コンピュータ・アーキテクチャ II
2020 年 10 月 9 日
「ライブラリ (Libraries)」と「システムコール (System Calls)」

ライブラリとシステムコールの違い

さて上で述べた「リンク」の話のところで出てきた「ライブラリ」の話を少ししましょう。プログラムを

作成する場合、必ず自分では書かない (書く必要のない) 部分が存在します。C や Java ではこうした、システムで用意してくれていて、自分でプログラムしなくてもよい部分はライブラリと呼ばれていました。

ではライブラリの中はどうなっているか考えたことはありますか?C や Java ではライブラリはまったくの「ブラックボックス (Black Box)」ではありません。

ちなみに、ブラックボックスとは内部を理解していなくても、その使い方さえ知っていれば十分な装置や機構のことで、ソースコードが公開されていないために内で何をやっているかわからない関数などをこのように呼びます。

C や Java では基本的なライブラリのソースコードはすべて公開されています。それをいちいちコンパイルしてから使用するのでは面倒なので、予め大きな単位ごとに纏めてコンパイルして、リンクをすればよいよう準備してあるのがライブラリです。

linux 上の C では「.a(static link の場合)」か「.so(dynamic link の場合)」という文字が最後についたファイルがライブラリです。また、Java では最後に「.jar」がついているものがそうです。ちなみに、Java ではこのファイルは zip という形式で圧縮されています。

これらのライブラリのソースコードはもちろん公開され、配布されているのですが、普通は自分のプログラムを書くのに忙しいためにライブラリの中身には興味を示さないので、最初からインストールしようとしないようです。

「**システムコール**」と呼ばれているものも見掛け上はライブラリと全く同じように呼び出します。また同じようなことが起こるので、なにも知識がなければ両者の区別をつけることはできないでしょう。

実際、Windows ではシステムコールもシステムで用意されているライブラリも差をつけずに、全て **API (Application Programming Interface)** という名前で呼んでいて区別がありません。

初心者がプログラミングをする時にはシステムコールとライブラリコールを区別しなくても問題は生じません。しかし実際にはファイルの読み書きやネットワーク越しの通信などを行う場合には、プログラマ本人はライブラリを呼び出したつもりでも、そのライブラリの関数が別なライブラリを呼び出し、またその呼び出された関数が別な、とずっと辿っていった先の先の下層では、OS の機能の一部を呼び出すために、必ずシステムコールが実行されています。

スタティックリンクライブラリ (static link library) というのは、実行する前にプログラムとリンクして保存する形式のライブラリのことです。昔はこれが普通でした。

ダイナミックリンクライブラリ (dynamic link library) というのは、実行を始めてからスタブ (stub) と呼ばれる仮の呼び出し口を経由して、OS にお願いして用意してあったライブラリをその場で結合して実行する形のライブラリです。

前者ではプログラムサイズが大きくなりますが、リンク時にすべての関数の参照が解決されているかチェックされるために、リンクでエラーが出なければ、同じ OS ならばどこへ持っていってもエラーを出さずに実行できます。ただし後述するシステムコールの動作が異なれば、当然ながらトラブルが発生します。

後者では OS 上にライブラリはたった一つだけしか存在せずに、全てのプロセスは同じライブラリを参照するので、メモリはあまり使わずに済むのですが、ライブラリのバージョンが合っているかどうか、プログラムを移した先で注意しないとプログラムは動作しなかったり奇妙な動作をしたりします。

セキュリティに関しては、公知のバグのある昔のライブラリを組み込んでしまったスタティックリンクライブラリよりも、後付でバグのないライブラリをいつでも採用することのできるため、ダイナミックリンクライブラリの方がよいとされています。

なお、C 言語でコンパイルするときに、cc -Bstatic というオプションをつけるとスタティックリンクライブラリが、特別に何もつけなければダイナミックリンクライブラリが用いられます。両者のサイズを比べてみると面白いと思います。

