カーネルとシステムコールを使わないプログラムの プログラミング技法

Tada Yoshikatsu
多田 好克

電気通信大学　大学院情報システム学研究科

Sep. 18 – 20, 2002.　　瓢きん［下呂］

0　はじめに

我々は、FreeBSD 2.2.5 上で稼働している UtiLisp **処理系を、 最終的に裸の** IBM-PC **上で動かすこと**を画策している。これは、汎用の計算機上で Lisp Machine のような専用の Lisp 処理系を作成し、その高速化を目指すものである。

本稿では、その第１段階として、FreeBSD 2.2.5 上で稼働する怪しい UtiLisp 処理系 (*los.onu*) について議論する。裸の計算機上で稼働する、最終的な UtiLisp Machine もどき（今後、*los* と呼ぶ）では、

- Unix のカーネルの機能を使わない；
- Unix のカーネルの機能を**手抜きして**プログラミングする；

ことになる。しかし、*los.onu* では、

- システムコールを極力使わない；
- ライブラリ関数も（基本的には）使わない；
- カーネルの機能の一部は、移植性を考慮して UtiLisp の関数、並びに、Unix 上のＣでプログラミングする；

ということを**目標としている**。つまり、*los.onu* のプログラミングでは、 **システムコールという資源**を意識し、使わないという制約が課せられていることになる。

つまり、Lisp Machine もどきの作成を目指している。なお、この研究の一部は文部科学省科研費基盤研究 (C)(2)「低価格計算機上への高性能言語処理系構築法 (13680403)」の助成を受けている。

los.onu: Lisp operating system on Unix。

手抜きしないと、Unix カーネルを丸々作る破目に陥る ?!

これらの目標の詳細は、次章以降で順次説明する。

「システムコールという資源」は、「カーネルという資源の一部」と読み替えることもできる。

‡Techniques for system calls free programming.
Yoshikatsu Tada,
The University of Electro-Communications.

本稿では続く1章で、*los.onu* の作り方を概観する。つまり、Unix 上で、カーネルとシステムコールを使わないプログラムを如何にして作るかという方法論を考える。続く2章では、現在、プログラミング中の *los.onu* の状況を説明し、3章では現状の *los.onu* の実行結果を吟味する。また、4章では、本方法論の問題点を述べ、今後の方針について議論する。

1 *los.onu* の作り方

1.1 システムコールの除去

los.onu を作るに当たって、我々はまず、システムコールを使わずにプログラミングすることを考えた。これは、組込みシステム(embedded systems)のプログラミングでは日常的に行われていることである。Cで組込みシステムのプログラミングをする場合には、 `crt0.o` **と** `libc.a` を使わない。もしくは、組込みシステムのための `crt0.o` と `libc.a` を新たに記述する。

`crt0.o` には、Cの `main()` を呼ぶまでの処理が書かれている。また、`libc.a` にはシステムコールを呼ぶためのライブラリとその他の有用なライブラリが用意されている。

los.onu の場合には、プログラム自体は Unix 上で作動するため、 `crt0.o` は**そのまま流用することができた**。

そのまま流用できるが、`crt0.o` の内部でライブラリを呼び出す可能性が残っている。*los.onu* の場合は問題は生じなかった。

他方、`libc.a` をリンクしないのは、ある意味やりすぎである。というのは、`libc.a` の中には、 **システムコールを使わないライブラリ関数**も多数存在するからである。しかし、ライブラリ関数の中にはエラー処理時等にシステムコールを呼び出すものも多いので、今回は `libc.a` 全体を使わないことにした。

たとえば、`strlen()` は、システムコールを呼び出さない。

また、カーネルを使わないという意味では、動的リンク(dynamic link)を使うこともできない。近年のCコンパイラの既定値(default)は、動的リンクを使用することになっているので、コンパイル時に`-static` オプションを使用し、これを抑制した。

さらに、Cのオブジェクトが使用する**実行時ライブラリ**(run-time library)**は必要なため**、これらは改めてリンクした。

実行時ライブラリ`/usr/lib/libgcc.a` の内容は CPU に依存する。通常は、浮動小数点数の演算や整数の割算などが必要に応じて用意される。

結局、*los.onu* をリンクするためのコマンドは、

```
% cc -static -nostdlib -o los.onu \
          /usr/lib/crt0.o 各種オブジェクト lsys.o -lgcc
```

