ソフトウェア第三
— デバッガ，オペレーティングシステム，デバイスへのアクセス —

機械情報工学科　水内郁夫
http://www.jsk.t.u-tokyo.ac.jp/~ikuo/lec/soft3/
(ID:soft3, pass:MechanoI)

2008 年 10 月 27 日 (月)

1　前回までの補足

1.1　クロス開発

通常のアセンブラ・コンパイラ・リンカは，それが動いているコンピュータ上で実行するファイルを生成するために使用する．組込み系のプロセッサのプログラムは，別の汎用のコンピュータ上でプログラムをコンパイル・アセンブルする場合がある．これをクロス開発・クロスコンパイルと言う．例えば，自動販売機のプロセッサ，ロボットの内部に分散したプロセッサ等は，そのプロセッサ上でコンパイラを動かすことは難しいので，PC 等の別の計算機の上で開発しコンパイルを行う．

情報理工学研究科から貸し出したノート PC には，機械情報工学科の演習で使用する，SH というプロセッサを搭載した基板向けのクロス開発環境をインストールして利用できる．SH 上で走る実行ファイルを生成する時は，`gcc` の代わりに，`sh-linux-gcc` を使用する．同様に，`sh-linux-size`，`sh-linux-objdump -d`，`sh-linux-nm`，`sh-linux-ar` 等もある．

Windows の場合は，`http://rpm.sh-linux.org/rpm-2004/cygwin/` に Cygwin[1]用のコンパイル済みのクロス開発ツールがあるので，ダウンロードして使用することができる．binutils-sh-linux, gcc-sh-linux, glibc-sh-linux の三種類があれば，SH の上で動くプログラムをクロス開発することができる．Cygwin を起動して，

```
$ cd /; tar jxvf binutils-sh-linux-2.15.91.0.1-1.tar.bz2
```

のようにすれば良い．

10 月 6 日配布資料「**4.3 gcc によるコンパイルの様子を見る**」の調査をクロス開発環境を用いて行い，クロスの GCC が生成するコードとネイティブ[2]の GCC が生成するコードを比較してみよう．PC に搭載された Intel のプロセッサと，機械情報演習で使用する基板の Renesas の SH プロセッサとは，命令セットが大幅に異なり，ニーモニック (mnemonic) も違う．オペランドの並び順が異なる場合もある．

[1] `http://www.cygwin.com/`
[2] クロスではないプロセッサ向けのことをこのように表現することがある．

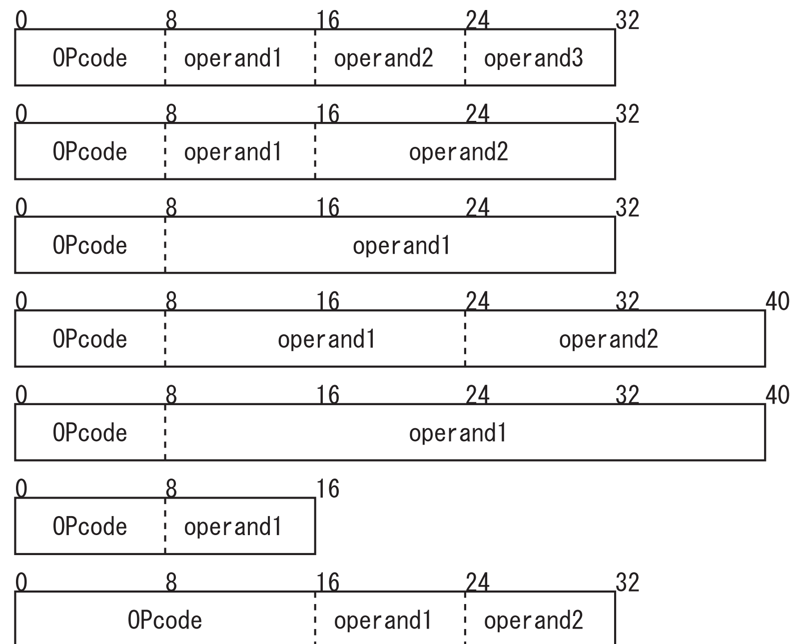

図 1: 機械語のビット列の例

1.2 アドレッシングモードと命令セット

1.2.1 アドレッシングモード

機械語の命令は，最終的にはオペコード (操作内容) とオペランド (操作対象) を含む，0 と 1 の羅列である．CPU(の命令解釈部) は，その 0 と 1 のビット列を解釈して，どのようにオペランドを見つけ出すのだろうか．オペランドの部分に，メモリアドレスが書いてある場合，レジスタ番号が書いてある場合，等，通常いくつかのアドレッシングモードが利用できるようになっている．アドレッシングには，即値アドレッシング，レジスタ直接アドレッシング，レジスタ間接アドレッシング，メモリ直接アドレッシング，メモリ間接アドレッシング，インデックスアドレッシング，スタックアドレッシング等がある．

