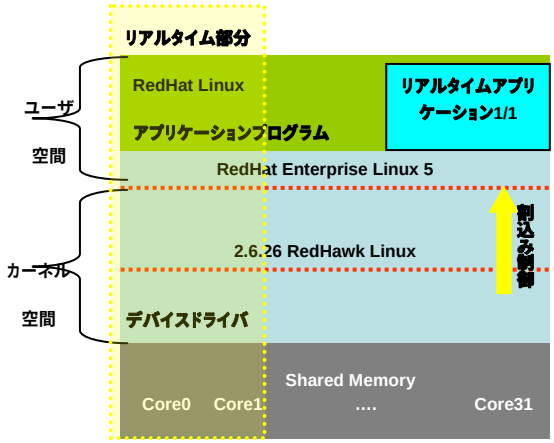
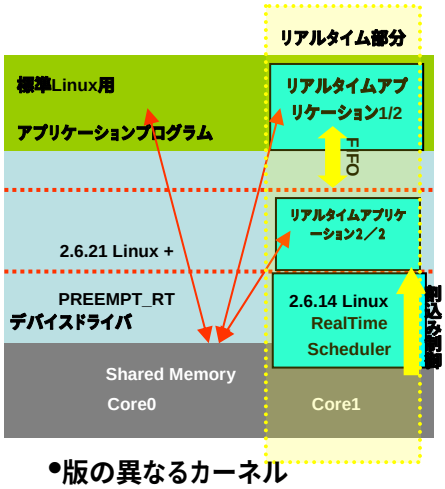
RTLinuxからRedHawk Linuxへの移植ガイド

リアルタイムカーネルのアーキテクチャ

RedHawk Linux は、シングルカーネルを採用しているため、通常のアプリケーションの作成と、リアルタイムアプリケーションに差はありませんが、RTLinuxは、マルチカーネルを作用しているため、リアルタイムアプリケーションは、カーネル内で実行するように作成し、FIFOを使って通信するように作成しなければなりません。





- ただ1つのカーネル
- POSIXプログラムモデル(RedHat互換)
- 全てのアプリケーションがリアルタイムになります (ドライバ、ネットワーク、グラフィクス)

- 版の異なるカーネル
- 異なるプログラムモデル
- 全てのアプリケーションがリアルタイムにはなりません (ネットワークやグラフィクス)
- 標準Linuxを低優先度スレッドとして実行します

このため、RT Linuxで、作成されているアプリケーションプログラムを移植するには、２つの方法が考えられます。

1. RT Linuxのリアルタイムアプリケーション1/2+2/2のパートを一体になるように書き直す。
2. RT Linuxの構造を模擬する。

１の方法は、移植の手間がかかりますが、より高速になり、２の方法は能力が１の方法に比べて劣り、エミュレートライブラリを新規に開発する必要があります。

また、RTLinuxのアプリケーションプログラムは、pthreadベースの関数で作成されているため、

移植にはpthreadを利用することになると考えられます。しかし、リアルタイム拡張規格であるPOSIX1003.1bは、シングルタスに対する規格であるため、Process/PthreadモデルでPOSIX1b関数を利用するためには、各の定義を正しく理解する必要があります。

標準Linuxのスレッド実装であるNPTL (Native POSIX Threads Library)は、新しい Pthreads の実装でLinux-Threadsと比べると、NPTL は POSIX.1 の要求仕様への準拠の度合いが高く、多数のスレッドを作成した際の性能も高くNPTLはLinux 2.6カーネルに実装されている機能を必要とします。

NPTLでは、一つのプロセスの全てのスレッドは同じスレッド・グループに属し、スレッド・グループの全メンバーは同じProcessIDを共有すします。NPTLは管理スレッド (manager thread) を利用しません。NPTLは内部でリアルタイムシグナルのうち最初の2つの番号を使用していて、これらのシグナルはアプリケーションでは使用できません。さらに、NPTLにもPOSIX.1に準拠していない点があり、NPTLスレッドは同一プロセススレッドで、共通の nice 値を共有しません。