ということになった。なお、 `lsys.o` は次節で説明する**未定義関数の再定義によって作られる**オブジェクトファイルである。

この時点では、何も定義されていないオブジェクトファイルと思って良い。

以上を実行すると、 **当然のことながら**多数の未定義関数参照がエラーとして報告される。UtiLisp の場合には表 1に示した 46 個の関数が未定義となった。

当然のことながら、`libc.a` に用意されている関数参照がすべて未定義となる。

1.2 未定義関数の再定義

表 1に示した関数の機能をすべて再定義すれば UtiLisp Machine もどきが完成するわけだが、それはそう**簡単ではない**。そこで、これらの関数を以下の手順で順次書き換えることにした。

1. これらの**関数をすべて他の名前に書き換える**。

簡単な関数も、もちろん存在する。たとえば`bzero(p, l)` は、

```
while (l-- > 0)
    *p++ = 0;
```

の2行で定義できる。

該当する関数呼出しは372ヶ所あったが、ほとんど機械的な書き換えである。

表 1: UtiLisp が使う libc.a の関数

abort	atexit	atoi	bcmp	bcopy	brk
bzero	chdir	close	execl	exit	fprintf
fputs	free	getenv	getpagesize	getpid	localtime
_longjmp	lseek	malloc	open	perror	printf
putenv	rand	read	sbrk	_setjmp	signal
sigsetmask	sprintf	srand	strcmp	strlen	strncmp
strncpy	syscall	system	time	times	ttyname
unlink	vfork	wait	write		

- 関数名の先頭に los_を付けて、los_abort のような名前にする。

2. 以下に示す SH() と_SH() というマクロを用意する。

これらのマクロは los_XXXX という**関数定義の生成**に使用する。

SH(abort) と書くと los_abort() という関数が定義される。この関数は、呼び出されると関数名 abort を表示し、本来の Unix ライブラリ関数 abort() を呼び出す。また、_SH(abort) と書くと、関数は定義されなくなり、自由にプログラミングすることができる。

```
#define SH(name)                                    \
        los_ ## name () {                           \
        puts("\t\t\t\t\t\t\t\t" ## #name);          \
        asm("leave");                               \
        asm("jmp _" ## #name);                      \
}

#define _SH(name)       /* just dismiss */
```

3. 前述の 46 個のライブラリ関数を、取り敢えずすべて、SH(abort) のように定義する。
4. これらの関数の中から、適宜、書き換える関数を選ぶ。書き換えることにした関数は、SH(XXXX) を_SH(XXXX) に変更し、その下に実際の定義を記述する。

以下は、**まだ書き換えていない関数** los_abort() **と、既に書き換えた関数** los_strlen() の定義部分である。

los_abort() はライブラリ関数 abort() を呼び出す。他方、los_strlen() は自前のプログラムを実行する。

```
SH(abort)
/* void los_abort(void) { } */

_SH(strlen)
unsigned int los_strlen(const char *s) {
        int     i;
        i = 0;
        while (*s++ != '\0')
```

```
                i += 1;
        return(i);
}
```

現に、マクロ SH() の定義の中でライブラリ関数 puts() を利用している。ただし、最終的な目的を考えると、その利用は慎重に吟味する必要がある。

なお、これらの関数定義においては、 Unix のライブラリ関数を**利用することはできる**。定義した関数群は lossys.c というファイルにまとめられ、以下のようなコマンドによって、取り敢えず、libc.a がローカルにリンクされる。

```
% ld -r -o lsys.o lossys.o /usr/lib/libc.a
```

2 *los.onu* の現状

この方法論の良いところは、Unix 上で UtiLisp の動作を確認しながら、順次、システムコールとカーネルの影響を減らしていけることである。我々は、これを「にじり寄り法」と呼んでいる。

現在は、システムコールとカーネルの存在を頭の片隅に感じながら、46 個の関数を**順次再定義している**ところである。 以下では、いままでに書き換えた関数を列挙し、個々の関数に対する書き換えの方針や決定事項、その理由を簡単に説明する。また、次章では、現状の *los.onu* を実際に実行した時に、マクロ SH() で出力された、書き換え前の関数のリストを吟味する。

2.1 素直に書き換えられる関数

これらは関数のインタフェースを合わせて、素直に再プログラミングするだけである。

書き換えの終わった関数で、 **基本的に問題の無い関数**は以下の通りである。

atoi(), bcmp(), bcopy(), bzero(), strcmp(), strlen(), strncmp(), strncpy()

将来的には、アセンブリ言語で高速化を図る必要があるかも知れない。

これらの関数は、 今のところ**実行速度をあまり気にせず**、素直にCで記述してある。

また、

_longjmp(), _setjmp()