即値アドレッシング 機械語の命令に値がそのまま埋め込まれている．図 1

レジスタ直接アドレッシング レジスタにはそれぞれ番号がついており，その番号を指定する．

レジスタ間接アドレッシング 指定されたレジスタの値をアドレスとみなして，その番地のメモリ内容にアクセスする．

メモリ直接アドレッシング アクセスするメモリ番地が機械語の命令に埋め込まれる．

メモリ間接アドレッシング 指定されたメモリの内容をアドレスとみなして，その番地のメモリ内容にアクセスする．

インデックスアドレッシング (アドレス+指定されたレジスタの値) のアドレスにアクセスする．

```
00001001 00000001 00000010 00000101                                  
   ADD      R1       R2       R5      (R2+R5->R1)                    レジスタ直接アドレッシング
---------------------------------------------------------------------------------------------
00001010 00000001 00000101 00100011                                  レジスタア直接ドレッシング
   ADD      R1       R5      #35      (R5+35->R1)                    即値アドレッシング
---------------------------------------------------------------------------------------------
00001011 00000001 00000101 11011100                                  レジスタ直接アドレッシング
   ADD      R1     #1500(#x05dc)      (R1+1500->R1)                  即値アドレッシング
---------------------------------------------------------------------------------------------
00000011 00000001 00000000 00000001 11100010 01000000                レジスタ直接アドレッシング
   MOV      R1     #123456(#x0001e240)                (123456->R1)   即値アドレッシング
---------------------------------------------------------------------------------------------
00000011 10000001 00000001                                           レジスタ間接アドレッシング
   MOV     (R1)      R1     (R1 の値の番地のメモリ内容 ->R1)          レジスタ直接アドレッシング
---------------------------------------------------------------------------------------------
00000101 00000100 00000010 00010010 00110100 00000000
   MOV     R4        R2     0x123400(メモリアドレス A)  (A+R2 の番地のメモリ内容 ->R4)
                                                                     レジスタ直接アドレッシング
                                                                     インデックスアドレッシング
---------------------------------------------------------------------------------------------
00100000 00010010 00110100 01010111 11001000
   JMP                 0x123457c8       (0x123457c8 にジャンプ)      直接アドレッシング
---------------------------------------------------------------------------------------------
00101010 11110100                                                    ＰＣ相対アドレッシング
   BRA      -12     (現在のアドレス -12 へ分岐)                       (PC: program counter)
```

図 2: 機械語の命令のビット列の例

1.2.2 機械語の命令の長さ (ビット数)

例えば，加算演算の命令の場合，A+B→C という演算を行う場合，A, B, C の三つのオペランドを指定するか，結果の代入先を A とする (A+B→A) ことで A, B の二つのオペランドを指定するか，によって，機械語の命令の長さが変わってくる．(図 2 参照)

さらに，一つのオペランドに何ビット割り当てるかという問題がある．オペランドがレジスタを指定する場合 (レジスタ直接・レジスタ間接)，レジスタの数に応じて必要なオペランドの長さが決まる．例えば，汎用レジスタが 20 本ある CPU であれば，レジスタ番号は 5bit あれば足りる．図 2(様々な機械語の命令のビット列の例) では，レジスタの指定に 8bit を使っているが，通常はもっと少ないビット数である．

メモリ直接や，メモリ間接のアドレッシングの場合，命令にメモリアドレスを埋め込むことになるが，扱える最大メモリ容量に応じて，メモリアドレスの表現に必要なビット数が増える．4GB のメモリ空間を 1 バイトずつアドレッシング可能にするには，0x00000000〜0xFFFFFFFF のアドレスが必要であり，32 ビットのビット幅が必要になる．

アドレッシング方式の指定は，それぞれ異なった命令番号 (OPcode) を割り当てる場合 (図 2 はそのようになっている) もあるし，オペランドのビットの中に 1〜3bit 程度のモード指定ビットがある場合もある．レジスタ番号の指定に 5bit 使いモードの指定に 3bit 使う．

オペランドを三つとるような命令がある 32 ビット CPU の場合，現実的なビット配分は，次の三通りのようになるだろう．

1. レジスタ＋レジスタ→レジスタの形：オペコードに 8bit，モードに 1bit，代入先レジスタ (DEST) に 5bit，ソースレジスタ１に 5bit，ソースレジスタ２に 5bit．残りの 8bit は使用しない．

2. レジスタ＋オフセット→レジスタの形：オペコードに 8bit，モードに 1bit，代入先レジスタ

(DEST) に 5bit，ソースレジスタ１に 5bit，オフセットに 13bit.

3. 24bit でアドレス (やジャンプ距離) 指定のジャンプの形：オペコードに 8bit，アドレスやジャンプ距離の指定に 24bit.

1.2.3 RISC と CISC

RISC とは Reduced Instruction Set Computer を意味し，CISC は Complex Instruction Set Computer を意味する．RISC と CISC は，CPU(プロセッサ) の命令セットの設計の指針を表す単語である．

RISC は，命令のバリエーションを少なくして (reduced)，複雑な命令に相当する機能は，複数の命令を組み合わせることで実現するという考え方で設計されている．例えば，メモリ間接アドレッシングのモードは持たないが，(1) メモリアドレスをレジスタに MOV する，(2) レジスタ間接アドレッシングによりそのメモリ番地のメモリ内容にアクセスする，というな形である．

初期のプロセッサは，機械語でのプログラミングが基本であり，人間にわかりやすいように様々な命令が用意されるようになっていった．さらに，コンパイラがコンパイルする結果としての機械語命令に必要なバリエーションを用意するという目的もあった．このような設計思想が CISC である．CISC は，一つの命令を実行するのに必要な時間 (クロックサイクル数) が命令によって異なる形になっていた．また，機械語の命令のビットサイズが，命令によって異なるため，命令を解釈する回路も複雑になった．

RISC は，全ての演算を高速に実行できるように単純な命令のみを実装し，全ての命令が同じビット幅となるようにし，パイプラインによる実行効率の向上や，回路の単純化によるクロックスピードの向上を目指した．

パイプラインは，命令の読み込み・解釈・実行・結果の保存等の何段階 (ステージ) に分けて実行し，前の命令の実行中に次の命令の解釈・次の次の命令の読み込みを行うというような形で，各段階の回路 (命令読み込み回路 (fetch)・演算回路 (execute)・解釈回路 (decode)・結果記録回路 (store) 等) が，同時に動作するようにして，単位時間当たりの命令実行数を向上させた．また，各ステージの実行時間は短くなり，クロックサイクルも速くなった．

高級言語で書かれたプログラムをコンパイルすると，RISC の方が CISC よりオブジェクトモジュールのサイズが大きくなる．プログラムを記憶するメモリ容量の拡大やコンパイラ技術の進化も，RISC の発展を助けた．