このことは、スレッド個別に優先度を設定する必要がある事を意味します。

RTLinux 関数から POSIX 関数へ

下記に、RTLinux 関数から POSIX 関数へのマッピング表を掲げます。
基本的には、rtl_pthread_xxx()関数は rtl_部分を削除すれば移植が可能ですが、その意味が異なる場合がありますので、注意してください。

RT Linux V1 API	RT Linux V2 API	RedHat/RedHawk/POSIX API
rt_get_time	gethrtime システムが起動してからカウントを始めるリセットや調整が不可能な時間を返してくる関数 周波数は 1193180Hz によって管理している。	clock_gettime参照
	clock_gettime 時間を返してくる関数 周波数は 1193180Hz で時間を管理している gethrtimeだと多少誤差がでてくるため他の方法で行っている。 以下の方法で時間を管理できる。 * CLOCK_REALTIME The standard POSIX clock * CLOCK_8254 The clock that is used on non-SMP x86 systems * CLOCK_APIC The local APIC clock on SMP x86 systems.	#include <time .h> int clock_gettime (clockid_t which_clock, struct timespec *setting) ; gcc [options] file -lccur_rt ... 引数は以下のように定義されます。 which_clock それから時間を取得するクロックにおける識別子。which_clockの値は、CLOCK_REALTIMEまたはCLOCK_MONOTONICです。 setting which_clockの時間が返される構造へのポインタ。 0の返り値はclock_gettimeへの呼び出しが成功したことを示します。-1の返り値はエラーが発生したことを示します。errnoはエラーを示すように設定されています。起こり得るエラーの種類のリストについてはclock_gettime (2) man pageを参照してください。
rt_task_init	pthread_create	同じ
rt_task_delete	pthread_delete_np	なし
rt_task_make_periodic	pthread_make_periodic_np	なし Posix timerを使用しなければならないが、timerは、プロセスに対してシグナルを発生

