit:
mov     eax, 1
mov     ebx, 0
int     0x80
```

全体をアセンブラで書くことの（数少ない）メリットは、ファイルサイズを小さくするレースに参戦できる点である。この方向性から考えると、`-f elf` オプションでまっとうな実行ファイルの生成をするのは、手ぬるいかもしれない。そこで、リンカ（`ld`）を通さないで、Linux の実行形式のファイル (ELF 形式) を、直接書いてしまうこともできる。ELF 形式についてはマニュアル (`man elf`) が参考になる。もちろんフォーマット通りに完全に書くとファイルサイズが大きくなるので、最低限必要な部分だけを書く。イリーガルな方法なので、例は載せない。ELF 形式について kernel がどう認識しているかを調べるには、`readelf -l` コマンドが役立つ。なお、ELF のヘッダは次のような構造になっている。

オフセット	単位	繰り返し	説明
0x0000	1	4	固定バイト列。0x7f,E,L,F。
0x0004	1	1	32bit 用 (1) か 64bit 用 (2) か。
0x0005	1	1	エンディアン。リトルエンディアンなら 1、ビッグなら 2。
0x0006	1	1	ELF のバージョン。通常 1。
0x0007	1	1	ターゲット OS と ABI。0 で構わない。
0x0008	1	1	ABI のバージョン。0 で構わない。
0x0009	1	1	パディング開始位置。実質固定バイト列。0。
0x000A	1	6	大抵は 0。
0x0010	2	1	実行可能かどうか。実行可能なら 2。
0x0012	2	1	アーキテクチャ。i386 なら 3。
0x0014	4	1	ファイルバージョン。固定バイト列 1。
0x0018	4	1	実行開始仮想位置。
0x001C	4	1	プログラムヘッダテーブルの位置。
0x0020	4	1	セクションヘッダテーブルの位置。
0x0024	1	4	固定バイト列。0, 0, 0, 0。
0x0028	2	1	ELF ヘッダサイズ。0x34。
0x002A	2	1	プログラムヘッダテーブルの 1 エントリあたりのサイズ。0x20。
0x002C	2	1	プログラムヘッダテーブルのエントリ数。
0x002E	2	1	セクションヘッダテーブルの 1 エントリあたりのサイズ。0x20。
0x0030	2	1	セクションヘッダテーブルのエントリ数。
0x0032	2	1	セクション名文字列テーブルに関連付けられたエントリのセクションヘッダテーブルインデックス。

❷ 関数をアセンブラで作る場合

これは最も現実的な方法であろう。この場合も、上記と同様に NASM で関数をコンパイルしてから、他の言語で作ったオブジェクトと一緒にリンクすればよい。関数の呼び出しにおいては、次の点に留意する。

- 引数はすべて 32bit として、右から順にスタックに push して呼び出し先関数に渡す。処理が戻ってきてから、esp をスタックの移動バイト数だけ add する。
- 呼び出し先は、戻り値を eax レジスタに入れて返す。浮動小数点の戻り値は FPU レジスタの st0 で返す。32bit より大きな場合（構造体など）では、そのアドレスを eax レジスタに入れて返す。
- 関数は ecx、edx を破壊しても構わない。ret で呼び出し元に制御を戻す。

以下はアセンブラによる関数を C 言語から呼び出す例である。この関数は char 型 2 つを引数にとるが、これは中では 32bit に拡大されて渡されるので、int 型でもよい。

asmadd.c

```
#include <stdio.h>

int asmadd(char, char);
int main(void){
int re=0;
re=asmadd(123,45);
printf("%d\n",re);
return(0);
}
```

asmadd.asm

```
section .text
global asmadd
asmadd:
mov    eax, [esp+4]    ; 第一引数を eax へ
add    eax, [esp+8]    ; 第二引数と加算
; ちなみに [esp] には呼び出し元のアドレスが入っている
ret
```

逆に、アセンブラから C 言語の標準ライブラリ関数を呼び出すこともできる。アセンブラのデバグにはよく使う。スタック操作はすべて呼び出し元が行うので、1 回 push したことによるスタック変化を戻すために、関数コールのあと、「add esp, 4」を実行しなくてはいけない。

call.asm

