システムソフトウェアと オペレーティング・システム 89－１３
シ ス テ ム 評 価 2－１３
（２００２．２．１５）

汎用OSのアプリケーション特化の半自動化

森若和雄 * 北須賀輝明 † 中西恒夫 ‡ 福田晃 †

* 九州大学 システム情報科学府 † 九州大学 システム情報科学研究院
‡ 奈良先端科学技術大学院大学 情報科学研究科

あらまし

現在汎用 OS をアプリケーションに特化することで、オペレーティングシステム (OS) の消費するメモリ量を削減する研究をしている。

アプリケーションプログラムが API(Application Program Interface) のサブセットしか使わないなら、OS の一部のコードは利用されない。アプリケーションプログラムが利用する API から、OS の中で必要とされている関数をとりだすために、ライブラリと OS それぞれについて、関数の呼び出し関係グラフを作って解析を行う。

本稿では、アプリケーションプログラムを解析して、それが利用している API を調べる手法を提案している。intel386 アーキテクチャ用の linux 上で動作するアプリケーションプログラムの解析について、部分的な自動化を行い、init プログラムに適用した。

Semi-Auto Specializing Technique of Multi-Purpose Operating System for Application Specific

Kazuo Moriwaka * Teruaki Kitasuka * Tsuneo Nakanishi † Akira Fukuda *

*Graduate School of Information Science and Electrical Engineering, Kyushu University
†Graduate School of Information Science, Nara Institute Science and Technology

Abstract

We propose a technique that reduces memory footprint of multi-purpose operating systems (OSs) for application specific.

Application programs use a subset of API (Application Program Interface). It implies which functions of OS are used. We generate a function call graph of OS and library, which can associate APIs with OS functions.

In this paper, a method to analyze application programs is described. We partly automate this method on applications for linux operating system on intel 386 architecture. We applied this method to `init` program.

1 はじめに

実績のあるツールやミドルウェアがあることや、慣れている技術者が多いことなどから、汎用 OS を組み込みシステムに利用するケースが増えている [1]。

組み込みシステムでは、アプリケーションが変更されず、固定的であることが多い。そのためアプリケーションが利用する OS の機能は制限できる。組み込みシステム向け OS では、必要な機能だけを選択して実装することにより、無駄なメモリを消費しないようにできることが多い [2]。

現在、アプリケーションの利用する API に注目して、アプリケーションが OS のどの機能を利用しているかを調べ、その情報を元に汎用 OS が消費するメモリサイズを縮小することを狙った研究を行っている [2]。

API で切り分けることによって、ある OS の実装に対する解析は一回だけでよくなる。アプリケーションについてはその都度解析が必要である。

本稿では、PC 向け汎用 OS である linux を対象に、アプリケーションが利用するシステムコールを、ある程度自動的に調べる手法について提案し、実際に利用した結果について報告する。

2 関連研究

μITRON に沿った OS など、スーパーバイザモードを利用した OS の保護を行わない形式の組み込み向け OS の多くでは、OS をライブラリ形式で実装することによって、アプリケーションが必要とする機能だけを提供する仕組みを提供している。アプケーションは OS の機能を関数呼び出しによって呼び出す。明示的にシンボルを指定して呼び出しているので、リンカを利用してライブラリの中から必要な関数だけを取りだすことができる。

特定アプリケーションに特化した OS の変更を行う研究としては、コンポーネント指向 OS について、いくつかの研究が行われている。コンポーネント指向 OS では、OS を複数のコンポーネントに分割して実装する。必要な機能だけを選択的に OS に含ませるだけでなく、一つの機能 (メモリ管理など) に対して複数の実装をコンポーネントとして用意し、適切なものを選択することができる [5]。

コンポーネント間の依存関係を管理する方法として、実装言語の機能を拡張して依存関係の宣言ができるようにするもの [6] や、実装言語とは別に、コンポーネント間の依存関係を定義する言語とそれを利用したコンフィギュレータを利用して必要なコンポーネントをみつけるもの [7] がある。

また、汎用 OS を特定のアプリケーションに特化させる研究としては、web サーバーの挙動を考慮してディスクスケジュールを改造する研究 [8] が行われている。

3 処理の概要

API は関数の集合という形で定義されているとする。以下では API 定義に含まれるそれぞれの関数を API 関数と呼ぶ。

本手法に必要な処理としては以下の 3 つがあげられる

- アプリケーションの利用する API 関数を解析する。
  API 関数の一覧表を用意し、スタティックリンクを行った後、バイナリファイル中のシンボル情報から利用されている関数を調べる。ユーザープログラム中で API 関数と重複する名前の関数がある場合には、利用されていない API が利用されていることになってしまう。

- 各 API 関数を解析して、利用するシステムコールを探し、各 API について、それに対応するシステムコールの表を作る。
  ある API がライブラリ内だけで実装されていてシステムコール呼び出しを必要と

```
804c371:       b8 43 00 00 00          mov    $0x43,%eax
804c376:       8b 5d 08                mov    0x8(%ebp),%ebx
804c379:       53                      push   %ebx
804c37a:       89 db                   mov    %ebx,%ebx
804c37c:       cd 80                   int    $0x80
```