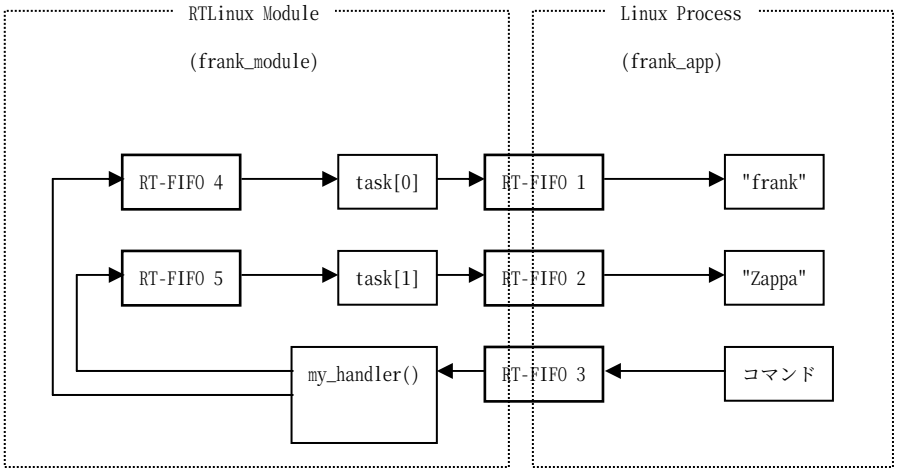
		させるので、pthreadに対して正しくシグナルマスクを設定しなければなりません。
rt_task_wait	pthread_wait_np. pthread_make_periodic_npによって指定された次の開始時間までスレッドを停止させる。	なし プロセスをサスペンドさせたり、再起動させるpthread関数は存在しない。 任意のプロセス、スレッドから再起動させることの出来るメソッドはシグナルしかないため、sigwait()を使用する。 POSIXにこだわらないのであれば、server_block(),server_wake()が利用できる。
	pthread_attr_getcpu_np. RT-Linuxではどのスレッドがどのcpuで実行されているのか調べてくれない。また，スレッドは そのスレッドを呼び出したCPUで実行される。しかしRT-Linux はそのスレッドがどこで実行されているかを聞いてくる。 そこでこの関数を用いてpthreadの属性を調べる。	pthread_getaffinity_np() CPUのアフィニティマスクを得る。 しかし、どのＣＰＵで実行されているかは、この関数では解らない。 実行されているＣＰＵを調べるためには、mpadvise (MPA_PRC_GETRUN,MPA_TID,0,setp); を使用する。
	pthread_attr_setreserve_np(V3)	
	pthread_attr_setcpu_np. pthreadの属性を変更する。	pthread_setaffinity() CPUのアフィニティマスクを設定する。
	pthread_attr_getreserve_np(V3)	
	pthread_delete_np. リアルタイムスレッドを消去する。	なし
	pthread_setfp_np. 浮動小数点数の利用の可，不可を決定する。0 で不可，それ以外で可となる。	なし
	pthread_suspend_np. pthread_wakeup_np を呼び出すまでスレッドをサスペンドさせる。	なし 周期タイマーを停止させる。
	pthread_wakeup_np. pthread_suspend_np によってサスペンドさせたスレッドを復帰させる。	なし プロセスをサスペンドさせたり、再起動させるpthread関数は存在しない。 任意のプロセス、スレッドから再起動させることの出来るメソッドはシグナルしかないため、sigwait(),sigsupend()を使用する。 POSIXにこだわらないのであれば、server_block(),server_wake()が利用できる。
	pthread_make_periodic_np. 周期的なスレッドを作成する。	なし 周期タイマーを生成し、シグナルを発生させる
	pthread_linux	なし pthread_create()
Comドライバ		
	rt_com_read. シリアルポートからデータを読み込みこむ。	read()
	rt_com_setup. シリアルポートに関するパラメータを変更する。	tcseta(),ioctl()
	rt_com_table.3 シリアルポート 1 つにつき 1 つ作られるテーブルである。	
	rt_com_write.3 シリアルポートへデータを送る。	write()
rtf RT-FIFO 用のキャラクタ型デバイスのこと メジャーナンバ 150，マイナーナンバ 063 デバイスファイルは"/dev/rtf"になる。		
	# rtf_create. RT-FIFO を作る。 その際，名前とサイズを決定する。	mkfifo()
	rtf_create_handler RT-FIFO 用のハンドラをインストールする。 ハンドラは Linux のプロセスが RT-FIFO にアクセスする度に呼び出される。	なし メソッドはメッセージキューの通知スレッドが最も近い
	rtf_destroy rtf_create によって作られた RT-FIFO の削除。	close(),unlink()

	rtf_get. RT-FIFO からデータを読み込む。	read()
	rtf_put. RT-FIFO へデータを書き込む。	write()
request_RTirq	rtl_request_irq. IRQ への割り込みを行うリアルタイムハンドラのインストール。	なし
free_RTirq 割り込みハンドラの開放	rtl_free_irq. IRQ への割り込みを行うリアルタイムハンドラの開放。	なし
	rtl_get_soft_irq ソフトウェアハンドラのインストール。	なし
	rtl_free_soft_irq. ソフトウェアハンドラの開放。	なし
	rtl_getcpuid. 現在スレッドの実行されている CPU の ID を調べる。	mpadvise()
	rtl_getschedclock. リアルタイムスケジューラの利用しているクロックの ID を返してくる。	なし 常にCLOCK_REALTIME
	rtl_global_pend_irq. Linux への割り込みを行う IRQ のスケジュールを決める。	なし
	rtl_hard_disable_irq. IRQ を使用不可にする	なし
request_RTirq	rtl_hard_enable_irq. IRQ を使用可にする。	なし
	rtl_allow_interrupts. cpu への割り込み処理の許可。	なし
	rtl_no_interrupts. 割り込みに関するパラメータを保存してから割り込みの中止。	なし
	rtl_restore_interrupts. rtl_no_interrupts によって保存されたパラメータを読み込んでから割り込み可能にする。	なし
	rtl_stop_interrupts. cpu への割り込み処理の中止。	なし
	rtl_setclockmode. クロックモードの設定。	なし