2 デバッグ

2.1 gdb

プログラムのバグを取るツールは，色々とある．最も単純なものは，printf デバッグである．ここでは，GNU のデバッグツールである gdb を紹介しよう．gdb は，ステップ実行，ブレークポイント，ウォッチポイント，バックトレース等の機能を持つデバッガである．

gdb を使うためには，gcc によるコンパイル及びリンクの際に，`-g` というオプションを付ける事が必要である．これにより gcc は gdb が必要とするデバッグ情報を埋め込んだオブジェクトモジュール及び実行可能ファイルを生成する．最適化オプション`-O` をつけてコンパイルすると，実行中の行と機械語の対応が一対一でつけられない部分などが生じる場合がある．

```
% gcc -g -c gdbtest.c
% gcc -g -o gdbtest.exe gdbtest.o
% gdb gdbtest.exe
(gdb) run
```

gdb を利用するには「gdb 実行ファイル」として起動し，(gdb) というプロンプトに gdb のコマンドを入力する．主なコマンドは以下のとおり．

r [args]	run でも可．プログラムを gdb の中で実行開始する．引数を渡すには，r 1 2 のようにする．
b arg	break でも可．ブレークポイント[3]を設定する．arg には，関数名又はファイル名:行番号を与える．b main とすると，main 関数開始時点で停止する．
c	continue．ブレークポイントで停止したプログラムの実行を再開．
n [n]	next．行単位での実行．関数呼び出しは中には入らず．
s [n]	step．行単位での実行．関数呼び出しは中に入る．
p args	print．変数の値を表示する．
watch args	ウォッチポイントを設定する．args に与えた変数が変更されると，それを伝える．
q	quit
help	ヘルプを見る．
bt	backtrace．現在実行中の関数を呼んだ関数を辿って表示する．

gdb は，Emacs(や Meadow[4]) の中から起動すると，ブレークやステップ実行時に，ソースファイルを開いて実行を指し示してくれる．起動方法は，M-x gdb とする．gdb 内で利用するコマンドは，コマンドラインから実行した場合と同様である．

組み込みマイコンのプログラムのデバッグ

組み込みプロセッサ (ボードマイコン・ワンチップマイコンなど) 用の gdb も存在する (sh-linux-gdb 等)．しかし，ステップ実行などを行うためには，実行環境をシミュレーションする (エミュレータ) か，特殊な回路を組み込んで一時停止等を可能にする (ICE; in-circuit emulator) 必要がある．ICE は高価な機器であった．近年では，JTAG と呼ばれるチップの端子状態を読み出すインタフェースと，UDI(user debugging interface) 等と呼ばれるデバッグ用回路を内蔵したマイコンの組み合わせ (JTAG-ICE と呼ばれる) により，安価なデバッグ環境が実現されているマイコンも多い．

[4] Meadow で gdb を使用する場合は，Meadow で Cygwin のディレクトリ構造を使う (デフォルトでは Windows のディレクトリ構造) ために，以下のセットアップを行うと良い (機械情報貸出ノート PC は設定済み)．
1.http://www4.kcn.ne.jp/~boochang/emacs/elisp.html から，
http://www4.kcn.ne.jp/~boochang/elisp/cygwin-mount-mw32.el と
http://www.blarg.net/~offby1/cygwin-mount/cygwin-mount.el をダウンロード．
2.(Meadow のインストールされたディレクトリ)/site-lisp/cygwin-mount-mw32.el に置く (cygwin-mount.el も同様)．
3.Meadow で M-x byte-compile-file により cygwin-mount-mw32.el をコンパイルする (cygwin-mount.el も同様)．
4.~/.emacs に，(if (featurep 'meadow) (progn (require 'cygwin-mount-mw32) (cygwin-mount-activate))) を追加する．

3 オペレーティングシステムの概要

オペレーティングシステム (OS) の役割は，大きく二つに分類できる[5]．一つは，**リソース管理**の役割である．複数のメモリ領域にロードされているプログラムを時分割方式 (TSS; time sharing system) で実行することは，メモリと演算装置というリソースを OS が管理することで実現される．もう一つは，**命令セットの拡張**としての役割であり，各プロセッサの提供する命令と OS が提供する機能 (system call) を合わせて，多機能の命令セットを持つプロセッサとみなすことことができる．

3.1 リソース管理の役割

コンピュータに備えられているハードウェアリソース (資源) には，プロセッサ，メモリ，外部記憶装置 (ディスク)，各種入出力デバイス等がある．これらを適切に管理し，必要なプログラムに必要なリソースを提供するための枠組みが，OS の一つの役割である．

リソース管理の一つの目的は，有効利用することである．プロセッサをアイドル (idle) にしておく時間をできるだけ少なくし常にビジー (busy) にしておくことが，そのプロセッサを有効利用していることになる．プロセッサ資源の有効利用のために，マルチプログラミングの概念が考案され，現在の OS のプロセス・スレッドの仕組みが出来上がった．メモリ資源の管理については，マルチタスクのメモリ管理や，仮想メモリの管理が OS の役割である．ファイルの概念，ディレクトリの概念を実現する，ファイルシステムも OS の役割である．ユーザの概念に基づくプロセスやファイルの所有者の取り扱いも OS の機能である．入出力とデバイスの管理も OS の機能である．

3.2 命令セットの拡張としての役割

機械語 (命令セット) はプロセッサが提供する機能の一覧である．機械語は必ずしも回路に直接対応してはおらず，場合によってはいくつかの回路動作手順[6] をまとめて一つの命令にしているような命令もある[7]．いくつかの機械語を組み合わせて実現される機能を**システムコール**として提供するのが OS の一つの役割であると考えることもできる．つまり，本来のプロセッサが持つ命令セットを拡張した仮想的なマシン (プロセッサ) とみなすこともできる．このような見方をする時，この仮想的なマシンをオペレーティングシステムマシンと呼ぶ．

機械語のレベルではコンピュータは原始的である．オペレーティングシステムマシンはこれを拡張したマシンとみなせ，コンパイラ・コマンドインタプリタ等の様々なシステムプログラムから，上位のアプリケーションプログラムまで，この仮想マシンの (仮想) 命令セットの命令，そしてさらにその命令を組み合わせた機能 (ライブラリ) を用いて実現されている (図 3)．