の２つの関数は、アセンブリ言語で記述したが、基本的には素直に書き換えることができた。

2.2 手抜きをして書き換える関数

実際、この節の後半で示すように、これらの関数はほとんど呼び出されない。

この節で説明する関数は、基本的には UtiLisp の組込み関数として使われているものが多く、 **通常の使用では使われない**。また、UtiLisp の初期設定時にのみ呼び出されるものも多い。

getpagesize()

一応、IBM-PC のページサイズの値である。

に関しては、**単に** 4096 **を返すだけ**の定義にした。 これは、

- 今のところ、他のアーキテクチャの計算機に移植するつもりがない；
- *los.onu* では、実際に MMU を操作する版までは作らない；

という理由による。

また、

```
fprintf(), fputs(), perror(), printf(), sprintf()
```

の各関数は、

- **下層の文字出力関数**があれば、比較的簡単にプログラミングできる；
- 書式(format)指定に使われているパターンは多くないので手抜きができる；

と言った方針でプログラミングすることができる。

`write(1, &c, 1)` とか `putchar(c)` のようなものを想定している。

```
chdir(), getenv(), getpid(), localtime(), putenv(),
rand(), srand(), time(), times(), ttyname()
```

の各関数も取り敢えずは手抜きしてプログラミングして良いと考えている。

`chdir()` は今後設計する**ファイルシステムの構造**に依存する。 *los.onu* では**必要に応じて数個のディレクトリを用意し**、それらの間を渡り歩けるような仕様にすれば良いのではないかと考えている。

4章で議論するが、ファイルシステムの枠組はCではなくてUtiLispで記述しようと考えている。

`getenv()`, `putenv()` や `getpid()`, `ttyname()` などはＯＳに依存する機能で、かつ、UtiLisp としては些細な関数に使われているだけので、これまた、手抜き可能であると考えている。また、時刻／時間に関する `localtime()`, `time()`, `times()` なども同様である。

絶対パス名を１つのファイル名とするような長い名前空間を提供し、ディレクトリは用意しないという方法もある。

`rand()`, `srand()` についても、取り敢えずは、単に適当な数を返す関数と、その返す数を変更する関数として実現する。

2.3 カーネル依存で別途ゆっくり考える関数

Unix のシグナル関連の関数、

```
abort(), signal(), sigsetmask()
```

と、プロセスの生成、制御関連の関数、

```
atexit(), execl(), exit(), syscall(), system(),
vfork(), wait()
```

は、カーネルの提供する機能と深く関わっている。これらの関数に関しては、実際に *los* を裸の計算機上に実装する時に、その実現法を考える。

現状では、 これらの関数は**単に** Unix **のライブラリ関数を呼ぶ**ようにプログラミングされている。ただし、次章で示すように、これらの関数は UtiLisp の特別な関数を呼んだ時に実行される場合が多く、*los.onu* の実現では気にする必要はないと考えている。

つまり、単に `SH(abort)` のままで、何も書き換えてはいない。

2.4 少し工夫をして書き換える関数

メモリ管理関連の関数、

```
brk(), free(), malloc(), sbrk()
```

については、以下の方針で書き換えた。

- *los.onu* 起動時に、Unix のライブラリ関数 `malloc()` を呼び出して **適当な大きさのメモリをまとめて確保する**。

現状では８ Mbyte のメモリを確保している。

実際には、sbrk() は brk() を、malloc() は sbrk() を、使ってプログラミングされている。

- brk(), sbrk(), malloc() **は**、この確保したメモリを配置する。
- free() に関しては、今のところ何もしない。

最後に残った関数は、

close(), lseek(), open(), read(), unlink(), write()

である。これらは、ファイル操作関連のライブラリ関数で、Unix のファイルシステム実現法にに深く関わっている。

ファイルシステム実現法の概略は、4.1節を参照して欲しい。

los.onu では、今後の *los* 実現を視野に入れ、**ファイルシステムの実現法を考え、試作する予定である。**

3 *los.onu* の実行

los.onu 実行時の画面への出力を script コマンドで取得した。行頭の番号は、以後の説明のために付加したもので、*los.onu* からの出力ではない。また、9行、42 行、47 行では、複数の行を省略して示した。