RTLinux 関数の模擬

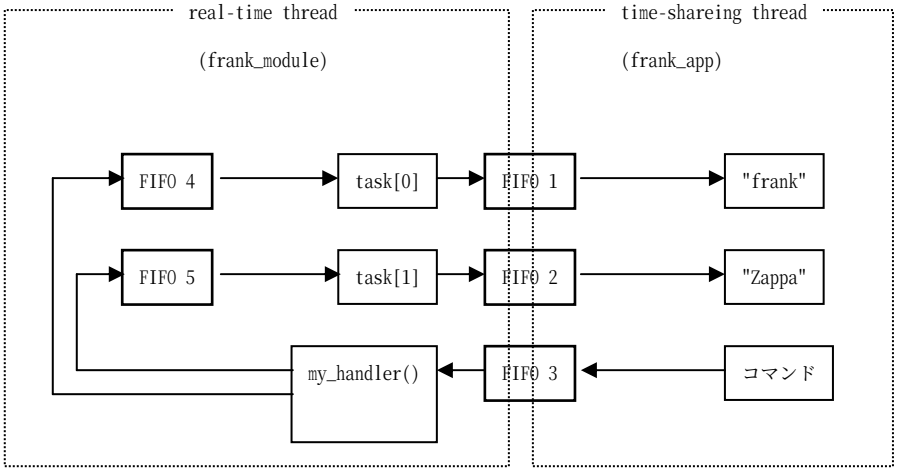
RTLinx 関数を模擬するプログラムのサンプルを下記に示します。
このプログラムは、RTLinux のサンプルプログラムである、frank プログラムを動作させるためのもので、RTLinux の関数の動作を標準 Linux の関数で模擬することで、移植における問題点を明確にすることができます。

RTLinux における frank プログラムは、カーネル空間スレッドである、frank_module.c とユーザ空間スレッドである、frank_app.c が FIFO を使って通信を行うものです。



カーネル空間スレッド　　ユーザ空間スレッド

このサンプルプログラムは、frank_module をリアルタイムスレッド、frank_app をタイムシェアリングスレッドとして、実行するためのライブラリです。



ユーザ空間スレッド　　ユーザ空間スレッド

RTLinux では、frank_module.c は、カーネルドライバモジュールであるため main()を持っていません。
そこで、ユーザ空間で実行可能にするため、frank_module.に以下の行を追加します。

```
int main()
{
        init_valueue();//ライブラリ変数の初期化
        init_module();
        pause();
        cleanup_module();
}
```

ここで、RTLinux の構造を模擬するために、以下の置き換えを行います。

”RT-FIFO”	→	”named pipe”
“カーネルスレッド”	→	“realtime pthread”
“周期スケジューリングスレッド”	→	“POSIX タイマー待ち pthread”
“rtf_get(),rtf_put()”	→	“非ブロック read(),非ブロック write()”
“pthread_wakeup_np()”	→	“リアルタイムシグナルの送信”
“pthread_wait_np()”	→	“リアルタイムシグナルの待ち
“pthread_suspend_np()”	→	“POSIX タイマー停止”
“pthread_make_periodic_np()”	→	”POSIX タイマーと pthread の生成”
“rtf_create_handler()”	→	“select()待ち pthread の生成”
“rtf_printf()”	→	”printf()”
“gethrtime()”	→	“リアルタイムカウンタの読み出し”