図 1: システムコールの呼び出しコード例 (sysvinit 2.78 より引用)

しない場合には、0 個以上のシステムコールが対応すると考える。ここでは、API 関数以外のライブラリ関数やユーザ関数は直接システムコールを呼び出さないと仮定している。

- カーネル内のシステムコールの実装から、各 API に対応して起点となる関数の表を作る。
  実装上ではメッセージを受信する関数や、割り込みハンドラといった関数が起点となるが、各 API に対応する関数の表を別に作ることによって、関数内の条件分岐の解析などを行わずに、関数を単位とした依存関係の解析が行える。

解析すべきものは、アプリケーション、API 関数を実装するライブラリ、カーネルとなる。[2] では MINIX[3] についてこの解析を机上で行い、各モジュール毎に必要になるソースコード行数を調査した。

4 Linux アプリケーションの解析

Linux 上で動作するアプリケーションを解析することによって利用するシステムコールを検出する。

4.1 API 関数

前述した概要では、API 関数だけが OS の機能を直接呼び出すような制約を仮定している (図 2)。しかし、linux で一般に使われている glibc では、どの関数が API 関数かという

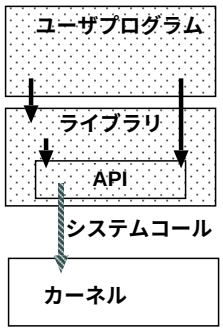

図 2: API とシステムコールの関係のモデル

明確な分類はない。さらに、glibc ではアプリケーションから直接 API を呼び出すための関数 `syscall` を提供している [4] ので、上の仮定を満たさないプログラムを簡単に書けるし、プログラム中でインラインアセンブラを利用することも考えられる。

アプリケーションが利用するシステムコールを半自動的に探す手法を考えた。

4.2 linux のシステムコール

Linux(intel386 版) のシステムコールの実装は、以下のようになっている [9]。システムコールの実行はアセンブリ命令の `int $0x80` を実行することによって行われる。この命令はベクタ番号 128 のソフトウェア割り込みを発生させる。カーネルは多くのシステムコールを実装するため、引数としてシステムコール番号を渡して、要求するシステムコールを識別する必要がある。引数を渡すために eax レジスタが利用される。

```
807594a:       ba 2d 00 00 00          mov    $0x2d,%edx
807594f:       89 d0                   mov    %edx,%eax
8075951:       89 da                   mov    %ebx,%edx
8075953:       89 db                   mov    %ebx,%ebx
8075955:       cd 80                   int    $0x80
```

```
asm ("movl %%ebx, %1\n"       /* Save %ebx in scratch register.  */
     "movl %3, %%ebx\n"       /* Put ADDR in %ebx to be syscall arg.  */
     "int $0x80 # %2\n"       /* Perform the system call.  */
     "movl %1, %%ebx\n"       /* Restore %ebx from scratch register.  */
     : "=a" (newbrk), "=r" (scratch)
     : "0" (SYS_ify (brk)), "g" (addr));
```

図 3: edx レジスタを経由した eax レジスタの初期化コードとその元のインラインアセンブラ記述 (glibc 2.13 より引用)

以上から、アプリケーションプログラムが `int $0x80` を実行する箇所と、その時の eax レジスタの値がわかれば、実際にどのシステムコールが呼ばれるか判別することができる。

4.3 アプリケーションが利用するシステムコールの検出

上述したように、i386 版の Linux では eax レジスタを設定し、`int $0x80` を実行することでシステムコールを呼び出す。実際のコードではどのように呼び出されるかを考える。

1. アセンブラで eax レジスタ代入、`int $0x80` 呼び出し。
2. インラインアセンブラで eax レジスタ代入、`int $0x80` 呼び出し。
3. syscall 関数による呼び出し。
4. syscall 関数と同等の関数による呼び出し。

経験的に、システムコール呼び出しは互換性の為に API 関数経由で行われることが多いので、1 と 2 がほとんどを占めると思われる。そこで 1,2 のほとんどの場合について自動化し、3,4 についても、どのシステムコールが利用されているかまではわからなくても、人手で確認できるように、どこで呼ばれているか発見する仕組みを考えた。

API の並び順には特に意味がないので、なんらかの計算結果が eax レジスタに代入されることは滅多になく、図 1 のように、eax レジスタに即値を代入している場合が多いと考えられる。

4.4 システムコール検出の自動化

最も簡単に解析できる場合を対象に、アプリケーションがどのシステムコールを呼び出しているかを検出し、またどのシステムコールを呼出しているか検出できない場合にはその箇所を示すことができるシステムを考えた。

1. アプリケーションをコンパイルおよびリンクして、実行形式にする。
2. 逆アセンブルして、命令列を取り出す。
3. call 命令や jmp 命令などのジャンプ先になっているアドレスで命令列を切り分ける。
4. 切り分けた命令列の断片のうちで、`int $0x80` 命令を含むものを探す。