```
section .data
MOJIRETU db "世界征服！", 0

section .text
extern puts    ;puts は外部関数だと宣言しておく
global _start  ;_start は外部に開かれた関数だと宣言
_start:

push MOJIRETU  ; 文字列のアドレスを push
call puts      ; 関数コール
add esp, 4     ; スタック位置を戻す
mov    eax, 1  ;  終了のシステムコール
mov    ebx, 0  ;  正常終了の 0 に設定
int    0x80    ;   システムコール実行
```

これを実行ファイルにするためには、次のように、DLL ローダ (/lib/ld-linux.so.2) を直に示す必要がある。DLL ローダは必要な共有ライブラリを順次探し出してロードするライブラリである。

call.sh

```
#!/bin/sh
nasm -f elf call.asm
ld --dynamic-linker /lib/ld-linux.so.2 -lc call.o
```

printf のような可変数引数関数の場合はどうするか。実は引数の数より多めのスタックを取ってデータを渡すだけであり、引数の個数などは渡されない。しかし下記のように、「puts」を「printf」にするだけではうまくいかない。

printferr.asm

```
section .data
MOJIRETU db "世界征服！", 0

section .text
extern printf  ;printf は外部関数だと宣言しておく
```

```
global _start   ;_start は外部に開かれた関数だと宣言
_start:

push MOJIRETU   ; 文字列のアドレスを push
call printf     ; 関数コール
add esp, 4      ; スタック位置を戻す
mov    eax, 1   ; 終了のシステムコール
mov    ebx, 0   ; 正常終了の 0 に設定
int    0x80     ;  システムコール実行
```

なぜうまくいかないのかというと、printf は出力がバッファリングされているためである。まだバッファがフラッシュされていないうちに終了コールが呼ばれるので、何も出力されない。うまく動かすためには、上記ソースの 2 行目、「MOJIRETU db "世界征服！", 0」に改行コード (10) を入れ、「MOJIRETU db "世界征服！", 10, 0」とするとよい。

C 言語に近いアセンブラの書き方を使う場合は、global _start よりも、global main として main 関数を書き、それを gcc でリンクした方が安全である。以下はその例である。

printfmain.asm

```
section .data
msg     db   "世界征服！", 10, 0

section .text
       global  main
       extern  printf
main:
       push    msg
       call    printf
       add     esp, 4
       mov     eax, 0
       ret
```

コンパイルは次のようにやる。

printfmain.sh

```
#!/bin/sh
nasm -f elf printfmain.asm
gcc printfmain.o
```

printf が使えるようになると、FPU レジスタ経由で long double 型 (80bit) のデータを受け渡すことを試験できる。FPU は 486DX より上位の CPU がもつ浮動小数点演算専用の回路であり、有効数字 18 桁で 10 の± 4931 乗の範囲の数値の演算ができる。浮動小数点演算の実行は CPU レジスタとは全く異なる浮動小数点演算専用レジスタに数値を設定した後、演算命令を実行する。

FPU のレジスタは st0、st1、... st7 のように 8 本がスタックを構成している。スタックポインタに相当する スタックトップ (st0) は fsw(Status Word) レジスタの 11〜13 ビットが指している。

longdouble.asm

```
section .data
Number46 dt  46.12345678901234567
MOJIRETU db "%17.17llf",10,0
section .text
extern printf  ;printf は外部の関数
global _start  ;_start は外部に開かれた関数
_start:
finit; FPU を初期化
fld  tword [Number46];10 バイト定数を st0 に push
sub  esp, 4; スタックを移動
fstp tword [esp]; st0 から [esp] へ 10 バイトを pop
push MOJIRETU;
call printf     ; 関数コール
add  esp, 8     ; スタック位置を戻す