この実装で、注意を要するのは、リアルタイムシグナルです。
リアルタイムシグナルは、NPTL の実装で利用しているため、SIGRTMIN+2 以上しか利用できません。
また、現在の NPTL では、POSIX タイマーはスレッドを生成できません。したがって、POSIX タイマーハンドリングスレッドを生成し、そこから、ターゲットスレッドを呼び出す必要があります。そのうえ１つの RTLinux スレッドから複数の rtf_create_handler()を呼び出す可能性があるため､タイマーハンドリングスレッド毎に異なるリアルタイムシグナルを割り当てる必要があります。このため、最大でも(SIGRTMAX-SIGRTMIN-2)個のタイマースレッドしか生成出来ません。
　また、

では、実際の動作に従って、エミュレーションライブラリを使った frank_module.c の動作を詳しく見て行きます。
frank_module.c の関数 init_module()で、thread_code と my_handler の 2 種類のスレッドを作成しています。

frank_module.c より

```
pthread_attr_init (&attr);
sched_param.sched_priority = 4;
pthread_attr_setschedparam (&attr, &sched_param);
ret = pthread_create (&tasks[0],   &attr, thread_code, (void *)1);
pthread_attr_init (&attr);
sched_param.sched_priority = 5;
pthread_attr_setschedparam (&attr, &sched_param);
ret = pthread_create (&tasks[1],   &attr, thread_code, (void *)2);
rtf_create_handler(3, &my_handler);
```

優先度 4 の thread_code pthread 生成

優先度 5 の thread_code pthread 生成

notification pthread 生成
このスレッドは daemon 型

```
/*エミュレーションライブラリ */
static void notification(void *arg)
{
        unsigned int fifo,sno;
        rtl_fifo_t   *tcb;
        int status=0,retval, fd_flags;
        fifo   = (int)arg;
        tcb = &RTL_FIFO[fifo];
        tcb->flag=1;
        while(tcb->flag)
        {
                tcb->timeout.tv_sec = 0;
                tcb->timeout.tv_usec = 500000;
                FD_ZERO(&tcb->readfds);
                FD_SET(tcb->f_id, &tcb->readfds);
                retval = select(FD_SETSIZE,&tcb->readfds, NULL, NULL, &tcb->timeout);
                if (retval>0)/* fifo にデータが到着すると True になる */
                {
                        if (FD_ISSET(tcb->f_id, &tcb->readfds))
                        {
                            tcb->handler(fifo);/*my_handler()はここから呼び出される*/
                        }
                }
        }
        pthread_exit((void*)&status);
}
```

thread_code()では、関数の冒頭①でret = pthread_wait_np()を実行し、pthread_wakeup_np()で起動されるか、次のスケジューリングタイムまでブロックされています。