---

[5] 参考書：

- Andrew S. Tanenbaum, Modern Operating System, second edition, Prentice-Hall, Inc., 2001. (邦訳: モダンオペレーティングシステム, 原著第 2 版, (訳) 水野忠則, 太田剛, 最所圭三, 福田晃, 吉澤康文, ピアソン・エデュケーション, 2004)
- Andrew S. Tanenbaum, Structured Computer Organization, fourth edition, Prentice-Hall, Inc., 1999. (邦訳: 構造化コンピュータ構成, 第 4 版, (訳) 株式会社ロングテール, 長尾高弘, ピアソン・エデュケーション, 2000)

[6] 命令セットより下位のソフトウェア (マイクロプログラム) で実行されている場合と，この手順を実施する回路 (ハードウェア) で実行されている場合がある．この部分は，マイクロアーキテクチャレベルと呼ばれる．

[7] 例えば，レジスタ間接アドレッシングでメモリ間のコピーを行うような命令 (`MOVE @R1, R1`) は，(1) レジスタ `R1` の値を読み出し，(2) その値の番地のメモリ内容をレジスタ `R1` に読み込む (主記憶アクセス) という二つの手順を行う．スタックへのプッシュ(`PUSH R1`) は，(1) スタックポインタの指すアドレスへ値をコピーし (`MOV R1, @SP`; これも二手順必要な場合もある)，(2) スタックポインタをずらす (`ADD #-4, SP` (32 ビットマシンで，尚且つスタックはアドレスの小さ

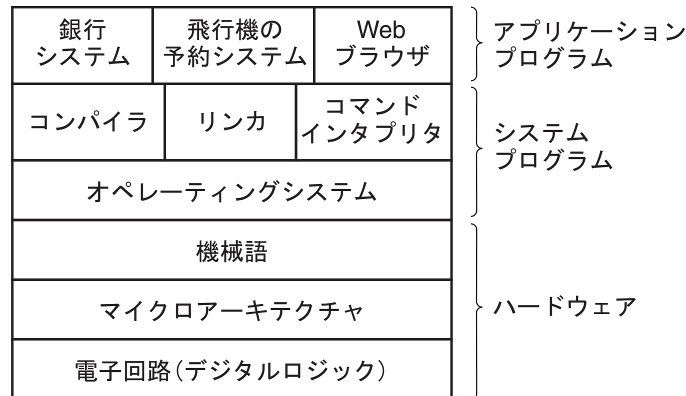

図 3: コンピュータシステムは，ハードウェア，システムソフトウェア，アプリケーションソフトウェアから成る．

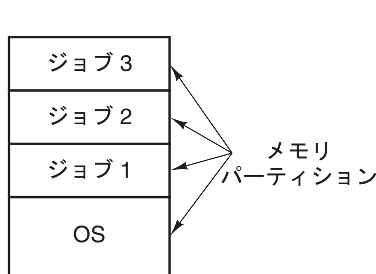

図 4: メモリ内に 3 つのジョブがあるマルチプログラミング

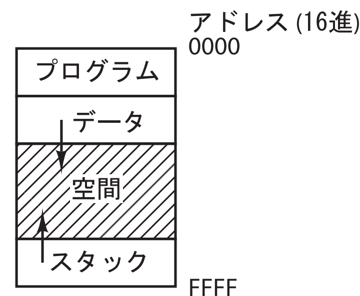

図 5: プロセスは，プログラム，データ (ヒープ)，スタックの 3 つのセグメントを持っている．

4 プロセッサ資源の有効利用

配線盤からバッチシステムへ　真空管を使った初期のコンピュータは，プログラムは配線を組みなおすことで行われていた．OS という言葉も無かった．配線を組み替える間 (プログラムを入力する間) は，演算回路は使用できなかった．1950 年代半ばにトランジスタが導入され，パンチカードに打たれたプログラムはジョブという単位で管理され，複数のジョブをテープにまとめて記録し，バッチというシステムがそれらを順に実行しそれぞれのジョブの結果を出力テープに書き出すというものになった．このバッチシステムが OS の原型であり，プロセッサという計算資源を遊ばせておく時間をなるべく少なくし有効利用するためのシステムであった．つまりプログラム入力の間にも，プロセッサを働かせておくことができるようになったのである．

マルチプログラミング　バッチシステムでは，プロセッサが入出力操作の完了を待っている時間などに，プロセッサがアイドルになっている時間が多くあった．これを解決するために考え出されたアイディアがマルチプログラミングであった．マルチプログラミングは，メモリをいくつかのパーティションに分けて，それぞれのパーティションに別々のジョブ (プログラム) を入れておき，あ

い方へ伸びる場合)) という手順を行う．

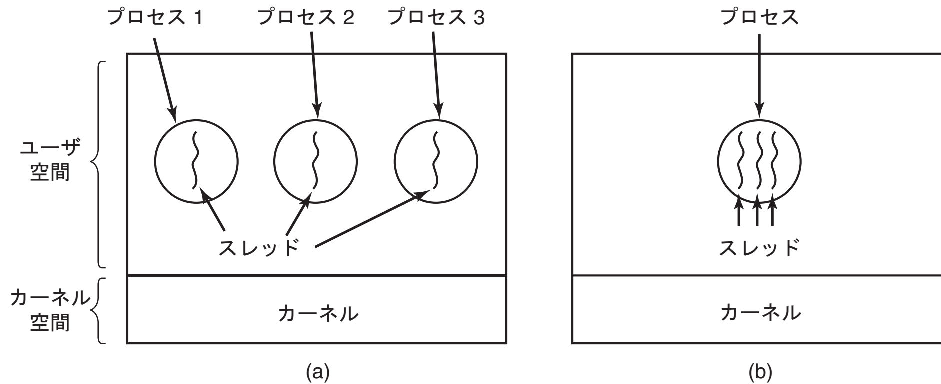

図 6: (a)1 つのスレッドを持つプロセスが 3 つ，(2)3 つのスレッドを持つプロセスが 1 つ

るジョブが入出力完了を待っている間，他のジョブが CPU を使用できるようになった (図 4)．他のジョブ (プロセス) が待ちに入らなくても，ある時間が経つとプロセスを切り替えるシステムが時分割方式である．メモリ上に複数のプログラムが置かれるようになり，メモリの管理も OS の役割の重要な要素になった．

