現所属：九州工業大学大学院生命体工学研究科
ishii@brain.kyutech.ac.jp

九州工業大学情報技術セミナー資料
1998/8/17 石井和男

RT-Linux

1. RT-Linux とは？

身の回りにもマイコン製品が多くなっていると思います。CDプレーヤやノートパソコン、テレビゲーム機等にもマイコンが搭載されています。このような身近なマイコン製品には、リアルタイム OS を搭載したものもあります。例えば、

- プロセス制御
- 自動車
- エアバス
- FAX
- エレベータ
- 自動改札機
- 電子レンジ

等にも使用されています。エアバスにはVxWorksという製品が搭載されています。OSにはWindows3.1/95やMacOS8、OS/2等がありますが、これらのOSはリアルタイムOSではありません。これらのOSでは、処理毎のの優先順位という概念を持たず、実行させているプログラムにはまんべんなくCPUサイクルを与えてしまいます。しかし、組み込みシステム等の、時間にクリティカルなシステムでは、重要な仕事は再優先で処理して欲しい場合がほとんどです。

リアルタイムOSでは、ひとつひとつの処理に優先度をつけることができます。また、処理と処理の間に時間経過が必要な場合の時間管理を行います。DOSでのプログラミングのように、単一のプログラムがCPUを独占する場合は、リアルタイムOSを使うまでもありません。しかし、CPUの性能が向上し、複数のプログラムを一つのCPUで処理するには、プログラムの優先度、時間管理ができるリアルタイムOSが必要となります。主なリアルタイムOSとして、以下のものがあります。

(1) OS-9　マイクロウェア・システムズ社

- マルチタスク・マルチユーザ
- UNIX ライクなプロセス、I/O モデル
- 100%ROM 化が可能
- CPU のアーキテクチャに依存しない構成

(2) VxWorks　　WindRiver System 社

- マルチタスク・マルチユーザ
- 開発環境が UNIX
- CPU のアーキテクチャに依存しない構成
- UNIX ネットワークを利用した分散開発環境

・サブルーチンが充実していて、目的の機能をサブルーチンの組み合わせとして達成可能

(3) Real/32　　リニア・テクノロジー社

・マルチタスク・マルチユーザ

・MS-DOS 互換。DOS の資源を活用できる。

・UNIX 以外にも、WindowsNT,NetWare との接続が簡単

これらはリアルタイム OS として実績があるが、非常に高価です。

では、本セミナーで取りあげる Linux はどうでしょうか？ Linux はマルチユーザ、マルチタスクのシステムであり、プログラムのスケジューリングには、TSS（タイムシェアリング）、つまりプログラムにCPUを平均的に分配しようとします。コンピュータを汎用的に利用する場合は効果的ですが、リアルタイムの処理には適していません。

そこで、Michael Baraonov、Victor Yodaiken によって Linux をリアルタイム OS に拡張されたものが、RT-Linux なのです[1]。RT-Linux では、Linux はリアルタイム・タスクがないときに動作する１つのタスクとして取り扱われ、リアルタイム・タスクが CPU を必要としたときは、横取りするように設計されています。図１に RT-Linux の構造を示します。I/O や割り込み等は、RT-Process が取り扱い、RT カーネルを通して Linux のプロセスと通信します。RT-Linux の特徴は、