ファイル形式を見るためのプログラムとして file コマンドがあります。例えば

```
% file a.out
```

というコマンドを実行すると、'elf format' という文字が見つかると思います。これは ELF 形式という比較的新しいバイナリフォーマットであることを示しています。

elf 形式というのは、ダイナミックリンクを積極的にサポートするために導入されたフォーマットで、Linux など多くの unix 系列の OS は、現在この形式を主に用いていて、標準でダイナミックリンクされたバイナリを出力するようになっています。

ちなみに、dwarf 形式というフォーマットも存在していて、これはデバッグのための形式です。unix のライブラリの大きな特色は、ほぼ全てのコードが PIC(Positon Independent Code) と呼ばれるアドレスに依存しないコードで実現されていて、メモリ上のどの番地に置かれても構わないようになっています。

Windows においては大分事情が変わります。Windows では、拡張子 (extension) と呼ばれる最後の 4 文字が「.exe」となっているものが実行ファイルです。これらはライブラリとして、拡張子が「.lib」というスタティックリンクライブラリや「.dll」となっているダイナミックリンクライブラリを呼び出しながら実行されます。

dll 形式のライブラリはこれらを参照するプログラムがすべて消滅するとメモリから消えるはずなのですが、dll 側が一つのサービスプログラムとしての機能を持って動くために、メモリから取り去ることのできないものがあります。また、unix の PIC の複雑さを嫌ってか、特定の dll はある特定の番地でしか動かすことができない、といったものがあるそうです (伝聞ですが)。

玉ねぎの皮; どこまで剥けばシステムコールが現れるのか

ではライブラリをどこまで読んでいけばシステムコールは現れるのでしょう。FreeBSD のソースコード　`https://www.tundraware.com/Technology/Docs/FreeBSD-Source/lib/libc/stdio/` の中から fputs() 関数のソースコードを覗いてみると、

```
int fputs(s, fp) const char *s;   FILE *fp;
  {
        int retval;
        struct __suio uio;
        struct __siov iov;
... 中略...
        FLOCKFILE(fp);
        retval = __sfvwrite(fp, &uio);
        FUNLOCKFILE(fp);
        return (retval);
```

```
}
```

のようになっていて、この関数からは`__sfvwrite()`という関数が呼び出されていて、これが下請け処理をしているらしいことが判ります。

さらにその先は、とどんどん先を辿っていくと、ライブラリのソースコード集の中にはソースコードが出てこない関数にたどり着きます。今の例だと`_write()` という関数が相当します。これが物理や化学の比喩を使えば、「ある物質をどんどん細かく割っていって最後にどうしても分割できないものに到達」したことになります。これが「原子 (atom)」ではなくて、「システムコール (System Calls)」と呼ばれるものです。

プロセスから見るとシステムコールはプロセッサの命令と同様に分割できない単独の命令のように見えていて、この命令 (システムコール) を実行した後は時間的な遅れが全くなしに、すぐに要求が叶えられているように見えます (それでも実は時計が進んでいます。まるで眠り姫のように)。

OS から見ると、システムコールはプロセスから OS を呼び出す、ある種のサブルーチンコールのように見えます。実行が終了するまでは、呼び出したプロセスは氷付けになって待っているように見えるのです。

`_write()` のソースコードは実はライブラリの場所ではなく OS のシステムコールの部分にちゃんと存在しています。ですが、ここではサブルーチン呼び出しの知識以外に非常に大量の知識が必要になります。この部分をいきなり読んでもとうてい理解できません。その中で何をやっているのかを理解するために、この授業があるのです。

システムコールの実際と問題点

unix(含 Linux) では、システムコールは OS の機能呼び出しで、プロセス単体では処理しきれない「ファイル出力をお願い (fputs() の場合はこれ)→write システムコール」とか「入力を下さい →read システムコール」とか「メモリをもっと下さい →sbrk システムコール」といった要求に答えるためにあります。これが前の項で説明した「コンピュータの機能拡張