4.1 プロセス

プロセス (process) は，基本的には，実行中のプログラムである．それぞれのプロセスにはアドレス空間 (そのプロセスがアクセスできるメモリ空間) があり，そこにはプログラム，データ (ヒープ)，スタックがある (図 5)．また，各プロセスは，固有のプロセス識別子 (process ID: **PID**)，プログラムカウンタ，スタックポインタ，レジスタセット等，プログラム実行に必要な全ての情報を保持している．プロセスの生成は**親プロセス**が**子プロセス**を生成することで行われ，プロセス木を構成する．

4.2 スレッド

1 つのプロセスの中に複数の処理の流れを持つことができる．この処理の流れの 1 つ 1 つをスレッド (thread) と呼ぶ (図 6)．同一のプロセスの中の複数のスレッドは，同一のメモリ空間を利用する．スタックは各スレッドで独立している．スレッドはプロセスの特徴のいくつかを持っているので，軽量プロセス (lightweight process; **LWP**) と呼ばれることもある．複数のスレッドを使うことを，**マルチスレッド** (multithread) と呼ぶ．

4.3 スケジューリング

プロセッサの数より多いプロセスやスレッドが実行中の場合にどのプロセス・スレッドを実行するかを決めるのが，**スケジューラ** (scheduler) である．スケジューラによって，例えば，図 7 のように複数のプロセス・スレッドが短時間ずつプロセッサ資源を使用する．

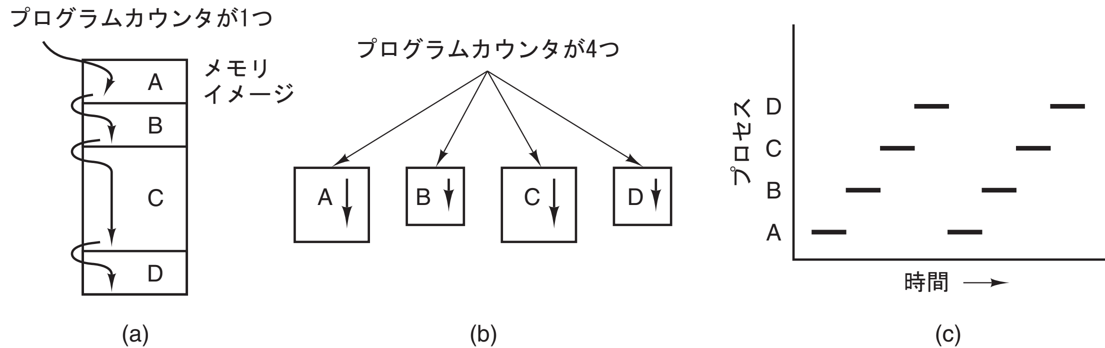

図 7: (a)4 つのプログラムによるマルチプログラミングのメモリイメージ，(b)4 つの独立な逐次プログラムの概念的モデル，(c)1 つのプロセッサで 4 つのプロセスを実行する場合の実行例

プロセス・スレッドには，実行中 (running)・実行可能 (ready)・待ち (blocked)[8]の三種類の状態がある．実行可能状態のプロセス・スレッドは列 (queue) になってスケジューラが CPU を割り当ててくれるのを待っている．CPU が実行するプロセス・スレッドの切り替えを，**コンテクストスイッチ** (context switch) と言う．スケジューラのスケジューリング方式には様々な種類がある．代表的なスケジューリング方式には，到着順 (first-com first-served)，最短ジョブ優先 (shortest job first)，最小残り時間優先 (shortest remaining time first)，ラウンドロビン[9](round-robin scheduling)，優先度スケジューリング (priority scheduling) 等がある．

5　メモリ管理

メモリにはアクセス速度と容量のトレードオフ (trade-off) が存在する (図 8)．理想の高速で大容量のメモリに見せるために，**キャッシュ**(cache) や**仮想記憶** (virtual memory) の仕組みがある．キャッシュの制御はプロセッサ内部で行われ，インストラクションセット (機械語) 以上のレイヤからは隠蔽[10]されている (図 3)．これに対し仮想記憶は，アクセスされていないアドレス領域の内容

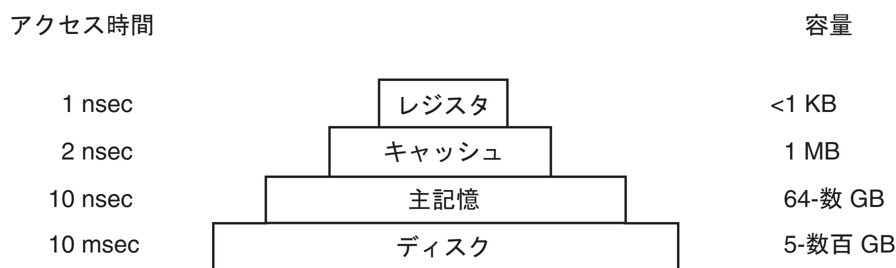

図 8: メモリ階層 (memory hierarchy):アクセス時間と容量はトレードオフの関係

[8]待ち (ブロック) の状態は，入出力の完了を待つような場合に生じる．例えば `read` システムコールを実行し入力が来る前だと，待ちの状態になる．

[9]ラウンドロビンスケジューリングでは，各プロセスは**クオンタム** (quantum) と呼ばれるある時間を割り当てられ，CPU を割り当てられたプロセスはその時間を使い切るか待ち (blocked) の状態になるかすると CPU の切り替えが発生する．