・マルチタスク・マルチユーザ

・Linux の資源が活用できる

・開発環境とターゲットシステムが同じ

・ネットワークへのアクセスが容易

・フリーウェアであり、ソースコードが公開されている

などが挙げられます。

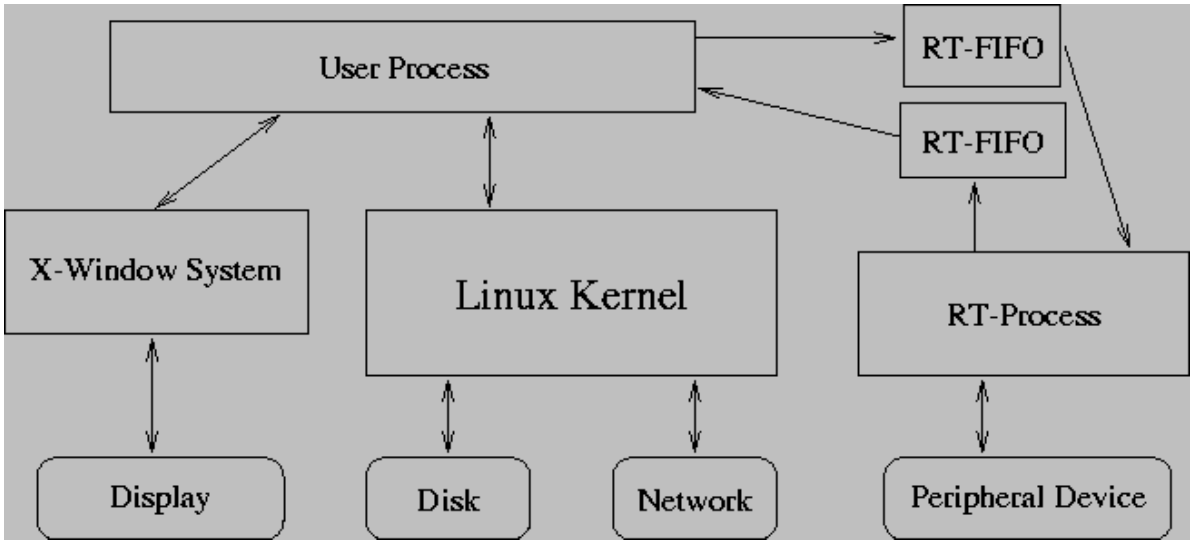

Fig.1 RT-Linux の構造

[1] Michael Baranov, Victor Yodaiken, http://luz.cs.nmt.edu/~rtlinux

2 RT-Linux のインストール

RT-Linuxをインストールするには、linuxのカーネルのソースコードにRT-Linuxのパッチをあてる必要があります。RT-Linuxのバージョンは、現在（1998年8月）でVersion 0.6であり、Linux のカーネル 2.0.33 に対応しています。従って、

・linux-2.0.33.tar.gz

・rtlinux-0.6-2.0.33.tgz

が必要になります。まずこれらをダウンロードしましょう（tar.gz と tgz は同じ意味）。keys.ces.kyutech.ac.jp に用意してありますので、

```
>ftp    keys.ces.kyutech.ac.jp
```

としてください。するとユーザ名、パスワードを聞かれますので、

ユーザ名　　ftp

パスワード　　Email アドレス

を入力します。ログインできたら、

```
>cd pub/linux
>bin
>get  linux-2.0.33.tar.gz
>get rtlinux-0.6-2.0.33.tgz
>quit
```

とし、自分のコンピュータにダウンロードします。これらのファイルは、常時 keys に置いてありますので、ご自由に利用してください。

次にこれらのファイルを /usr/src に移動して解凍します。

```
>cd /usr/src
>tar xvfz linux-2.0.33.tar.gz
>tar xvfz rtlinux-0.6-2.0.33.tgz
```

すると、/usr/src/linux にlinuxのカーネルのソースコード、/usr/src/rtlinux-2.0.33 にrtlinuxのソースコードが展開されます。

次に linux のカーネルのソースコードにパッチをあてましょう。

ディレクトリ　/usr/src/linux　において

```
> patch -p1 < ../rtlinux-0.6-2.0.33/kernel_patch
```

と入力します。無事終了したら、カーネルをコンパイルします。

```
>make menuconfig
>make dep
>make clean
>make zlilo
```

RT-Linux はカーネルのモジュールとして提供されるので、さらに、カーネルのモジュールもコンパイルします。

```
>make modules
>make modules_install
```