のための OS」であり、「コンピュータを使いやすくするため」の OS の側面です。

システムコールそのものはトラップ (trap) 命令やソフトウェア割り込み (software interrupt) 命令の形で実装する場合が多いようです。これを各種の言語から呼び出す場合、unix ではライブラリ関数を用意して、ちょうどお饅頭のように餡であるシステムコールを皮のようにライブラリで包んで呼び出すことになります。こうした枠組みを (Windows の用語を使って)、API(Application Programming Interface) ないしは ABI(Application Binary Interface) と呼びます。

では、本当にこうした機能はライブラリという形では提供できないのでしょうか? あるいは、より抽象化された puts() や printf() というレベルで OS が機能を提供することはできないのでしょうか? 実はどちらも可能ではあります。しかしこれを行っていないのはそれなりの理由があります。

ブルックスの「人月の神話—狼人間を撃つ銀の弾はない」http://www.amazon.co.jp/exec/obidos/ASIN/4795296758/249-2605332-0792341 という本があるのですが、この本について文献 1 は次のように述べています。

「OS/360 の設計者の一人である Fred Brooks は後に、OS/360 のそのときの経験を記述したウィットの効いた辛辣な本を書いた。ここでその本の要旨を紹介することは不可能であるが、「有史以前の動物の群れがコールタールの穴にはまり込んでしまっている」表紙を見ればそれで十分だろう。」

さらにこの部分の前から引用します。
「結局 (OS/360 は) 巨大で途方もなく複雑なオペレーティング・システムになった。おそらく、FMS(前の世代の OS) よりも 2 桁から 3 桁大きいシステムだったろう。数千人のプログラマが、数百万行にもわたるアセンブリ言語で記述していた。無数のバグを抱え、これを修正するために継続的に新しいリリースを行っていく必要があった。それぞれのリリースでは、いくつかのバグは修正されるが、新たなバグも複数組み込んでしまう。従って将来は一定の個数のバ

グが残るだろう。(訳しなおしてみると、最後は正しくは「新しいリリースによって、あるバグは修正されるが別なバグが紛れ込み、全体のバグの個数は概ね一定だった。」)」

実は当時の OS はエディタやコンパイラまでが OS の一部として動いていました。そうした OS の開発は非常に効率が悪かったのです。ここでちょっと OS の内部のデバッグをどのように行うか想像してみてください。

デバッガは OS の機能そのものに大きく依存して作成されたプログラムです。そうしたものが一切ない、いわば目隠しをされて手探りでデバッグをすることになるのです。(今の技術を使うと、OS のエミュレータなどが使えるため少し簡単になります。)

ともかく、大きなソフトウェアは開発が大変になります。そこで開発するべき量を出来るだけ減らすことが鍵となりました。システムコールの仕様、つまりどこまでを OS に任せるかはこの面から規定されています。つまり、

- より高級な層で行う場合 (puts(),printf() などを取り込む) には、OS 側で書かなければならないプログラムのコード量が増え、バグが入り込みやすくなる。
- より低級な層で行う場合 (write() をさらに分割する) には、ユーザー側が知らなければならない余分な知識がさらに必要となり、ライブラリが複雑怪奇なものとなる。

ということで、unix ではこれらの条件のトレードオ

フ (Trade-Off) を考えて、OS が為すべきことを考えて実現したのが現状だということです。

一方、Microsoft 社の Windows においては、事情が完全に異なります。

Windows では unix のシステムコールに相当するものが WindowsAPI に包含されていて、ライブラリとシステムコールを切り分けることが事実上できません。Windows のバージョンによってどこまでが OS の中で実行され、どこまでがライブラリとして実行され、単独で実行されるかあるいは別な dll(Dynamic Link Library) の内部で実行されるかが異なっています。そのため比較として利用者が比べることができるのはライブラリの関数の個数くらいです。Windows ではその個数は数千個に及びます。

