ＯＳに中粒度の実時間性を付加する仕組みとその応用
または、ある日のＴ研究室 ＃４
もしくは、bit の没原稿 ＃１

Presented by 多田 好克 (Tada Yoshikatsu)

コンピュータサイエンス誌 “ bad ” 2000 年７月号 Vol.32, No.7.

1 プロローグ

ある日のＴ研究室[1]。二日酔いの頭をだましながら、Ｔ教授がメールを読んでいる。

Ｔ： なんぢゃ、これは？
Ｂ： 何ですか、それは？
Ｔ： おぉ、Ｂくんか。なんぢゃ、その綺麗な恰好は…
Ａ： お早うございます。
Ｔ： ありゃ。Ａさんまで美しいお姿で… 今日はいったい何があるのぢゃ？
Ａ： 卒業式ですよ、先生。修士の２年間、本当にお世話になりました。
Ｔ： そうか。卒業式か。うんうん、君たちは本当にお世話したぞ。
Ａ： 何を言ってるんですか。
Ｔ： 社会人学生のＤくんの恰好は、いつもとあまり変わらないなぁ。で、Ｃくんは？
Ｂ： 卒業旅行と称して海外に行っています。
Ｔ： 何！ 彼は卒業までに１度は bit に出てみたいと言っていたのに、運の悪い奴ぢゃ。
Ａ，Ｂ： えーっ、また bit の原稿なんですか?
Ｔ： うん。多田くんから変なプログラムが送られて来てのぉ。
Ｂ： 面白そうですねぇ。
Ａ： あっ、私、卒業式に出てきま〜す。
Ｔ： うむ。今回は紙面も少ないし、君たちは付き合う必要はない。ただし、夕方からの呑み会は忘れないようにな。
Ａ，Ｂ，Ｄ： はーい。

[1] Ｔ研究室については bit の 1998 年 10 月号「スレッドの作り方」の記事を参照のこと。

2 *et* と *MRSA* について

というわけで、Ｔ教授は多田くんから送られてきたメールの中身を吟味することにした。メールには、tar して gzip して uuencode したプログラムと以下のような説明とが含まれていた。

◇ ―――――――――――――――――――― ◇

Ｔ先生：

多田です。

ＯＳの外部で Unix システムとは独立に走行するカーネルモードのスレッドを作りました。で、*et*(external thread) と命名しました。可愛いでしょ。*et* を実現する仕組みは *MRSA*(moderate real-time system adapter) と言います。耐性黄色葡萄球菌とは何の関係もありません。フリビ[2]の 2.2.5 で動きます。遊んでみてください。なお、詳細は文献 [1] を見てくださいね。

―― Ｙｏｓｈｉ Ｔａｄａ

◇ ―――――――――――――――――――― ◇

うーむ、相変わらずだのぉ。とつぶやきながら、Ｔ教授はメールで送られてきたプログラムを展開し始めた。

Readme によると、*MRSA* は FreeBSD 2.2.5 に標準の lkm (loadable kernel module) を利用して実現しているらしい。Unix の全システムと対等な立場で *et* を走らせ、メタス

[2] FreeBSD のこと。ちなみに、ポケビはポケットビスケッツ、トラギはトランジスタ技術。

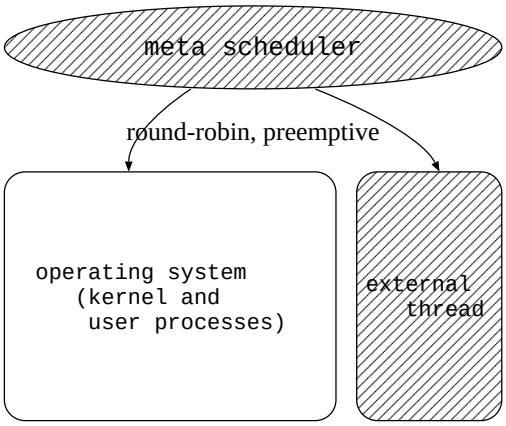

図 1: *MRSA* と *et* の概念図.