表 1: init プログラムがシステムコールを経由して呼ぶ linux カーネル内の関数

```
old_mmap sys_access sys_alarm sys_brk sys_chdir sys_close sys_dup
sys_dup2 sys_execve sys_exit sys_fcntl sys_fork sys_ftruncate
sys_getdents sys_getegid sys_geteuid sys_getgid sys_getpid sys_getresgid
sys_gettimeofday sys_getuid sys_ioctl sys_kill sys_llseek sys_lseek
sys_mknod sys_mprotect sys_msync sys_munmap sys_newfstat sys_newstat
sys_newuname sys_open sys_pause sys_pipe sys_prctl sys_pwrite sys_read
sys_readlink sys_reboot sys_rt_sigreturn sys_sched_rr_get_interval
sys_select sys_setitimer sys_setrlimit sys_setsid sys_sigaction
sys_sigprocmask sys_sigreturn sys_socketcall sys_time sys_umask
sys_unlink sys_wait4 sys_write
```

5. `int $0x80` の前で eax レジスタを設定している命令をみつけて、即値が代入されていればシステムコールその番号を出力する。そうでなければその箇所を出力して人手で処理させる。

Linux カーネル バージョン 2.2.17、glibc バージョン 2.1.3、sysvinit バージョン 2.78 を対象にして実験を行った。init プログラムをスタティックリンクし、objdump プログラムで逆アセンブルした結果を対象に、上記の処理を行って調べたところ、一箇所を除いて全てのシステムコール呼び出しでは、eax レジスタに即値を代入したのち `int $0x80` を実行していた。唯一の例外である図 3 のコードでは、edx レジスタを経由して eax レジスタを初期化していた。これはインラインアセンブラを展開する際に発生したコードである。syscall または同等の関数を利用したシステムコールの呼び出しは存在しなかった。

同じシステムコールが複数の箇所で呼ばれていることもあった。

上のプログラムで得られた番号一覧から、linux ソースコードに含まれる arch/i386/entry.S 内の `sys_call_table` を参照して、init プログラムが利用するシステムコールの一覧を得た。 `sys_call_table` は、システムコールのソフトウェア割り込みを受けたハンドラが、システムコール毎に対応する関数へ分岐するために利用するポインタ配列である。この結果、init プログラムはシステムコールを通じて表 1 の 55 個の関数を呼ぶことがわかった。

5 おわりに

intel 386 版 Linux の上で動作するアプリケーションを解析し、そのアプリケーションが利用しているシステムコールのほとんどを自動的に列挙するプログラムを作成した。実際に init プログラムに対して適用し、必要なシステムコールを見つけ、それが呼び出すカーネル内関数を、ほぼ自動的に得ることができた。

今後の課題としては、今回例外として扱ったような、他レジスタを経由した即値の代入について、データ依存を解析して自動的に取得できるようにすることがあげられる。さらに、ソフトウェア割り込みを利用してシステムコールを実行する形式の OS 全般で利用できるようなフレームワークを構築できれば、静的に OS のどの機能が利用されているかを解析するツールとしての価値が上がるだろう。

謝辞

本研究の一部は，科学技術振興事業団（JST）の戦略的基礎研究推進事業（CREST）「高度メディア社会の生活情報技術」プログラムの支援のもとに行われたものである。

参考文献

[1] 堀内岳人, 加納護: 組み込み OS としての BSD ,BSD マガジン, Vol.6, pp.17–23 (2000).

[2] 森若和雄他: 特定用途のための汎用 OS のサイズ縮小に関する考察, 情報処理学会研究報告，2001-OS-87, pp.1–8 (2001).

[3] Tanenbaum, A.: The MINIX Home Page (1996). `http://www.cs.vu.nl/~ast/minix.html`.

[4] Sandra Loosemore et al. :The GNU C Library Reference Manual (1999).

[5] Frohlich, A. A.: Tailor-made Operating Systems for Embedded Parallel Applications, *Proc. of the 4th International Workshop on Embedded HPC Systems and Applications (EHPC)*, pp. 1361–1373 (1999).

[6] Reid, A., Flatt, M., Stoller, L., Lepreau, J. and Eide, E.: Knit: Component Composition for Systems Software, *Proc. of 4th Symposium on Operating Systems Design and Implementation(OSDI)*, pp. 347–360 (2000).

[7] Red Hat Inc.: *eCos User's Guide* (2000). `http://sources.redhat.com/ecos/docs-1.3.1/guides/user-guides.html`.

[8] スカンヤ・スラナワラッ, 谷口秀夫: WWW サーバにおけるサービスの処理内容を考慮したディスクスケジュール法の実現と評価, 情報処理学会論文誌, Vol. 42, No. 6, pp. 1472–1482 (2001).

[9] Bovet, D. P., Marco Cesati 著 高橋浩和, 早川 仁訳: 詳解 LINUX カーネル, オライリージャパン (2001).