[10]このように階層構造の下の段 (レイヤ, 層) の機能により実現され上の段からは詳しくは見えない (見る必要が無い) ようにすることを，**抽象化** (abstraction) という．

をディスクに書き込んでおくことで，主記憶の容量が実際より大きいかのように使える仕組みだが，これは OS の機能として提供される．

単純なスワッピング (swapping) は，プロセス全体をメモリからディスクへ移したり (スワップアウト)，ディスクからメモリに戻したり (スワップイン) する．仮想メモリは，1 つのプロセスの一部分を実行部分付近をメモリに置き，非実行部分をディスクに置く．

仮想メモリを使用するコンピュータでは，メモリアクセスは仮想アドレス空間のアドレスにより扱われる．仮想アドレス空間は**ページ**と呼ばれる単位に分割されており，メモリとディスクの間の転送はページ単位で行われる．**MMU**(memory management unit) は，仮想アドレスを物理アドレスに変換する．実メモリ上に無いアドレスにプロセスがアクセスすると，CPU はページフォールト (page fault) というトラップ[11]を発生し，OS はそれを受けて最もアクセスの無かったページをディスクへ書き出し，アクセスのあった仮想アドレスに対応する物理アドレスを含むブロックをディスクから実メモリにロードする．そして MMU に更新を伝え，トラップされた命令から再実行する．

仮想アドレスと物理アドレスの関係は，**ページテーブル** (page table) と呼ばれる表によって記憶される．ページテーブルは，ページサイズ (例えば 4KB) と仮想メモリ空間サイズ (例えば 4GB(32bit)) の大きさの比率 (この例えなら約 100 万) が非常に大きいので，テーブル自体のサイズも非常に大きくなる．ページテーブルはメモリに置かれるが，メモリアクセスのたびにアドレス変換のためにページテーブルへのアクセスが必要になるので，一度のメモリアクセスプログラムの実行に，時間のかかる主記憶へのアクセスが何度も発生してしまう．そこで，**TLB**(translation lookaside buffer) と呼ばれる高速・小容量のデバイスを MMU 内に設け，ページテーブルの部分エントリをいくつか記憶しておく．TLB はキャッシュと同様に，TLB に無いエントリが必要になると，最もアクセスの無いエントリを破棄し，主記憶からページテーブルの必要なエントリをコピーする．

更に，近年は 64bit のシステムも増え，**逆引きページテーブル** (inverted page table) を使用して使用メモリ量に応じたテーブルサイズになるような工夫もされている．

6 その他のリソース管理

6.1 ファイルシステム

OS は，ディスクや入出力デバイス装置の特質を隠蔽し，装置に依存しないファイルの生成，削除，読み出し，書き込みのシステムコールを提供する．ファイルの読み出し・書き込みの前後に，オープン，クローズのシステムコールも必要である．また，ファイルをグループ化する方法としてディレクトリの概念を持っている．ファイルのオープンに成功するとシステムコールは**ファイル記述子** (file descriptor) と呼ぶ整数を返す．その後の操作は，ファイル記述子を用いて行う．

UNIX のファイルシステムで重要な概念は，**マウント**である．あるデバイスをマウントすると，元からあるディレクトリツリーの指定された枝に，そのデバイスの持つディレクトリ構造を接続することができる (図 9)．

6.2 入出力・デバイスドライバ

プリンタ・マウス・音声・通信などの入出力デバイスは，UNIX では**デバイスファイル**を介してアクセスする．デバイスファイルは，`/dev/`の下にあり，アクセス方法の違いでブロック型デバイ

[11] 意味は各自調べよう．

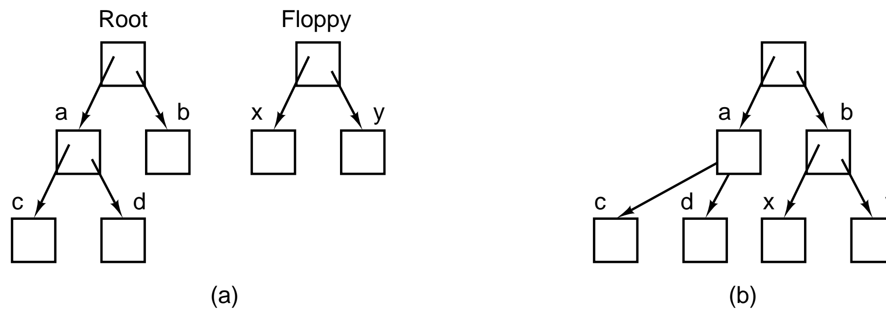

図 9: UNIX ではマウントすることでフロッピーの中身が見えるようになる．

スとキャラクタ型デバイスがある．デバイスファイルをオープンすると，そのデバイスを扱うためのデバイスドライバというソフトウェアを介して，デバイスの使用開始の準備を整える．そしてプログラムがそのファイルへの読み書きを行うことで，デバイスドライバはそのデバイスの操作を行う．

6.3 UID

OS はユーザアカウントの機能を提供する．ユーザアカウントには固有の **UID**(user ID; ユーザ識別子) が割り当てられる．プロセスの所有者，ファイルの所有者は，UID で表され，他の UID のアカウントからは許可されたアクセスしかできない．各ユーザはグループのメンバーになれ，グループごとにグループ識別子 (**GID**) が割り当てられる．ファイルやディレクトリへのアクセスの許可・不許可 (**パーミッション**; permission) は，Read/Write/eXecute を意味する rwx で指定される．例えば，シェルのコマンドである `ls -l` はパーミッション情報を表示し，`chmod` はパーミッションを変更する．

6.4 シェル

シェルは，OS の一部ではないが，OS の機能を頻繁に使用する．sh，csh，tcsh，bash，ksh などの多くのシェルがある．一番元のシェルは sh である．