mov  eax, 1; いつもの
mov  ebx, 0; 終了の
int  0x80   ; おまじない
```

最後に余談になるが、実は printf に直に文字列を渡すのはあまりよくない。万一漢字列の中に % が入ってたら、誤動作する可能性があるからだ。よって、C 言語においてもいちいち「printf("%s","世界征服！");」と書くべきである。

❸ gcc のインラインアセンブラ

アセンブラ部分を関数にすることの問題点は、関数呼び出しにかかるオーバーヘッドである。これを回避するために、一つの関数の中で C 言語とアセンブラを混在させる方法がある。それが「インラインアセンブラ」である。ただしこの方法には「最適化対策」という困難な問題がつきまとう。高速化を目的とする以上、C コンパイラによる最適化は必須だが、大抵はコンパイラがレジスタ割り当てを勝手に変更してしまい、アセンブラの部分と C の部分で不整合を起こして、まずまともには動かない。従って、現実的にはインラインアセンブラを使うためには、レジスタの割付についての情報をコンパイラに与えることが（ほぼ）必須となる。これが「拡張アセンブラ構文」である。

コンパイルは、普通に gcc を通すだけでよいが、内部で GNU のアセンブラ (as) を呼び出すので、アセンブラ自体の書き方が異なる。上記 (Intel 記法) と違い、AT&T 記法という記法を使わねばならない。

- AT&T 記法は「命令 元のレジスタ 処理先のレジスタ」の順になる。これは NASM などの Intel 記法と逆である。
- レジスタ名の表記は先頭に % をつける。eax なら %eax。インラインアセンブラでは %%eax のように「%」が 2 つつく。
- 定数は数値の前に$をつける。
- 命令の最後に b、w、l のどれかをつけて、扱うメモリのサイズを指定する。

- Intel 記法ではレジスタの指すメモリは [ ] で囲むが、AT&T 記法では ( ) で囲む。
- 間接参照の構文は、offset(base,index,scale) となる。

インラインアセンブラでは、アセンブラの各命令行を `asm();` で囲って C ソース中に埋め込む。囲み方には以下の 4 つの方法がある。コンパイラによる最適化の影響を回避するためには、極力アセンブラ部分をブロック化するべきであり、最後の記法が望ましい。

1.

bit 数	データ型	
8bit(char)	db	byte
16bit(short)	dw	word
32bit(int, float)	dd	dword
64bit(double, long long)	dq	qword
80bit(FPU, long double)	dt	tword

❷ レジスタ

一般的に、CPU はレジスタのデータを読み込み、演算をする。計算結果もレジスタ経由で与えられる。レジスタには次のような種類がある。

- 主レジスタ (自由に使える)：
  eax/ax/ah/al、
  実際には 32bit の eax が 1 本存在するだけである。その下位 16bit を特に指すのが ax であり、ax の中の下位 8bit が al、上位 8bit が ah である。これらは計算機が 8bit, 16bit だった時代の名残である。以下、ecx, edx も同じである。
  bax/bx/bh/bl、
  ecx/cx/ch/cl、
  edx/dx/dh/dl
- ポインタレジスタ：
  esi/si（自由に使える)、
  edi/di（自由に使える)、
  ebp/bp（引数の位置を表す；ここまでは pusha で push される)、
  esp/sp（スタックの位置を示す)、
  eip（プログラムの実行位置を示す）
- セグメントレジスタ（普通いじらない)：
  cs(コード領域のセグメントを示す)、
  ds(データ領域のセグメントを示す)、
  ss(スタック領域のセグメントを示す)、
  es(自由に使える)、
  fs(自由に使える)、
  gs(自由に使える)
- フラグレジスタ：eflag
  事実上書き込み禁止。各ビットが様々なフラグに対応している。下位から順に次のような意味を持つ。
  00 cf(キャリーフラグ；桁あふれで 1 に)
  01 常に 1
  02 pf(パリティフラグ；参照レジスタのビット数が奇数なら 1)
  03 常に 0
  04 af(予備フラグ)
  05 常に 0
  06 zf(ゼロフラグ；演算結果が 0 なら 1)
  07 sf(符号フラグ)
  08 tf
  09 if(割り込みフラグ；1 なら割り込み許可)
  10 df(方向フラグ；アドレス参照方向の指定)
  11 of(オーバーフローフラグ)
  12 iopl
  13 iopl
  14 nt
  15 常に 0
  16 rf
  17 vm
  18 ac
  19 vif
  20 vip
  21 id
  22 これ以降は常に 0
- 浮動小数点レジスタ（浮動小数点の計算用)：
  st0、st1....st7：数値レジスタスタック
  fcw、fsw、ftw：コントロールレジスタ
  fop、fcs、fip：命令ポインタ
  fds、fea：データポインタ
  実際には st0〜st7 しか使わない。CPU のレジスタ (eax など) と違い、st レジスタはスタックになっている。
- MMX レジスタ群（浮動小数点のレジスタを違う目的で利用する仕組み。実際には FPU レジスタなので、FPU 命令と混在できない。割愛）
- SSE, SSE2, SSE3 レジスタ群（明らかに基礎レベルを越えてるので割愛）

❸ よく使われるニーモニック

- **汎用命令**
  - ▷ mov：**値の代入**