ケジューラによって Unix と *et* とを交互に実行する（図 1参照）。*et* は Unix の時分割スケジューリングの対象外となり、20m 秒[3]程度の中粒度の周期性を持って CPU を占有した処理を遂行できる。また、メタスケジューラは CPU 横取りをするので、*et* を使ってサーバのような無限ループの処理を記述することも可能である。

図の斜線部分が *MRSA* だな。その一部分で *et* が動いているという... うーむ、分かったような分からないような。

3 *et* の利用者インタフェース

et の利用者は、*et* のプログラムを書くことになる。*et* のプログラムは Unix のカーネルモードで動作する。そのため、通常のＣライブラリ (libc.a) を呼び出すことはできない。一方、カーネルのデータにアクセスしたり、カーネルプログラムの関数を呼び出したりすることはできる。このことを除けば、*et* のプログラムは通常のＣのプログラムと変わらない。

ただし、プログラムの実行は main() ではなく et_main() という関数から始まる。この関数が終了すれば *et* の実行も終了する。また、et_exit() を呼び出して、*et* の実行を陽に終えることもできる。

Unix と *et* とを交互に動かす間隔は、記号定数 UNIX_SLICE と ET_SLICE を定義して指定する。この単位はチック (tick) と呼ばれる時間間隔であり、現在のフリビでは 10m 秒である。ただし、Unix のカーネルを書き換えれば１ m 秒程度まで短くすることが可能である。

et の実行がが ET_SLICE チックを使い切ると、CPU 横取りが起こり、Unix の実行が再開される。そして UNIX_SLICE チック後には、再び *et* の実行が再開する。*et* が et_yield() を呼び出すと *et* の実行は ET_SLICE を使い切る前に一時中断する。この場合も UNIX_SLICE チックが経過すると、再び *et* の実行が再開する。

et_lock() は *et* に対する CPU 横取りを禁止する。ただし、*et* が制御を確保し続けると Unix カーネルに不都合が生じる。et_lock() を呼び出した場合は、短期間の内に et_unlock() を呼び出し、CPU 横取りを可能にしなければならない。

上述の関数の他に *MRSA* では Unix との同期や入出力のための関数を用意している。まず、p_sleep() と p_wakeup() は Unix カーネルと同期を取るための関数である。これらの関数には、Unix の sleep(), wakeup() と同様に、同期を取る事象を識別するための適切な変数の番地を引数として与える。

p_putchar(), p_printf() は *et* から Unix コンソールへの出力関数である。また、p_write(), p_writec(), p_read(), p_readc() は Unix のプロセスと情報のやり取りをする。Unix プロセス側ではデバイス/dev/etio をオープンし、read, write することによりデータの授受ができる。ちなみに、p_wrtec(), p_readc() は１文字の受渡しをする。なお、これらの関数はランデブーを基本とし、相手がデータの読み書きをするまでブロックする。

4 *et* の応用プログラム

ふーっ。パイプをくゆらしながら、Ｔ教授は長いため息をついた。面白そうだが、いったい何に使うのだろう。usr という名のディレ

[3]フリビのカーネルの記号定数を変更すれば、数ｍ秒程度までの、より細かい周期性も実現できる。

クトリにいくつかの応用プログラムがあったので、それを make することにした。

make してみると、proc_mod.o, map_mod.o, io_mod.o というオブジェクトが作られた。これらは、それぞれ、Unix のプロセスの状態を可視化する *et* プログラム、あるプロセスの仮想アドレスのマッピング情報を可視化する *et* プログラム、そして、プロセスから直接入出力装置を制御するための *et* プログラムである。

なるほど、*et* は Unix とは独立に作動するので、Unix の各種状態をプローブ効果なしに観測できるわけだな。そうすると、*et* を使って Unix カーネルの各種パラメータを動的にチューニングすることもできるのか。io_mod.o は入出力装置を直接制御するという魂胆だな。デバイスドライバを書かなくても、*et* ではある程度の実時間性を実現できるので、好都合というわけだ。