この章では、前章で述べた現状での、*los.onu* の実行結果を吟味する。まず、以下は Unix 上で *los.onu* を実行した時の**画面への出力**である。

```
 1    lib60-27> los.onu
 2                                                    signal
 3                                                    signal
 4                                                    signal
 5                                                    signal
 6                                                    open
 7                                                    read
 8                                                    read
 9    .......... read が 50 回
10                                                    read
11                                                    getenv
12                                                    open
13                                                    read
14                                                    read
15                                                    read
16                                                    read
17                                                    write
18    .utilisprc loaded
19                                                    write
20                                                    read
21                                                    close
22                                                    read
23                                                    read
24                                                    close
25                                                    write
26    >                                               read
27    (cons 1 2)
28                                                    sprintf
29                                                    sprintf
30                                                    write
31    (1 . 2)
32                                                    write
33    >                                               read
34    (call "ls -l")
35                                                    vfork
36                                                    execl
```

```
37                                                      wait
38    total 5366
39    -rw-r--r--  1 yoshi  yoshi       0 Jul 12 17:24 LOS-fsys
40    -rw-r--r--  1 yoshi  yoshi    3248 Aug 29 16:11 Makefile
41    -rw-r--r--  1 yoshi  yoshi   11610 Jul 14 16:31 Memo
42    ..........
43    -rw-r--r--  1 yoshi  yoshi    6801 Aug 30 14:25 lossys.c
44    -rw-r--r--  1 yoshi  yoshi    5103 Aug 30 14:25 lossys.o
45    -rwxr-xr-x  1 yoshi  yoshi  117847 Aug 30 14:25 lsys.o
46    -rw-r--r--  1 yoshi  yoshi   13811 Jul 11 11:34 main.c
47    ..........
48                                                      sprintf
49                                                      write
50    0
51                                                      write
52    >                                                 read
53    (date-time)
54                                                      time
55                                                      localtime
56                                                      write
57    "020904153740"
58                                                      write
59    >                                                 read
60    (time)
61                                                      times
62                                                      sprintf
63                                                      write
64    32
65                                                      write
66    >                                                 read
67    (quit)
68                                                      exit
69    lib60-28>
```

まず、全体の流れは以下のようになっている。

主な仕事はエラー処理、端末からの割込み、ヒープ溢れの検出。

2〜5行 ：SIGBUS, SIGSEGV, SIGINT, SIGILL の各**シグナルハンドラを設定**。
6〜10行 ： UtiLisp の実行時ライブラリ `lispsys.l` を読み込む。
11〜24行 ： 利用者定義の設定ファイル`.utilisprc` を探索し、読み込む。
25〜31行 ：入力促進記号(prompt)の出力と、読み込んだ関数`(cons 1 2)` の評価、出力。
32〜50行 ： `(call "ls -l")` の実行。
51〜57行 ： `(date-time)` の実行。
58〜64行 ： `(time)` の実行。
65〜68行 ： `(quit)` の実行。

これで分かるのは、

- ファイル操作関連のライブラリ以外は、ほとんど Unix に依存していないと考えて良い；
- シグナル処理に関しては、別途、少しプログラミングする必要がある；

- プロセスの生成、制御関連や時刻／時間関連のライブラリは、UtiLisp の**特別な関数**を処理した時に必要になる；

特別な関数という意味では、(gc) 実行時にも times() が呼び出される。

ということである。したがって、ファイル操作関連のライブラリを、別途、**何らかの方法で**プログラミングすれば、*los.onu* 当初の目的である裸の計算機上で UtiLisp を動かすことも不可能ではないと考える。

4 *los.onu* の開発法の問題点

この章では、まず、現在試作中のファイルシステムの実現方針を述べる。その後、本稿で説明した、**システムコールやカーネルの機能を使わないプログラミング方法論**に関して、その問題点を議論する。

つまり、UtiLisp Machine もどきの作り方のこと。

4.1 ファイルシステムの実現方針

この節は、本稿を書いた時点で、まだ、未確定の内容を含んでいる。したがって、この節の内容は現在の雰囲気を示すものと思っていただきたい。

los.onu のファイルシステムは、将来の *los* の実現を想定して、

- **ディスク装置の制御法に近いインタフェース層**を設ける；
- 処理の多くを、UtiLisp **側で請け負う**；

los では、このインタフェースより下層をハードウェア制御プログラムとする。

UtiLisp 側に処理を移すことにより *los* 実現時のプログラミングが手抜きできる。

という方針で設計している。

以下、これら２つの方針に関して、簡単に補足する。

まず、*los.onu* の起動時に**ファイルを１つオープンする**。このファイルが、*los.onu* のディスク装置である。このファイルに対して、セクタ単位の位置指定と、同じくセクタ単位での読み書きのルーチンを用意する。ファイルシステムへのすべてのアクセスは、これらのルーチンを使って実現する。