  - ▷ xchg：**値の交換**
  - ▷ push：**スタックへデータを転送**
  - ▷ pop：**スタックからデータをもってくる**
  - ▷ pusha：push eax ebx ecx edx esi edi ebp **と同じ**
  - ▷ popa：pusha **の逆。**
- **演算**
  - ▷ add：**加算**
  - ▷ sub：**減算**
  - ▷ mul：**整数のかけ算**
  - ▷ div：**除算**
  - ▷ inc：1 **増やす**
  - ▷ dec：1 **減らす**
  - ▷ neg：**符号反転**
  - ▷ cmp：**比較**
- **論理演算**
  - ▷ and：**論理積**
  - ▷ or：**論理和**
  - ▷ xor：**排他的論理和**

- ▷ not：否定
- ▷ test：フラグを立てるだけの and

- シフト
  - ▷ shr：ビットの右シフト
  - ▷ shl：ビットの左シフト
  - ▷ shrd：ビットのダブル右シフト
  - ▷ shld：ビットのダブル左シフト
  - ▷ ror：ビットの右回転
  - ▷ rol：ビットの左回転
- 実行制御
  - ▷ jmp：実行位置の変更
  - ▷ je：cmp A,B で値が同じならジャンプ
  - ▷ jz：cmp A,B で値が同じならジャンプ
  - ▷ jne：cmp A,B で値が違えばジャンプ
  - ▷ jnz：cmp A,B で値が違えばジャンプ
  - ▷ ja：cmp A,B で A が大きければジャンプ
  - ▷ jae：cmp A,B で A が大きいか等しければジャンプ
  - ▷ jl：cmp A,B で A が小さければジャンプ
  - ▷ jle：cmp A,B で A が小さいか等しければジャンプ
  - ▷ loop：ecx をカウンタとしてループ。
  - ▷ call：サブルーチンへジャンプ
  - ▷ ret：サブルーチンから戻る
  - ▷ int：割り込み
- その他の一般命令
  - ▷ lea：動的変数のアドレスを得る
  - ▷ nop：何もしない。バイナリ改変で使う
- FPU ロード命令
  - ▷ fld：st0 へデータをロード
  - ▷ fld1：st0 へ $+1.0$ をロード
  - ▷ fldz：st0 へ $+0.0$ をロード
  - ▷ fldpi：st0 へ π をロード
  - ▷ fldl2e：st0 へ $\log_2 e$ をロード
  - ▷ fldln2：st0 へ $\log_e 2$ をロード
  - ▷ fldl2t：st0 へ $\log_2 10$ をロード
  - ▷ fldlg2：st0 へ $\log_{10} 2$ をロード
- FPU 演算命令
  - ▷ fadd：浮動小数点の可算
  - ▷ faddp：浮動小数点の可算と pop
  - ▷ fsub：浮動小数点の減算
  - ▷ fsubp：浮動小数点の減算と pop
  - ▷ fmul：浮動小数点の掛け算
  - ▷ fmulp：浮動小数点の掛け算と pop
  - ▷ fdiv：浮動小数点の割り算
  - ▷ fdivp：浮動小数点の割り算と pop
  - ▷ fabs：浮動小数点の絶対値
  - ▷ fchs：浮動小数点の符号を反転
  - ▷ fsqrt：浮動小数点の平方根
  - ▷ fxtract：浮動小数点の指数部と仮数部を得る
- FPU 比較命令
  - ▷ fcom：浮動小数点の比較
  - ▷ fcomp：浮動小数点の比較と pop
  - ▷ ftst：浮動小数点の test
  - ▷ fincstp：FPU レジスタのスタックポインタを inc
  - ▷ fdecstp：FPU レジスタのスタックポインタを dec
  - ▷ finit：エラー条件をチェックして FPU 初期化
  - ▷ fninit：エラー条件をチェックせずに FPU 初期化
- FPU 関数命令
  - ▷ fsin：正弦
  - ▷ fcos：余弦
  - ▷ fsincos：正弦と余弦
  - ▷ fptan：部分正接
  - ▷ fpatan：部分逆正接
  - ▷ f2xm1：$2^x - 1$
  - ▷ fyl2x：$y \log_2 x$
  - ▷ fyl2xp1：$y \log_2(x+1)$

❹ システムコール

Linux では、レジスタに機能番号や引数をセットしてから「int 0x80」を実行することで kernel の提供する機能を使う。これをシステムコールと呼ぶ。引数の個数は 0〜5 個であり、使用するレジスタは以下の通りである。

eax ..... システムコール番号を設定
ebx ..... システムコールの第 1 引数
ecx ..... システムコールの第 2 引数
edx ..... システムコールの第 3 引数
esi ..... システムコールの第 4 引数
edi ..... システムコールの第 5 引数

もし、もっとたくさんの引数を渡す必要があるときは、メモリに引数を入れて引数リストの先頭アドレスを ebx にいれてシステムコールを呼ぶ。

実際にどのような種類のシステムコールが提供されているかは、カーネルソースを見るのがよい。以下は、include/asm-i386/unistd.h から作った変換テーブルである。%assign は、別名を振るマクロである。kernel 2.6.18 では 317 番まで定義されているが、長々書いても仕方ないので 128 番まで載せた (深い意味はない)。システムコールを使うコードの先頭付近で「%include "systemcall.asm"」と書いて、このファイルを挿入する。

systemcall.asm