シェルを起動し以下の入力を実行してみよう．`^D` は Ctr+D である．

```
$ date
$ date > file
$ cat > file1
3 operating system
4 compiler
2 machine language
0 digital logic
1 microarchitecture
^D
$ sort < file1 > file2
$ cat file* | grep 3
$ sleep 10; ls -la | sort -k5n > ls-la &
$ jobs
$ cat ls-la
```

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

図 10: 主な POSIX システムコール．エラーが生じた場合，`-1` を返す．(`s`:リターンコード，`pid`:プロセス ID，`fd`:ファイル記述子，`n`:バイト数，`position`:ファイル内のオフセット，`seconds`:経過時間)

6.5 システムコール; 命令セットの拡張

プロセス管理，ファイル管理，ディレクトリ・ファイルシステムの管理，その他のシステムコールの例を図 10 に示す．Windows のシステムコールである Win32 API と，それと類似の UNIX のシステムコールを図 11 に示す．

7 デバイスへのアクセス

一般にコンピュータに接続されたデバイス (キーボード・プリンタ・ネットワークインタフェース・音声・マウス・通信等々) にアクセスするためのユーザプログラムを書く場合，OS のデバイスドライバを経由する．ユーザプログラムでは，デバイスファイル (UNIX では`/dev/`以下にあるファイル) を `open` し，`read`，`write`，`ioctl` 等のシステムコールを用いてデバイスを利用する．デ

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

図 11: UNIX のシステムコールと Win32 API コールの対応 (完全な 1 対 1 対応ではない)

バイスドライバは，デバイスを操作する具体的な方法を知っていて，ユーザプログラムの呼び出すシステムコールに応じたデバイス操作を行う．ここでは，Linux のシステムにおいてデバイスへのアクセスがどのように行われるかを例にとって説明する．

7.1 カーネル空間とユーザプロセス空間

OS の機能を実行するプログラム (OS そのもの) はカーネル (kernel) と呼ばれる．OS のカーネルの使用するメモリ領域は，一般に物理アドレスで，不動でありページアウトしない (MMU によるアドレス変換が行われない)．ユーザプロセスのメモリ空間は論理アドレスで，MMU による物理アドレスへの変換があり，ページングも発生する．ユーザプロセスは簡単にはカーネルのメモリ空間にアクセスすることはできない．カーネル空間のメモリにアクセスするための専用の関数 (`copy_to_user()` 等) が用意されている．

7.2 システムコール

システムコールは，ユーザプログラム・アプリケーションプログラムが，OS のサービスを呼び出すためのインタフェースである．プログラムからシステムコールが呼び出されると，ユーザプロセスモードからカーネルモードに切り替わる．

また，`fopen`，`printf` 等の標準ライブラリの関数も，中でシステムコールを呼び出している．システムのライブラリ関数の呼び出しは，システムコールに対しライブラリコールと言う．ライブラ

リコールの実行は，ユーザ空間で行われるが，その中から呼び出されるシステムコールの実行はカーネル空間で実行される．

7.3 デバイスドライバ

デバイスドライバは，ハードウェアとソフトウェアを結ぶインタフェースの役割を担うソフトウェアである．コンピュータに接続される物理デバイスは，それぞれが独自の制御装置を持っている．シリアルポートはたいてい UART というチップで制御されるし，ハードディスクは，IDE(ATA)・SCSI[12]・SATA 等のコントローラにより制御されている．個々のデバイスのコントローラは，そのデバイスを操作するためのコントロールレジスタや，そのデバイスの状態を表すステータスレジスタを持っている．デバイスドライバは，バスを介してこれらのコントローラのレジスタにアクセスし，デバイスとのやりとりを行う．

デバイスドライバには，デバイスの操作を行う部分と，ユーザプログラムからのアクセスに対応する部分とがある．

デバイスの操作を行う部分は，デバイス制御装置からの返事 (制御装置のステータスレジスタの変化など) を待たなければならない場合が多い．デバイス制御装置から返事が来たことを検知する方法にはポーリングと割り込みがある．ポーリングは頻繁にデバイスからの返事が来たかを確認するが，デバイスからの返事が来たら割り込みがかかるようにする方が CPU の使用量は少なくてすむ．

Linux では，割り込みの際の処理が軽い場合は即座に行う場合もあるが，割り込みの際に実行する処理量が多い場合は，割り込み処理ルーチンの中では処理せず，割り込みがあったことの記録だけされ，実際の処理は後で行われる．後で行う処理として登録する機構をボトムハーフハンドラ (bottom half handler) と言い，定期的にカーネル内で実行される．

デバイスドライバは一般に OS 起動時にメモリ上 (カーネル空間) に読み込まれる．デバイスドライバのユーザプログラムからのアクセスに対応する部分は，システムコールを介して実行される．デバイスドライバは，ユーザプログラムがシステムコール `open` を呼んだ時に実行するべき関数をカーネルに登録してある．デバイスドライバの中で，ユーザプログラムからシステムコールを介して呼び出されるルーチンを，Linux ではトップハーフルーチンと言う．主なトップハーフルーチンは，`open`，`close`，`read`，`write`，`ioctl` に対応するものである．

ボトムハーフルーチンは，デバイスドライバの中で，物理的なデバイスにアクセスする役割を担うが，この中では，`inb` や `outb` により物理アドレスをしていして，バスに接続された各デバイスの制御装置のレジスタにアクセスする．

デバイス制御装置からの割り込み (IRQ: interrupt require) を受けて OS から起動される割り込みハンドラの登録は，デバイスドライバ読み込み時に行われる．

7.4 デバイスドライバの危険性

デバイスドライバはカーネル空間で動作するので，システム全体に不具合を及ぼすこともできるし，効率の悪い書き方 (例えば，無駄にビジーウェイトするような書き方) をしてあれば CPU を占有し他のプログラムの実行に支障をきたす場合もある．

また，デバイスドライバが利用するメモリ領域も，ページングされないカーネル空間に取られるので，万が一デバドラがメモリリークを起こすような場合，利用できる物理メモリが少なくなって