ふーっと、もう一度パイプの煙を漂わせ、T 教授は proc_mod.o を実行させることにした。

proc_mod.o は lkm のモジュールぢゃな。ということは、modload コマンドを使ってこれをカーネル内に押し込んでと...

```
% modload -u -o /tmp/et_mod
          -e et_init proc_mod.o
```

これで、*et* が動き始めたわけだな。次に *et* からの情報を取り出すプロセス proc を動かせば良い... proc は/dev/etio をオープンして read する通常の Unix プログラムだから、そのまま

```
% proc
```

と打てば良いのぢゃな。

おぉ、こ、これは...（と、画面には突然図2のウィンドウが現れた。）

画面には上のようなウィンドウが現れ、個々の小さな四角が緑や黄、赤に点滅している。色は Unix プロセスの状態を表し、緑が実行中、黄が実行可能、赤がサスペンド状態とのこと。また、図で黒く見えるのは実は青色で、事象を待って休眠中のプロセスらしい。

うーむ、カラーで見えない bit の読者が可哀相ぢゃ。美味しそうにパイプを吸いながら、T 教授はつぶやいた。*et* が Unix カーネル内の情報を調べ、それを書き出す。Unix プロセスはデバイスドライバから情報を読み出し、X ウィンドウに描画するというわけぢゃな。

どれ、少しプログラムを見ておくか。

```
struct dat {
    int     d_pid;
    char    d_stat;
} d_buf[MAXBUF];
```

というデータ構造に個々のプロセスの状態を集めるのが、以下の *et* プログラムぢゃな。

```
void et_main() {
  struct proc     *p;
  int             i;
  while (1) {
    i = 0;
    for (p=allproc.lh_first; p!=0;
         p=p->p_list.le_next) {
      d_buf[i].d_pid = p->p_pid;
      d_buf[i].d_stat = p->p_stat;
      if (i++ >= MAXBUF)
        break;
    }
    p_write(d_buf,
         sizeof(struct dat) * i);
  }
}
```

単にプロセスのリストを嘗めて d_buf に情報を集め、それをデバイスドライバに書き出しているだけだな。プログラムは無限ループだが、p_write() がブロックするし、CPU 横取りも起きるから問題無いということか...

で、Unix プロセスの方は、我々の良く知っている普通のプログラムだな。

```
#define BUFSIZE sizeof(struct dat)

main() {
  int fd;
  int i, j;
  if ((fd=open("/dev/etio",
              O_RDONLY)) == -1)
```

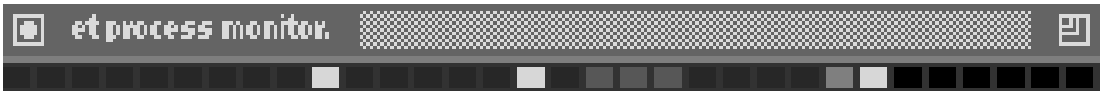

図 2: Unix プロセスの実行状態の可視化

```
    exit(1);
  while (1) {
    i = read(fd, d_buf,
             BUFSIZE * MAXBUF);
    i /= BUFSIZE;
    /* d_buf の値を使った描画 */
  };
  close(fd); /* 実行されない */
}
```

これまた単純だなぁ。多田くんの書くプログラムはいつもこうぢゃ。要するに/dev/etio から d_buf に情報を読み出す。で、その情報を使って描画してるだけだ。このプログラムも無限ループだが、read がブロックするので *et* と同期が取れるのだな。

ps コマンドを使うと、ps コマンドを実行しているプロセスが常に実行中になるが、*et* を使って情報の収集をやると Unix の自然な姿が観測できるのぢゃな。それに、描画は Unix のプロセスでやるから、プログラムも簡単と。こりゃ、なかなか便利ぢゃな。

仮想アドレスのマッピング情報を見るプログラムも似ているな。*et* ではカーネル関数が利用できるので、マップの情報を得るのも簡単なこと。100m 秒の間隔で情報を取得しているが、もし、描画処理が間に合わなくても入出力がランデブーなので OK ということか…