このブロックは、my_handler()中②の pthread_wakeup_np()より起こされます。
pthread_wait_np()は、エミュレーション関数では、リアルタイムシグナル待ちの sigwait()③によって置き換えられていますので、pthread_wakeup()は、④の pthread_kill()に置き換えられます。

```
void *thread_code(void *t)
{
   int fifo = (int) t;
   int taskno = fifo - 1;
   struct my_msg_struct msg;
   while (1) {
      int ret;
      int err;
      ret = pthread_wait_np();/*①：下記②から起こされる*/
      if ((err = rtf_get (taskno + TASK_CONTROL_FIFO_OFFSET,(char*) &msg, sizeof(msg))) == sizeof(msg)) {
         rtl_printf("Task %d: executing the ¥"%d¥" command to task %d; period %d¥n",
                    fifo - 1,    msg.command, msg.task, msg.period);
         switch (msg.command) {
            case START_TASK:
               rtl_printf("RTLinux task: START_TASK: command¥n");
               pthread_make_periodic_np(pthread_self(), gethrtime(), msg.period * 1000);/*⑤*/
               break;
            case STOP_TASK:
               rtl_printf("RTLinux task: STOP_TASK: command¥n");
               pthread_suspend_np(pthread_self());/*⑨*/
               break;
            default:
               rtl_printf("RTLinux task: bad command¥n");
               return 0;
         }
      }
      rtf_put(fifo, data[fifo - 1], 6);
   }
   return 0;
}
int my_handler(unsigned int fifo)
{
   struct my_msg_struct msg;
   int err;
   while ((err = rtf_get(COMMAND_FIFO,(char*) &msg, sizeof(msg))) == sizeof(msg)) {
      rtf_put (msg.task + TASK_CONTROL_FIFO_OFFSET,(char*) &msg, sizeof(msg));
      rtl_printf("FIFO handler: sending the ¥"%d¥" command to task %d; period %d¥n", msg.command,
            msg.task, msg.period);
      pthread_wakeup_np (tasks [msg.task]);/*②：上記①を起こす*/
   }
   if (err != 0) {
      return -EINVAL;
   }
   return 0;
}
```

```
int pthread_wakeup_np(pthread_t thread)
{/*エミュレーションライブラリ */
   int tno;
   tno = get_tno(pthread_self());
   RTL_TSD[tno].state=RTL_RUN;
   return(pthread_kill(thread, WAITSIGNAL)/*④*/);
}
```

```
int pthread_wait_np(void)
{/*エミュレーションライブラリ */
   sigset_t diset,eiset,oset;
   int ret,signo,tno;
   tno = get_tno(pthread_self());
   sigfillset(&diset);
   for(signo=WAITSIGNAL;signo<=SIGRTMAX;signo++)
   {
      sigdelset(&diset,signo);
   }
   if (pthread_sigmask(SIG_BLOCK,&diset,&oset)==(-1))
   {
      return (-1);
   }
   sigemptyset(&eiset);
   sigaddset(&eiset,WAITSIGNAL);
   sigaddset(&eiset,RTL_TSD[tno].timer_sno);
   sigaddset(&eiset,SIGINT);
   sigaddset(&eiset,SIGIO);
   RTL_TSD[tno].state=RTL_SUSPEND;
   ret = sigwait(&eiset,&signo);/*③*/
   RTL_TSD[tno].state=RTL_RUN;
   if (pthread_sigmask(SIG_SETMASK,&oset,&eiset)==(-1))
   {
         return (-1);
   }
   return(ret);
}
```

pthread_wakeup_np()によって起こされたスレッドは、rtf_get()の START_TASK メッセージによって⑤の pthread_make_periodic_np(pthread_self(), gethrtime(), msg.period * 1000);が実行されます。

```
int pthread_make_periodic_np(pthread_t thread,hrtime_t start_time,hrtime_t period)
{/*エミュレーションライブラリ */
        static struct sigevent event;
        static timer_t timer;
        static struct itimerspec itspec;
        int tno;
        void * val=NULL;
        tno = get_tno(thread);
        /* Create the POSIX timer to generate signum */
        event.sigev_notify = SIGEV_SIGNAL;/*⑥*/
        RTL_TSD[tno].timer_sno = event.sigev_signo = get_sno();
        if (event.sigev_signo < (WAITSIGNAL+1))
        {
             return (-1);
        }
        if (timer_create((clockid_t)CLOCK_REALTIME,&event,&timer)==(-1))/*⑦*/
        {
             RTL_TSD[tno].timer_key=(timer_t)-1;
             return (-1);
        }
        /* Set the initial delay and period of the timer.
        This also arms the timer */
        clock_gettime(CLOCK_REALTIME,&itspec.it_value);
        start_time+=10000LL;
        itspec.it_value.tv_sec = (unsigned long int)(start_time>>32);
        itspec.it_value.tv_nsec = (unsigned long int)(start_time & 0xFFFFFFFFLL);
        itspec.it_interval.tv_sec = (unsigned long int)(period>>32);
        itspec.it_interval.tv_nsec = (unsigned long int)(period & 0xFFFFFFFFLL);
        if (timer_settime( timer, TIMER_ABSTIME, &itspec, NULL)==(-1))/*⑧*/
        {
          RTL_TSD[tno].timer_key=(timer_t)-1;
          return (-1);
        }
        RTL_TSD[tno].timer_key=timer;
        RTL_TSD[tno].state=RTL_RUN;
        sched_yield();
        return(0);
}
```

```
int pthread_suspend_np(pthread_t thread)
{*エミュレーションライブラリ */
   int tno;
   tno = get_tno(pthread_self());
   if (pthread_equal(pthread_self(),thread) )
   {
      if (RTL_TSD[tno].timer_key!=((timer_t)-1))
         timer_delete(RTL_TSD[tno].timer_key);/*⑩*/
   }else{
      return(pthread_kill(thread, WAITSIGNAL));/*⑪*/
   }
}
```