すると、ディレクトリ　/lib/modules/2.0.33　以下にRT-Linuxのカーネルモジュールがインストールされます。カーネルモジュールとは、カーネルの一部であり、必要な場合にインストールできるものです。

以上でRT-Linuxのインストールが終了しました。コンピュータを再起動しましょう。

3. RT-Linuxを起動する

実際にRT-Linuxを使ってみましょう。コンピュータにrootでログインし、RT-LinuxのAPI(Application Program Interface)を使えるようにするには、先ほどコンパイルしたモジュールをメモリーにロードする必要があります。

```
>cd /lib/modules/2.0.33/fs
>insmod rt_fifo_new.o
>cd ../misc
>insmod rt_prio_sched.o
```

これでRT-Linuxの関数が使えるようになりました。モジュールがロードされたか確認するには

```
>lsmod
```

を使います。このように、コンピュータを起動するたびにinsmodを使ってモジュールをロードする必要があります。コンピュータが起動したときに、自動的にモジュールをロードするように設定するには、

```
/etc/rc.d/rc.modules
```

というファイルに、

```
/sbin/modprobe rt_prio_sched.o
/sbin/modprobe rt_fifo_new.o
```

を付け加えます。

モジュールを取り除くには、

```
>rmmod　モジュール名
```

とします。 これで、RT-Linuxの関数が使えるようになりました。また、後ほどRT-FIFOというリアルタイムプロセスと通常のプロセスが通信するためのバッファが必要になるので以下のファイルを作成し、実行します。

ファイル名：rtf.sh

```
#sh
for i in 0 1 2 3; do mknod /dev/rtf$i c 63 $i; done
```

このファイルを実行するには、

```
>sh rtf.sh
```

を入力します。これで/dev/rtf0から/dev/rtf3までの４つのRT-FIFOができました。

4. RT-Linuxを使ってみる

いよいよ、RT-Linuxでプログラミングしましょう。例を挙げて説明します。リアルタイムでデータをポーリングし、データをファイルに書き込むアプリケーションが必要だとします。RT-Linuxでの重要な設計方針は次のようになります。

リアルタイム・プログラムは小さくシンプルなパーツに分解し、ハードなリアルタイム処理が必要な部分とします。残りの部分は、通常のLinuxのアプリケーションとします。

この基本原則に従えば、アプリケーションを２つの部分に分割することになります。ハードなリアルタイム処理が必要な部分はリアルタイムタスクとして実行され、デバイスからRT-FIFOという専用のI/Oインターフェイスへデータをコピーします。プログラムのメインの部分は、普通のLinuxプロセスとして実行されます。この部分はRT-FIFOからデータを読みだし、画面に表示すると共にファイルへ記録します。

リアルタイム部はカーネル・モジュールとして記述する必要があります。Linuxでは、リブートせずにカーネルモジュールをコンパイルしロードすることができます。以下にリアルタイム部のサンプルプログラムを示します。

（ファイル名：rt_process.c の先頭部分）

```
#define MODULE

#include <linux/module.h>
#include <linux/rt_sched.h>
#include <linux/rtf.h>
RT_TASK mytask;
```

モジュールとして記述するので、一行目にMODULEを定義し、module.hをインクルードしています。その後に、リアルタイム・ヘッダ rt_sched.h と rtf.h をインクルードし、RT_TASK構造体を宣言します。ここではリアルタイムタスクをmytaskとしています。リアルタイム構造体には、コードへのポインタ、データ、タスクのスケジューリング情報があります。リアルタイムタスク構造体は、rt_sched.h に定義されています。リアルタイムタスクが通常の Linux プロセスと通信するには、RT-FIFO を用いる方法と、Shared Memory を使う方法があります。ここでは RT-FIFO を用いる方法を示しています。

```
void mainloop(int fifodesc)
{
        static int data=0;
        while(1) {
                data++;
                rtf_put(fifodesc,(char*)&data, sizeof(data));
                rt_task_wait();
        }
}
```