さらに困ったことに、ウインドウシステムは unix では OS とは完全に独立している一アプリケーションプログラムでしかありませんが、Windows では逆に完全に OS と一体となっています。だから、これをどのように数に入れるかは微妙な問題になります。

さらに悪いことに、Windows は最近まで Win16API という大昔の Windows3.1 の時代の API を引きずっていました。その結果、「ファイルを開く」といった単純なことであってもそのやり方はどの時代のどのようなやり方のインターフェースを用いるかによって複数通りあって、状況に応じてそれを呼びわける必要があります。

これだけでも頭が痛いのですが、状況はもっと悪いのです。Windows には文書化されていない API が多数あり、これらは Microsoft のアプリケーションだけが使用しています。これらを使用しない他社のアプリケーションよりも、これらを使用している Microsoft のオフィスアプリケーションは速度的に優位に立つことができるのです。

また、他社がマイクロソフト社製アプリケーションを解析して、これらの「文書化されていない (Undocumented)」API を使おうとすると訴訟沙汰が待っています。ひょっとすると、これらの API はマイクロソフト製品が他社の製品よりも優位に立つことを保障するために入れられたのではないかと思ってしまいます。

Windows は現在では.NET Framework という新しい取り組みを行っており、.net は Windows アプリケーションだけでなく XML や Web サービスを取り込んだ新しい環境に変わりつつあります。そのため、昔の WindowsAPI ではなく、新しく出てきた.net フレームワークに基づく環境でプログラムを開発するように推奨しており、ここで述べたことは時代遅れになりつつあります。

以下は unix と Windows のシステムコール/API コールの比較表です (文献 1 (英語版) より)。これらはまったく同一のものというわけではないので注意して下さい。.NET ではこれらは大分違った形になっています。

unix	Win32API	説明
fork	CreateProcess	新しいプロセスを作る
waitpid	WaitForSingleObject	(子) プロセスの終了を待つ
execve	(なし)	メモリにプログラムをロードし実行。
		Windows では CreateProcess = fork + execve
exit	ExitProcess	プロセスの実行終了
open	CreateFile	ファイルを作成する、ないしは既存のファイルを開く
close	CloseHandle	ファイルを閉じる
read	ReadFile	ファイルからデータを読み込む
write	WriteFile	ファイルにデータを書き込む
lseek	SetFilePointer	ファイルポインタを移動する

stat	GetFileAttributesEx	様々なファイル属性を取得する
mkdir	CreateDirectory	新しいディレクトリを作成する
rmdir	RemoveDirectory	空っぽのディレクトリを削除する
link	(後までなかった)	ファイルをリンクし、リンクカウントを増やす
		(つまりファイルに別名を付ける)
unlink	DeleteFile	リンクカウントを減らし、0 になったらファイルそのも
mount	(なし)	ファイルシステムをマウントする
umount	(なし)	ファイルシステムのマウントを外す
chdir	SetCurrentDirectory	カレントワーキングディレクトリ
		(Current Working Directory;cwd) を変更する
chmod	(WindowsNT 以降にはあり)	ファイルの属性を変更する
kill	(なし)	プロセスにシグナル (signal) を送る
time	(GetLocalTime)	現在の時刻を取得する

Windows-PC のハードウェアについて

具体的な OS の中の話を始める前に、Windows を動かす PC(Personal Computer) に付属していて OS によって管理されている様々なデバイスの紹介と、アーキテクチャ I では触れなかったハードウェアの話をもう少し細かく見ておきましょう。ホテルにはどのような設備があり、冷蔵庫内にはどのような食

材があるかという話です。

CPU(Central Processing Unit)