この関数は、スレッドを周期的に呼び出すための API で、スレッドの開始時間 start_time と呼び出す周期 period を hrtime_t 型変数として取ります。hrtime_t 型変数は long long 型として定義されています。

エミュレーションライブラリでは、⑥イベントとしてリアルタイムシグナルを定義し、⑦POSIX タイマーを生成し、起動時間を⑧で設定しています。

この動作によって、呼び出したスレッドに対して、リアルタイムシグナルが発信されますが、この実装では、自分自身にしかイベントは送信されないため、完全なエミュレーションを実現するためには、管理スレッドを生成し、そのスレッドから改めて、pthread_kill()を発行する必要があります。

同様に、rtf_get()の STOP_TASK メッセージによって⑨の pthread_suspend_np(pthread_self());が実行された場合には、エミュレーションライブラリで、⑩タイマーを停止し、⑪pthread_kill()を実行します。

下記にこの実装の全てのソースコードを掲げますが、この説明に必要ない機能は、コメントになっています。

```
/*
rtl.h
*/
#ifndef _RTL_H_
#define _RTL_H_
#include <stdio.h>
#include <pthread.h>
#include <limits.h>
#include <time.h>
#include <mqueue.h>
#include <signal.h>
#include <errno.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <sched.h>
#include <string.h>
#include <sys/mman.h>
#include <cpuset.h>
#include <sched.h>
#include <mpadvise.h>
#include <termios.h>



#define _EMULATE_RTLINUX_
#define hrtime_t long long

/* Section 2  */
#define rtl_close           close
#define rtl_ftruncate       rtl_ftruncate
#define rtl_gpos_unlink     unlink
#define rtl_inb                         inb
#define rtl_inb_p           inb_p
#define rtl_inl                         inl
#define rtl_inl_p           inl_p
#define rtl_inw                         inw
#define rtl_inw_p           inw_p
#define rtl_ioctl           ioctl
#define rtl_lseek           lseek
#define rtl_mmap            mmap
#define rtl_munmap          unmap
#define rtl_open            open
#define rtl_outb            outb
#define rtl_outb_p          outb_p
#define rtl_outl            outl
#define rtl_outl_p          outl_p
#define rtl_outw            outw
#define rtl_outw_p          outw_p
#define rtl_read            read
#define rtl_shm_open        shm_open
#define rtl_shm_unlink      shm_unlink
#define rtl_unlink          unlink
#define rtl_write           write


/* Section 3 */

    /* malloc */
#define     rtl_size_t      size_t
#define rtl_rt_malloc       malloc
#define rtl_rt_free         free
#define rtl_rt_realloc      realloc
    /* calloc */
```

```
#define       rtl_rt_calloc  calloc
//void *rtl_rt_malloc_pool(int pool, rtl_size_t size);
//void *rtl_rt_calloc_pool(int pool, rtl_size_t nmemb, rtl_size_t size);
//int rtl_init_mem_pool(int pool, int type, int chunksize, int numchunks);
//void rtl_cleanup_mem_pool(int pool);
      /* debugpr */
#define       debugpr      printf
      /* onewayq */ // not support

int rtf_create(unsigned int fifo,int size);
int rtf_create_handler(unsigned int fifo,int (*handler)(unsigned int fifo) );
#define rtf_create_rt_handler             rtf_create_handler
int rtf_destroy(unsigned int fifo);
int rtf_flush(unsigned int fifo);
int rtf_get(unsigned int fifo,char *buff,int count);
      // rtf_isempty
      // rtf_isused
      // rtf_link_user_ioctl
      // rtf_make_user_pair
int rtf_put(unsigned int fifo,char *buff,int count);
      // rtf_resize
      // rtl_a_clear
      // rtl_a_decr
      // rtl_a_incr
      // rtl_allow_interrupts