ファイル名は、今のところ、"LOS-disk"になっている。

他方、UtiLisp 側から呼び出されるはずの、los_open() や los_read() などは素直に実装せず、ファイル名の解析や**ディスクブロックの管理**などは Lisp の関数として実現する。

"LOS-disk"のセクタ単位のブロックの管理のこと。Unix でのｉノード回りの処理に対応する。

また、 UtiLisp に用意されている string-stream **系の機能**を利用すれば、string と"LOS-disk"のブロックとの間での基本的な入出力処理を用意するだけで、ファイル操作のほとんどの部分は実現できるのではなかと考えている。ただし、Unix の概念での標準入出力 (standard input/output) や標準エラー出力 (standard error output) の扱いは、別途、考える必要がある。

stirng オブジェクトを stream としてファイル入出力系の関数で扱う機能。

4.2 *los.onu* 開発法の問題点

以上、システムコールやカーネルの機能を使わないプログラミング方法論として、*los.onu* の開発法を紹介した。この開発法は、比較的有効であると考えているが、以下に示すような問題点も存在する。

- カーネルの提供していた機能を大きく変更できない、または、インタフェースの仕様変更が難しい。
- カーネルの機能を手抜きして実現するためには、開発するプログラムが使用する機能を動的に選択する必要がある。

まず、前者の問題点について議論する。

本手法は、システムコール並びにライブラリ関数のインタフェースを利用して、カーネルの機能を切り離す。したがって、個々の機能を再プログラミングする場合には、 この**インタフェースに引きずられる**ことになる。

たとえば、ファイル操作で、open/read,write/close パラダイムを変更するのは容易ではない。また、標準入出力への操作がキーボードと画面へのアクセスとなるように、ライブラリ関数を再定義すると都合が良い。

今回は、ファイルシステムのインタフェースを変更し、カーネルの提供する機能とは異なる機能の実現を目指している。しかし、UtiLisp インタプリタ側から見ると、従来のファイル操作のインタフェースは残っている。これを大きく変更するには、それなりの作業量が必要と考えられる。

次に、後者の問題点であるが、本稿の冒頭にも書いたように、カーネルの機能を手抜きしてプログラミングしないと、Unix カーネルを再プログラミングする羽目に陥る。本方法論では、 **リンクされないライブラリ関数は使わない機能であると考えて**、それらはプログラミングしないという手抜きをした。

つまり、カーネルの機能を静的に取捨選択したわけである。

しかし、それだけでは不十分で、 **個々のライブラリ関数の使われない機能をも手抜きすべき**だろう。このような手抜きは、プログラムを動的に調べないと遂行できない。ソフトウェア工学におけるスライスの概念が適用できるのではないかと考えているが、 **そう簡単ではない**と思われる。

たとえば、`printf()` のほとんどの書式は使われない。

本手法と同様の方法で、Lisp インタプリタの機能を使わずに Lisp プログラムを動かそうと考えた場合、`eval` の呼び出しに対応できないことは明白である。

5 おわりに

本稿では、カーネルの機能を前提としたプログラムを、裸の計算機上で動かすための方法論を議論した。 適切な機能を用意した**カーネルのようなもの**を使ってプログラムを開発し、最後に本手法を使ってそのプログラムをターゲットマシンに移植することで、組込みシステムのソフトウェア開発効率を上げ、ターゲット上でのオブジェクトを小さく抑えることも可能ではないかと考えている。

ランタイムライブラリ群のようなもの？

その場合、ホストが提供する機能の選択は重要である。たとえば、今回の *los* の実現のように、Unix のような高機能のカーネルを使用すると、そう簡単には実現できない機能も使われるからである。*los.onu* に話を限ると、

- シグナルの処理
- プロセスの生成、制御

の実装は、一筋縄ではいかない。また、

- ファイルシステム

に関しても、それなりの手間は必要である。

ただし、現在の *los.onu* は、Unix という仮想マシン上に実装をしているために難しい問題も発生している。裸の計算機上に実装する場合には、 **ハードウェアの機能を使うことにより**、比較的簡単に実装できる機能も少なくないと思われる。

たとえば、割込みやメモリ管理機構を活用できるだろう。

取り敢えず、*los.onu* の実装を終えた時点で、これらの問題に取り組んでみたいと考えている。