このmailloopという関数がリアルタイム部の仕事の中味です。引数のfifodescはRT-FIFOの番号が入ります。mainloopがコールされる毎に、dataが１だけインクリメントされます。その後rft_put関数を使って、RT-FIFOにdataを書き出します。rtf_put関数は第１引数として、RT-FIFOの番号、第２引数に送るデータのポインタ、第３引数にデータの大きさをバイト数で記述します。次にrt_task_wait関数を呼び出し、CPUを次に呼び出されるまで開放しています。

モジュールには初期化ルーチンが必要となります。プログラム例の初期化ルーチンでは、

・現在時刻を記録する
・リアルタイムタスク・構造体を初期化する
・タスクをピリオディック・スケジューリングに組み込む

という作業を行っています。init_moduleという名前は、決められた名前であり、モジュールがロードされると最初にコールされます。

```
int init_module(void)
{
	#define RTfifoDESC	0

	RTIME now= rt_get_time();
	rtf_create(RTfifoDESC,1000);
	rt_task_init(&mytask, mainloop, RTfifoDESC, 3000, 4);

	rt_task_make_periodic(&mytask, now+1000, RT_TICKS_PER_SEC);
	return 0;
}
```

RTIME変数として宣言されたnowにrt_get_time関数を用いて現時点でのタイマーの値を入力します。rtf_create関数でRT_FIFOを使えるようにしています。最初の引数は、RT_FIFOの番号であり、ここでは０を入力し、/dev/rtf0を使うことを示しています。２つめの変数はRT-FIFOのバッファの大きさであり、1000バイトの大きさであることを示しています。

rt_task_init関数は、リアルタイム・タスク構造体を初期化し、タスクへの引数を用意します。この例では、mytaskに対して、実行されるコードがmainloopであること、mainloopの引数がRTfifoDESC、つまり０番目のRT-FIFOであること、タスクのスタックサイズは3000バイト、優先度は４であることを示しています。

rt_task_periodic関数は、新しいタスクをピリオディック・スケジューリング・キューに組み入れます。ピリオディック・スケジューリングというのは、一定時間毎にタスクが稼働状態になるようにスケジューリングされることです。この例では、mytaskを現時刻から1000単位時間後から、RT_TICKS_PER_SEC毎に、つまり１秒後毎に起動します。

RT_TICKS_PER_SECは既に定義されている値であり、一秒あたりのタイマーのカウント数です。従って、0.5秒おきに起動したい場合は、

RT_TICKS_PER_SEC*0.5

とします。

Linuxでは、モジュール全てにクリーンナップする関数が必要です。リアルタイム・タスクでは、無効なタスクがスケジューリングされていないことを保障する必要があります。cleanup_moduleという関数の中にタスク及びRT-FIFOを開放するコードを記述します。この例では、mytaskとRT-FIFOの0番を開放しています。cleanup_moduleは決められた名前です。

```
void cleanup_module(void)
{
        rt_task_delete(&mytask);
        rtf_destroy(RTfifoDESC);
}
```

以上でリアルタイム部のプログラムは終わりです。残りの部分を実行するプログラムも必要です。これは普通のLinuxプロセスとして実行されます。以下にプログラム例を示します。

このプログラムでは、RT-FIFOからデータを読みだし、stdoutへ、つまり画面上へデータを書き出します。

（ファイル名：app1.cの先頭部分）

```
#include <stdio.h>
#include <errno.h>
#include <sys/time.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
```

必要なヘッダファイルをインクルードしています。おまじないのようなものと思ってもいいでしょう。次に、内容を示します。

```
#define BUFSIZE 10
int buf[BUFSIZE];
```