```
%assign     sys_exit                    1
%assign     sys_fork                    2
%assign     sys_read                    3
```

```
%assign    sys_write                4
%assign    sys_open                 5
%assign    sys_close                6
%assign    sys_waitpid              7
%assign    sys_creat                8
%assign    sys_link                 9
%assign    sys_unlink              10
%assign    sys_execve              11
%assign    sys_chdir               12
%assign    sys_time                13
%assign    sys_mknod               14
%assign    sys_chmod               15
%assign    sys_lchown              16
%assign    sys_break               17
%assign    sys_oldstat             18
%assign    sys_lseek               19
%assign    sys_getpid              20
%assign    sys_mount               21
%assign    sys_umount              22
%assign    sys_setuid              23
%assign    sys_getuid              24
%assign    sys_stime               25
%assign    sys_ptrace              26
%assign    sys_alarm               27
%assign    sys_oldfstat            28
%assign    sys_pause               29
%assign    sys_utime               30
%assign    sys_stty                31
%assign    sys_gtty                32
%assign    sys_access              33
%assign    sys_nice                34
%assign    sys_ftime               35
%assign    sys_sync                36
%assign    sys_kill                37
%assign    sys_rename              38
%assign    sys_mkdir               39
%assign    sys_rmdir               40
%assign    sys_dup                 41
%assign    sys_pipe                42
%assign    sys_times               43
%assign    sys_prof                44
%assign    sys_brk                 45
%assign    sys_setgid              46
%assign    sys_getgid              47
%assign    sys_signal              48
%assign    sys_geteuid             49
%assign    sys_getegid             50
%assign    sys_acct                51
%assign    sys_umount2             52
%assign    sys_lock                53
%assign    sys_ioctl               54
%assign    sys_fcntl               55
%assign    sys_mpx                 56
%assign    sys_setpgid             57
%assign    sys_ulimit              58
%assign    sys_oldolduname         59
%assign    sys_umask               60
%assign    sys_chroot              61
%assign    sys_ustat               62
%assign    sys_dup2                63
%assign    sys_getppid             64
%assign    sys_getpgrp             65
%assign    sys_setsid              66
%assign    sys_sigaction           67
%assign    sys_sgetmask            68
%assign    sys_ssetmask            69
%assign    sys_setreuid            70
%assign    sys_setregid            71
%assign    sys_sigsuspend          72
%assign    sys_sigpending          73
%assign    sys_sethostname         74
%assign    sys_setrlimit           75
%assign    sys_getrlimit           76
%assign    sys_getrusage           77
%assign    sys_gettimeofday        78
%assign    sys_settimeofday        79
%assign    sys_getgroups           80
%assign    sys_setgroups           81
%assign    sys_select              82
%assign    sys_symlink             83
%assign    sys_oldlstat            84
%assign    sys_readlink            85
%assign    sys_uselib              86
%assign    sys_swapon              87
%assign    sys_reboot              88
%assign    sys_readdir             89
%assign    sys_mmap                90
%assign    sys_munmap              91
%assign    sys_truncate            92
%assign    sys_ftruncate           93
%assign    sys_fchmod              94
%assign    sys_fchown              95
%assign    sys_getpriority         96
%assign    sys_setpriority         97
%assign    sys_profil              98
%assign    sys_statfs              99
%assign    sys_fstatfs            100
%assign    sys_ioperm             101
%assign    sys_socketcall         102
%assign    sys_syslog             103
%assign    sys_setitimer          104
%assign    sys_getitimer          105
%assign    sys_stat               106
%assign    sys_lstat              107
%assign    sys_fstat              108
%assign    sys_olduname           109
%assign    sys_iopl               110
%assign    sys_vhangup            111
%assign    sys_idle               112
%assign    sys_vm86old            113
%assign    sys_wait4              114
%assign    sys_swapoff            115
```