入出力装置の制御プログラムは便利そうだなぁ。Windows にはソフトウェアでタイミングを取る入出力装置も多いが、*et* を使えば Unix でも似たようなことができそうぢゃ。*et* を使ってロボットの動きを制御し、かつ、動作のプランニングには Unix プロセスを使うといった応用にも使えるかもしれない。うーむ。もう少し付き合ってみるか。

5 *MRSA* の仕組み

T 教授は、*et* を実現している *MRSA* のプログラムを調べることにした。Readme にはフリビの lkm を利用していると書いてあったが、それならばモジュールの取込み時に初期化をやっているはずだな。どれどれ…

```
et_load(struct lkm_table *lkmtp,
        int cmd) {
  extern int  etio_begin();
  int         err;
  err = etio_begin();
  stack = (unsigned int)
      kmem_alloc(kmem_map, s_size);
  sp = (int *)((stack+s_size) & ~3);
  *--sp = (int)et_exit;
  et_buf[R_ESP] = (int)--sp;
  et_buf[R_EIP] = (int)et_main;
  timeout(to_et, 0, UNIX_SLICE);
  return(err);
}
```

ふむ。思った通りぢゃ。etio_begin() によってデバイスドライバの準備をして、後は *et* のコンテクスト作りぢゃな。カーネル関数の kmem_alloc() を使って *et* のスタックを確保し、et_buf に *et* の初期コンテクストを格納すると… 多田くんのことだから、おそらく、et_buf と unix_buf という 2 つのコンテクスト格納領域を使って、*et* と Unix を交互に動かすつもりだろう。してみると、この 2 つのバッファは jmp_buf 型で、コンテクスト切替えには setjmp() と longjmp() を使うのだな [2]。多田くんも進歩がないなぁ[4]。しかし、最後の timeout() は何だ？ これはカーネル関数で、確かマニュアルがあったはずだぞ。

と、T 教授はフリビのマニュアルの timeout(9) を読み始めた。ちなみに、Unix マニュ

[4] 放っといてくれ。

アルのセクション９と言うのは耳慣れないかもしれないが、intro(9) によるとカーネル内の関数のインタフェースなどがまとめられているらしい。

何々、第３引数で指定したチックが経過した後に、第１引数の関数を呼び出してくれるのぢゃな。第２引数は関数への引数だが、今は使わないから関係ないと。ということは、UNIX_SLICE チック後に to_et() という関数を呼び出せと言っているのだな。どうせ、to_et() は unix_buf に Unix のコンテクストを保存して、et_buf から *et* のコンテクストを回復するのぢゃろう。一応、確認しておくか…

```
void to_et() {
  if (setjmp(unix_buf)) {
    /* just return */
  } else {
    timeout(to_unix, 0, ET_SLICE);
    longjmp(et_buf);
  }
}
```

ふっふっふっ、図星だな。なるほど、今度はここで ET_SLICE 後に to_unix() を呼び出すように設定する。to_unix() は to_et() と同じようなものだから見るまでもなかろう。

どれ、呑みに行くかな。Ａさん達はまだかな？

6　RT-Linux との比較

Ａさん、Ｂくん、Ｄくんが卒業式から帰ってくるのを待つ間に、Ｔ教授の元にもう１通のメールが届いた。

◇ ―――――――――――――――――――――――――― ◇

Ｔ先生：

多田です。

MRSA と良く似たシステムに Real-Time Linux と言うのがあります。有名なシステムなので、Web で捜せばすぐに見つかると思います。文献 [3][4] もあります。bit に記事を書くなら、比較しておいてください。

－－ Ｙｏｓｈｉ Ｔａｄａ

◇ ―――――――――――――――――――――――――― ◇

比較しておいてくださいだとぉ！ どこにそんな暇があると言うのぢゃ。と思ったが、取り敢えず卒業式が終わるまでは暇か… どれ、少し調べてみよう。

何々、RT-Linux も *MRSA* と同じように Unix のカーネル外でカーネルモードのスレッドを走らせるのか。違いはどこにあるのぢゃ？ ふむふむ、RT-Linux では制御の獲得をタイマ割込みで実現しているのか。*MRSA* よりも粒度の細かな制御ができると言うことか。しかし、プログラミングは大変そうぢゃのぉ。Unix カーネルとの排他制御も考えなければならないぞ。