まず、リアルタイム部から送られてくるデータとして、bufというint型の変数を宣言しています。

```
int main()
{
        fd_set rfds;
        struct timeval tv;
        int retval;
        int fd0;
        int n;
        int i,j;
```

fd_set 変数として rfds を宣言しています。この変数に、RT-FIFO にあたる部分のビットを設定します。timeval 構造体として tv を宣言し、tv にある時間を書き込むことにより、設定した時間毎に RT-FIFO を調べます。

```
        if ((fd0 = open("/dev/rtf0", O_RDONLY)) < 0) {
                fprintf(stderr, "Error opening /dev/rtf0\n");
                exit(1);
        }
```

open 関数を使って、RT-FIFO の 0 番、/dev/rtf0 を読み込みモードでオープンしています。オープンに成功した場合、ファイルディスクリプタの値が fd0 に返ります。

```
        for (i = 0; i < 100; i++) {
                FD_ZERO(&rfds);
                FD_SET(fd0, &rfds);
```

最初に fd_set 変数の rfds を FD_ZERO 関数を用いて初期化しています。次に、FD_SET 関数を用い、先ほどオープンしたファイルディスクリプタfd0に該当する部分のビットを立てています。

```
                tv.tv_sec = 1;
                tv.tv_usec = 0;

                retval = select(FD_SETSIZE, &rfds, NULL, NULL, &tv);
                printf("retval=%d\n",retval);
```

tv に時間をセットしています。この例では、一秒をセットしています。次に select 関数を用い、該当するファイルディスクリプタにデータが入っているかどうか調べています。tv に一秒がセットされているので、select 関数により、一秒間だけデータが入るまで待ち状態になりますファイルディスクリプタにデータがある場合は、１が返ります。データがない場合は０が返されます。ここで、FD_SETSIZE は 1024 という数字が入っています。

```
            if (retval > 0) {
                if (FD_ISSET(fd0, &rfds)) {
                    n = read(fd0, (char*)&buf, BUFSIZE*sizeof(int));
                    for(j=0;j<10;j++)
                        printf("FIFO 0: %d\n", buf[j]);
                    printf("\n");
                }
            }
        }
        return 0;
}
```

select関数では、すべてのファイルディスクリプタを調べます。ここでは、RT-FIFOに該当するディスクリプタfd0に該当する部分にデータが入力されているかどうか、FD_ISSET関数を用いて調べます。

fd0にデータが入力されている場合、read関数を用いてデータを読み込みます。read関数の引数は、読み込み先のディスクリプタ、データを読み込むバッファ、データのサイズとなっています。その結果をprintf関数により表示します。

以上でプログラムは終了です。この二つのプログラムをコンパイルするためのMakeファイルは、次のようになります。

（ファイル名：Makefile）

```
all: rt_process.o app1

RTLINUX = /usr/src/linux
INCLUDE = ${RTLINUX}/include

CFLAGS = -O2 -Wall

app1: app1.c
        gcc -I${INCLUDE} ${CFLAGS} -o app1 app1.c

rt_process.o: rt_process.c
        gcc -I${INCLUDE} ${CFLAGS} -D__KERNEL__ -D__RT__  -c rt_process.c

clean:
        rm -f app1 rt_process.o
```

プログラムがあるディレクトリから

```
>make
```

と入力しましょう。コンパイルされます。プログラムを実行するには、

```
>insmod rt_process.o
>app1
```

と入力します。画面上に、RT-FIFO に入力された数字が表示されるはずです。

5 RT-Linux の API

RT-Linux に用意されている関数を以下に示します。

SYNOPSIS

```
#define MODULE
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/version.h>
#include <linux/errno.h>
#include <linux/rtf.h>
```

```
int free_RTirq(unsigned int irq);
```

request_RTirq とセットです。割り込みを開放します。

```
int request_RTirq(unsigned int irq, void (*handler)(void));
```

irq の割り込みが発生したときに、関数 handler()を実行します。

```
RTIME rt_get_time(void);
```

現在のタイマーの値を返します。

```
int rt_task_delete(RT_TASK * task);
```

リアルタイム・タスクを消去します。

```
int rt_task_init(RT_TASK * task, void (*fn)(int data), int data, int stack_size, int priority);
```