```
%assign    sys_sysinfo             116
%assign    sys_ipc                 117
%assign    sys_fsync               118
%assign    sys_sigreturn           119
%assign    sys_clone               120
%assign    sys_setdomainname       121
%assign    sys_uname               122
%assign    sys_modify_ldt          123
%assign    sys_adjtimex            124
%assign    sys_mprotect            125
%assign    sys_sigprocmask         126
%assign    sys_create_module       127
%assign    sys_init_module         128
```

❹ その他の補足的知識

❶ コマンドライン引数

スタックは、アドレスの値が小さくなるほど新しく入れられたデータになる。つまり、push すると、esp の値は小さくなり、pop すると大きくなる。

プログラムが起動された瞬間のスタックは次のようにデータが入っている。なお、ワイルドカードはシェルによって実際のプログラム名に展開される。

- esp が示す位置→引数の数 (argc)
- esp+4 が示す位置→プログラムのファイル名
- esp+8 が示す位置→第一引数のデータが入っているアドレス。

  |
- esp+4*argc が示す位置→ 0
- esp+4*argc+4 が示す位置→一番目の環境変数が入っているアドレス。
- esp+4*argc+8 が示す位置→二番目の環境変数が入っているアドレス。

  |

次の例は、実行ファイル名と第一引数の文字列を画面に出す。

printargv.asm

```
section .text
extern puts
global _start
_start:
add esp, 4
call puts
add esp, 4
call puts
sub esp, 8

mov    eax, 1
mov    ebx, 0
int    0x80
```

printargv.sh

```
#!/bin/sh

nasm -f elf printargv.asm
ld --dynamic-linker /lib/ld-linux.so.2
                        -lc printargv.o
./a.out ARGVDA!
```

❷ 任意のファイルの読み書き

ただのファイルを扱うならたいした意味はないが、ネットワークソケットや proc ファイルを高速で扱う場合にアセンブラが有効な場合がある。

ファイルの読み書きは sys_read と sys_write を使う。ファイルハンドル番号は sys_open で eax に返ってきた値を使う。使い終わったファイルは sys_close でクローズする。

ファイルを open するときに、様々な属性を加える。以下は 8 進数の属性表で、これらの or をとって属性値を決める。

フラグ名	設定値	意味
O_ACCMODE	0003	アクセスモードのマスク
O_RDONLY	00	読み込み専用
O_WRONLY	01	書き込み専用
O_RDWR	02	読み書き可能
O_CREAT	0100	ファイルを作成
O_EXCL	0200	排他的にオープン
O_NOCTTY	0400	ターミナルを割り当てない
O_TRUNC	01000	切捨てオープン
O_APPEND	02000	アペンド
O_NONBLOCK	04000	ノンブロッキング I/O
O_NDELAY	O_NONBLOCK	上に同じ
O_SYNC	010000	ライトスルーキャッシング
FASYNC	020000	非同期通知
O_DIRECT	040000	ダイレクトアクセス
O_LARGEFILE	0100000	2GB 以上のファイルを扱う
O_DIRECTORY	0200000	ディレクトリ
O_NOFOLLOW	0400000	シンボリックリンクでない
O_NOATIME	01000000	アクセス時刻を更新しない

以下の例は、システムコールのみを使って、*outputsample.txt* に「世界征服！」と書くものである。

write.asm

```
section .data
FILENAME db "outputsample.txt",0
OUTPUT db "世界征服！",10,0

section .text
```

```
global _start
_start:

; ファイルを作る
mov eax,5; sys_open
mov ebx,FILENAME
mov ecx,65; flag;create+write
mov edx,64*6+8*4+4; パーミッション
int 0x80

mov ebx, eax;ebx にハンドル番号

; 書き込み
mov eax, 4; sys_write
mov ecx, OUTPUT
mov edx, 11; 書き込みバイト数
int 0x80

; クローズ
mov eax, 6; sys_close
int 0x80

; 終了
mov eax,1
mov ebx,0
int 0x80
```

❺ 謝辞

本稿を書くにあたり、暗黒通信団メーリングリストの TAKESAKO 氏、木村氏、いそのん師匠の助言を参考にした。また、Intel 社の資料を多く参考にした。お礼申し上げる。

修正連絡等は、 srl@shiros.net までお願いしたい。

❻ 改変履歴

- 2007.12.31 コピー版頒布。
- 2008.1.15 環境依存する部分を削除して Web 配布開始。
- 2009.2.19 PDF 版配布開始