そもそも OS はプログラムなのでこれがないと話になりません。CPU はコンピュータアーキテクチャ I でどのように構成されているかの詳細を勉強しました。Windows や最近のマッキントッシュで見掛ける CPU は Intel,AMD(Advanced Micro Device), たまに省電力用として VIA が製造しているもののどれかで、これらはすべて IA-64/IA-32/IA-32e アーキテク

チャを持った CPU です。

見掛けは CISC(Customized Instruction Set Computer) ですが、内側では RISC(Reduced Instruction Set Computer) として動作しています。これらはインストラクションポインタ (ないしはプログラムカウンタ)、汎用レジスタ、ベースポインタ、スタックポインタ、プロセスステータスワードレジスタを持っています。レジスタとは CPU 内部に存在する超高速メモリです。

このCPUはOSが利用するために用意された保護機構レベル (Protection Level) として 4 段階の保護動作レベルがあり、少し前まではほとんどの OS でそのうち 2 つ (一番上と一番下) しか使っていませんでした。

一番上、ないしは内側をカーネル (Kernel) モードと呼びます。OS はこのレベルで動きます。管理用の命令を含め、CPU が全種類の命令を実行できるモードで、不用意なプログラムが接続された機器にダメージを与えることができるという意味では一番危険なモードでもあります。

一番下、ないしは外側をユーザー (User) モードと呼びます。一般用の命令だけが実行されます。この層で管理用の命令などより上位 (内側) のモードだけで許されている命令を実行しようとすると、トラップ (trap) が発動して (トラップって覚えていますよね) カーネルモードでそれを検知します。処理することができるかどうかは上位層次第です。

実行しているプログラムが自分が今どのモードで実行しているかを知るにはプロセスステータスワードレジスタを見ます。しかし、このレジスタに勝手なものを書き込もうとするとやはりトラップが起こってしまいます。

また CPU について、

「CPU は OS から管理される資源の一つであり、複数持つことに不都合はない。」

ことに注意してして下さい。

OS がきちんとマルチプロセッサ化して作成されていれば、CPU がたくさんある場合でもきちんと動作し、かつ CPU の数が増えるに従って性能がよくなるということです。もっとも単純に数が増えても数に比例して性能がよくなるわけではない、というのがせつないところです。

最近では AMD の ryzen3 が 4core8thread、ryzen5 が 6core12thread、ryzen7 が 8core16thread、ryzen9 が 12core24thread〜16core32thread と異なった個数の CPU を出していますが、普通の仕事をする限りは ryzen3 か ryzen5 で十分です。

なお core は普通の意味での CPU の個数を指し、thread は特殊な目的で浮動小数点レジスタなどを汎用レジスタに化けさせて 1 つの CPU 上で無理やり 2 つのプロセスを動かすやり方に対応していることを示します。

メモリ (Memory)

CPU とメモリの間にはキャッシュメモリ (Cache Memory) があります。また CPU とメモリの間に MMU(Memory Management Unit) があります。コンピュータアーキテクチャ I でやったように、主記憶として用いられている DRAM (Dynamic Random Access Memory) はコンピュータから見ると遅いデバイスです。そのため間にキャッシュメモリを置く

ことで高速化を図っています。レジスタを含めたメモリの階層を下に再掲します (文献 1 より)。

アクセス時間	カテゴリ	サイズ (例)
1ns 以下	レジスタ	4KB 以下
2ns	キャッシュメモリ	4MB-64MB
10ns	メインメモリ (DRAM)	4GB-256GB
10ms	磁気ディスク (2 次記憶)	(今は 128GB-12TB