だいたい、以下のような比較で良いかな（表1参照）。どれ、では、この表を多田くんにメールしておこう。

さて、これで今日の仕事は終わりぢゃ。無事 bit の原稿も書けたし、そろそろＡさん達も帰って来る頃だし、ぼちぼち呑みに行くことにするかな。

7　エピローグ

Ｔ教授が呑みに行ってしまったので、*et* と *MRSA* に関する話はこれで終わり。最後に、話を少しまとめて、この記事も終わることにする。

まず、*MRSA* を利用することにより、稼働中のカーネルに新しい機能を追加できる。また、あらかじめカーネルをモジュール化して作っておくことにより、必要な機能を追加したり不要な部分を取り外したりして、用途に合ったカーネルを動的に構築することも可能となる。

このようなカーネルの発展性は、マイクロカーネルとサーバ群による構成でも実現されてきた。しかし、サーバ間の通信にコンテクスト切替えが必要なことから、十分な処理速度を確保しにくい [5]。他方 lkm を使った方法では、単層カーネル (monolithic kernel) に機能を追加するので速度の問題はない。しかし、

表 1: *MRSA* と Real-Time Linux との比較.

比較項目	*MRSA*	RT-Linux
制御の獲得法	timeout ルーチン	タイマ割込み
最短起動周期	20 m秒*	数 10 μ秒
外部事象への対応	20 m秒*＋数 100 μ秒	＜ 20 μ秒
CPU 横取り	あり	なし
カーネル関数	利用可	利用不可
カーネルとの排他制御	不要	必要
デバイスドライバ	ランデブー	ブロック不可
プログラミング	比較的容易	難しい
割込み処理	オーバヘッドなし	かなりのオーバヘッド
システムの導入	簡単	ドライバの書換えが必要

*) Unix カーネルの簡単な変更により、数 m 秒程度まで改善可能。

モジュールのプログラムに虫があると、ＯＳ全体が脱線転覆するという問題がある [6]。

また、lkm ではシステムコールやファイル操作、デバイスドライバなど、制御の取得点が決っており、自発的に処理を開始するプログラムを記述しにくかった。*MRSA* はこの欠点を補っていると考えることができる。既に見てきたように、*et* のプログラミングにおいては無限ループを書くことも可能である。

et のプログラムは、短い時間間隔で周期的に実行される。いわゆる実時間処理ではないが、動画や音声のマルチメディア処理、ロボットの制御などに利用できるのではないかと考えている。中粒度の実時間処理を実行できる一方で、我々が馴れ親しんだ Unix の機能をそのまま使えることのメリットは大きい。プログラムも比較的簡単である。

今後は、*MRSA* と *et* を利用したシステムの記述に挑戦したいと考えている。

参考文献

[1] 多田好克, 中村嘉志: *MRSA*: Unix 上の中粒度実時間処理系, 信学技報, CPSY99–123, pp. 1–8 （2000 年 3 月）.

[2] 多田好克, 寺田実: 移植性・拡張性に優れたＣのコルーチンライブラリ実現法, 信学会論文誌, J73–D–1, 12, pp. 961–970（1990 年 12 月）.

[3] Barabanov, M., and V. Yodaiken: Introducing Real-Time Linux, Linux Journal, pp. 19–23 (Feb., 1997).

[4] 船木陸議： RT-Linux の構造と実装の詳細, インターフェース, 25, 11, pp. 88–100, CQ 出版社（1999 年 11 月）.

[5] Liedtke, Jochen: Toward Real Microkernels, CACM, 39, 9, pp. 70–77 (Sep., 1996).（谷口秀夫訳： 真のマイクロカーネルへ向けて, bit, 29, 8, pp. 11–21, 共立出版（1997 年 9 月）.）

[6] 鈴鹿倫之, 中村嘉志, 多田好克： カーネル拡張のための効率的な開発環境, 情報処理学会第 41 回プログラミング・シンポジウム報告集, pp. 57–64 （2000 年 1 月）.