[12]スカジーと読む

いってしまう．通信デバイスなどのデバイスドライバの場合，送受信データをためておくメモリ領域 (バッファ) が必要になるが，この場合のバッファもページングされないカーネル空間に確保される．

デバイスドライバは，ユーザプログラムのようにシステムコールやライブラリコールを使用することができない．動的メモリ確保は `malloc`, `free` でなく，`kmalloc`, `kfree` を用いる．`printf` の代わりは `printk` であり，syslog にも表示される．

7.5 DMA

データの流量が多い場合は，割り込みでのデバイスとのやり取りは，割り込みの頻度が高くなりすぎて破綻する．データ流量が多いデバイス (ディスク，高速ネットワーク等) は，DMA(direct memory access) を用いてデバイスとメモリの間でデータを直接やりとりすることで，CPU の使用を抑える．DMA で一定量のデータを転送し終わったタイミングなどで割り込みを用いる．

7.6 ローダブルモジュール

デバイスドライバは OS 動作中に動的にロード，アンロードする機構を備えた OS もある．Linux では，このようなデバイスドライバをローダブルモジュールと呼ぶ．デバイスドライバの動的ロードが可能だと，システムリソースの利用効率が向上する．また，カーネルを再構築する必要性が大幅に減る．

Linux で，デバイスドライバをロード，アンロードするコマンドは，それぞれ `insmod`，`rmmod` である．現在カーネルに組み込まれているドライバのリストは，`lsmod` で表示される．`insmod` コマンドによりカーネルに組み込まれる際には，トップハーフルーチン，ボトムハーフルーチンの登録が行われる．

7.7 デバイスファイル

UNIX 系の OS では，デバイスへのアクセスはデバイスファイルを介する．Linux ではデバイスファイルは，`mknod` コマンドにより作成できる．例えば，`mknod /dev/hello c 42 0` とすると，メジャー番号 42・マイナー番号 0 のキャラクタデバイス`/dev/hello` を作成する．

基本的には，一つのデバイスファイルは一つのデバイスに結び付けられている．同じ種類のデバイスを複数同時に利用する場合，たいていデバイスファイルは複数になるが，デバイスドライバは同じものが使用されるのが一般的である．

7.8 /proc ファイルシステム

Linux では，システムリソースに関する様々な情報を，`/proc/`以下の仮想的なファイルを見ることで知ることができる．`cat /proc/interrupts` とすれば，ハードウェアの割り込み番号とその割り込みを使用するデバイスが一覧表示される．`cat /proc/cpuinfo` とすれば，そのコンピュータの CPU が何なのかがわかる．`ls /proc` として他に観察できるリソースの種類と意味を調べてみよう．

8 課題

課題は，11 月 10 日 (月) 朝までに ikuo-soft3-08@jsk.t.u-tokyo.ac.jp 宛てに提出する．メールのサブジェクトは,「`soft3-kadai:20081027:xxxxx`」(xxxxx は学生証番号) とする．その他，質問・要望・意見などがあれば，遠慮なく ikuo@jsk.t.u-tokyo.ac.jp にメールをください．

8.1 課題 1

Meadow で gdb を使える環境を確認せよ．今週のサンプル (`http://www.jsk.t.u-tokyo.ac.jp/~ikuo/lec/soft3/docs/20081027.zip`) をデバッグしてみよう．まずは，readme.txt を参照してプログラムを実行し，ソースを読んで内容を理解し，そしてバグを見つけよう．必ずしも gdb を用いなくても良い．

hanoi, wordlist のプログラムの問題箇所を示し，修正例を示したレポートを提出すること．

余力のある人は，バッファオーバフローによるセキュリティの問題はどのようなもので，どのように解決されているかにも言及すると良い．

8.2 課題 2

次に挙げる単語で，わからない，あるいは理解が足りないと感じる単語に関し，各自で調査を行い，簡単に説明せよ．調べても良くわからなかった単語は，そのように記載すること．

ディスパッチャ，execv，CreateProcess，PPID，パイプ，共有メモリ，タイムスライス，
排他制御，同期，セマフォ，アトミック，デッドロック，飢餓状態，シグナル，pthread，mutex，条件変数，クリティカルセクション，スレッドセーフ，リエントラント，リーダライタロック，
SMP，クロスバスイッチ[a]，クラスタ・GRID・クラウド，LRU，TLB，TLB ミス，　フラグメンテーション，セグメンテーション，　ヒープ，動的メモリ確保，ごみ集め，
　パイプ，インタプリタ，タイマ，　認証，パスワードファイル，公開鍵・秘密鍵，電子署名，`rw-r--r--`，
　トロイの木馬，スプーフィング，バックドア，バッファオーバーフロー，CERT，VM，i ノード，FAT，
シンボリックリンク，ハードリンク，NFS，samba，　メモリマップド IO，多重割り込み，優先度，
ポーリング，ビジーウェイト，volatile，　リングバッファ，キャラクタデバイス，ブロックデバイス，
ソケット，TCP/IP，ネットワーク階層モデル，
デーモン，サービス，ソケットサーバ，MULTICS，UNIX，POSIX，Solaris，ブートローダ，Linux，
PC-Linux と SH-Linux，カーネルビルド，デストリビューション，GPL，組込みＯＳ，リアルタイムＯＳ，カーネル，モノリシック，マイクロカーネル，HAL[b]

[a] クロスバスイッチ：cross bar switch
[b] HAL: hardware abstraction layer (又は hardware adaptation layer)

8.3 課題 3

図 10，図 11 にある，システムコールを使ったテストプログラムを作って実行してみよ．Win32API も同様にテストしてみよ．Cygwin でコンパイルする方法は，サンプルを参照するとよい．
システムコールを少なくとも 4 つ以上使用するようなサンプルプログラムを作り，メールで送付すること．(POSIX 互換と Win32API それぞれ)