メモリが途方も無く進化した今日でもコンピュータの速度はメモリの速度に大きく依存しています。そしてより高速な RAM を開発しようと努力が行われています。

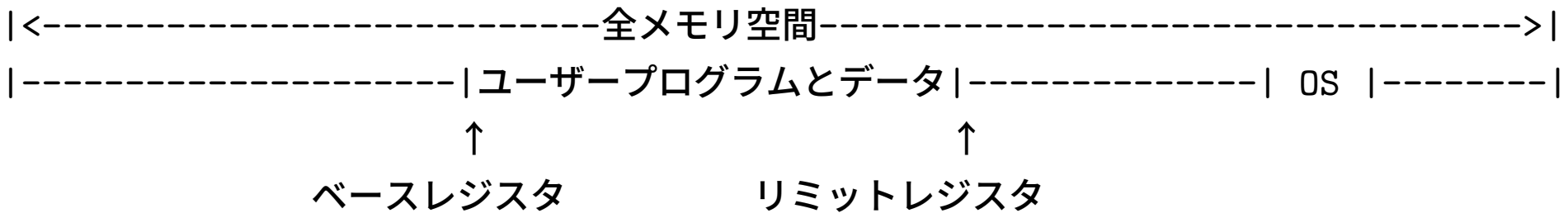
OS は (当然の事として) 同時並行的に複数のプロセスを実行することができます。このとき

1. メモリ上にロードされたプロセスが相互に干渉し合わないようにしたい。
2. メモリ上のプロセスをディスクなどの 2 次記憶に移したりまた元に戻したりするために、プロセスを実行しているメモリの番地を変えたい。

という要件が存在します。この 2 つを実現するために、

1. プログラムを全て 0 番地 (ないしある特定番地) から動くように作成する。
2. プログラムの先頭を示すベースレジスタと、そのプログラムのサイズを示すリミットレジスタという 2 つの特殊レジスタを用意し、CPU がメモリをアクセスする際に、ベースレジスタ分だけ足した値を出力する。メモリアクセスについ

てはそのアクセス範囲がリミットレジスタまでの範囲内にあるかどうかをチェックし、その範囲から外れたら CPU にトラップを上げて範囲外に出たことを知らせる。



ということをやっています。

当然この処理を行うためには「アドレスの足し算を行う」「アドレスが範囲内にあるかどうかをチェックする」という処理上のコストが掛かりますが、これらはハードウェア的に実現できるもので、他の方式に比べて比較的安上がりにこの 2 つを実現できます。

この方式を仮想アドレス (Virtual Address) 方式と呼びます。またこれを行うものを MMU(Memory Management Unit) と呼びます。基本的な発想はこうですが、実際に用いられている MMU はもっと複雑で機能が豊富です。メモリをアクセスするためにデータバスに出力されるアドレスのことを物理アドレス (Physical Address) と呼びます。

どのプロセスがどの物理番地、ひいてはメモリチップ上にあるのか、ディスク上にあるのかを知って管理を行うのは OS の仕事です。従って、プロセスが切り替わるごとに MMU に適切な値を設定するのは OS の仕事です。

プロセッサの種類によって MMU は CPU の内側だったり外側だったりするのですが、Intel 系列では MMU は CPU の内部に存在しています。

プロセスを次々に切り換えていくことをコンテクストスイッチ (Context Switch) と呼びます。コンテクストスイッチが起こるとほぼ間違いなく余分なメモリアクセスが発生しキャッシュミスが起こります。そのためコンテクストスイッチは起こらなければそれに越したことはありません。

コンテクストスイッチによるプロセスの実行は自動車の信号待ちに少し似ています。タイマによるプロセスの切り替えがそれほど頻繁でないのは、これが理由ですが、現在の CPU では 1/50 秒から 1/1,000 秒くらいの時間でプロセスを切り替えていますが、この時間をさらに短くして、1/10,000 秒 (100uS) にしても実行効率に大きな変化がないことが知られています。

プロセスの中は、text 領域、data 領域 (含 bss 領域、ヒープ領域)、stack 領域と 3 つに分かれています。これらについての説明は後で述べますが、分割をさらに細かく行うことで、複数のプロセスが単一のテキスト領域を共有し、データとスタック領域のみを個別に持つことでメモリの節約や高速なプロセス生成ができるようになります。

今日はここまで。