リアルタイム・タスクを初期化します。

```
int rt_task_make_periodic(RT_TASK *task, RTIME start_time, RTIME period);
```

リアルタイム・タスクを周期的に呼び出します。

```
int rt_task_suspend(RT_TASK *task);
```

リアルタイム・タスクを一時停止します。

int rt_task_wait(void);
rt_task_make_priodic によってスケジューリングされたリアルタイム・タスクにおいて CPU を開放します。

int rt_task_wakeup(RT_TASK *task);
suspend されたリアルタイム・タスクを起動します。

void rt_use_fp(int allow);
リアルタイム・タスクにおいて、浮動小数点を使えるようにします。

int rtf_create(unsigned int fifo, int size);
RT-FIFO を生成します。

int rtf_create_handler(unsigned int fifo, int (* handler)(unsigned int fifo));
RT-FIFO にデータが書き込まれた場合に、関数 handler を起動します。

int rtf_destroy(unsigned int fifo);
RT-FIFO を消去します。

int rtf_get(unsigned int fifo, char * buf, int count);
RT-FIFO からデータを読み込みます。

int rtf_put(unsigned int fifo, char * buf, int count);
RT-FIFO にデータを書き込みます。

int rtf_resize(unsigned int fifo, int size);
RT-FIFO のバッファの大きさを変更します。

6. 他のサンプルプログラム

(1)プリンタポートにデータを書き込むプログラム

```
#define MODULE
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/version.h>
#include <asm/io.h>
```

I/O ポートを扱うのでインクルードします。

```
#include <linux/rt_sched.h>
#include <linux/rt_time.h>

#define LPT 0x3BC
```

プリンタポートのアドレスです。自分のマシンに合わせて変える必要があります。

```
RT_TASK mytask;
RT_TASK mytask2;

void fun(int t) {
        while (1) {
                outb(t, LPT);           /* write on the parallel port */
                rt_task_wait();
        }
}
```

t には 1 か 0 が入ります。従って、1 ビット目が High と Low を繰り返します。

```
int init_module(void)
{
        RTIME now = rt_get_time();

        rt_task_init(&mytask, fun, 1, 3000, 4);
```

mytask はプリンタポートに 1 を書き込むタスクです。

```
        rt_task_init(&mytask2, fun, 0, 3000, 5);
```

mytask2 はプリンタポートに 0 を書き込むタスクです。

```
        rt_task_make_periodic(&mytask, now, RT_TICKS_PER_SEC*0.02);
        rt_task_make_periodic(&mytask2, now+0.005*RT_TICKS_PER_SEC, RT_TICKS_PER_SEC*0.02);
```

周期 0.02 秒、その内 0.005 秒間だけビットが High になります。

```
        return 0;
}
```

```
void cleanup_module(void)
{
        rt_task_delete(&mytask);
        rt_task_delete(&mytask2);
}
```

タスクを消去しています。これは rmmod の際に実行されます。

(2)浮動小数点を扱うプログラム

```
#define MODULE
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/version.h>
#include <asm/io.h>

#include <linux/rt_sched.h>
#include <math.h>
```

数学関数を扱うために、math.h をインクルードしています。

```
#define LPT 0x3BC

RT_TASK mytask;

void fun(int t) {
        int output;
        double sine;

        rt_use_fp(1);
```

浮動小数点を扱えるようにしています。

```
        while (1) {
                sine = sin((double)(rt_get_time())/10000.0);
                output = (sine > 0) ? 1 : 0;
                outb(output, LPT);
                rt_task_wait();
        }
}
```

```
int init_module(void)
{
        RTIME now = rt_get_time();

        rt_task_init(&mytask, fun, 1, 3000, 4);

        rt_task_make_periodic(&mytask, now + 1000, 1000);
        return 0;
}


void cleanup_module(void)
{
        rt_task_delete(&mytask);
}
```