      // rtl_a_set
      // rtl_breakpoint
      // rtl_cleanup_mem_pool
hrtime_t clock_gethrtime(clockid_t clock);
#define RTL_CLOCK_REALTIME CLOCK_REALTIME
#define rtl_timespec timespec
#define rtl_clockid_t        clockid_t
#define rtl_clock_getres   clock_getres
#define rtl_clock_gettime clock_gettime
#define rtl_clock_nanosleep               clock_nanosleep
#define rtl_clock_settime clock_settime
int rtl_cpu_exists (int cpu);
      // rtl_decr_dev_usage
      // rtl_free_irq
      // rtl_free_soft_irq
      // rtl_freq_cur
      // rtl_freq_list
      // rtl_freq_set
      // rtl_getcpuid
      // rtl_get_soft_irq
      // rtl_global_pend_irq
      // rtl_gpos_free
      // rtl_gpos_malloc
      // rtl_gpos_register_dev
      // rtl_gpos_unregister_dev
      // rtl_hard_disable_irq
      // rtl_hard_enable_irq
      // rtl_incr_dev_usage
      // rtl_init_mem_pool
      // rtl_irq_set_affinity
#define       rtl_longjmp   longjmp
#define rtl_jmp_buf         jmp_buf
#define       rtl_main_wait pause
#define       rtl_mkfifo     mkfifo
#define       rtl_mode_t    mode_t
      // rtl_namei
#define rtl_nanosleep       nanosleep
      // rtl_no_interrupts
int rtl_num_cpus(void);
#define rtl_perror            perror
#define rtl_printf            printf
#define rtl_pthread_t rtl_pthread
#define rtl_pthread_attr_destroy                  pthread_attr_destroy
#define rtl_pthread_attr_t                              pthread_attr_t
      // rtl_pthread_attr_getcpu_np
#define rtl_pthread_attr_getdetachstate                 pthread_attr_getdetachstate
      // rtl_pthread_attr_getfp_np
      // rtl_pthread_attr_getreserve_np
```

```
#define rtl_pthread_attr_getschedparam                  pthread_attr_getschedparam
#define rtl_pthread_attr_getstackaddr           pthread_attr_getstackaddr
#define rtl_pthread_attr_getstacksize           pthread_attr_getstacksize
#define rtl_pthread_attr_init                           pthread_attr_init
    // rtl_pthread_attr_setcpu_np
#define rtl_pthread_attr_setdetachstate                 pthread_attr_setdetachstate
    // rtl_pthread_attr_setfp_np
    // rtl_pthread_attr_setreserve_np
#define rtl_pthread_attr_setschedparam                  pthread_attr_setschedparam
#define rtl_pthread_attr_setstackaddr           pthread_attr_setstackaddr
#define rtl_pthread_attr_setstacksize           pthread_attr_setstacksize
#define rtl_pthread_cancel                              pthread_cancel
#define rtl_pthread_cleanup_pop                         pthread_cleanup_pop
#define rtl_pthread_cleanup_push                pthread_cleanup_push
#define rtl_pthread_condattr_t                          pthread_condattr_t
#define rtl_pthread_condattr_destroy            pthread_condattr_destroy
#define rtl_pthread_condattr_getpshared                 pthread_condattr_getpshared
#define rtl_pthread_condattr_init               pthread_condattr_init
#define rtl_pthread_condattr_setpshared                 pthread_condattr_setpshared
#define rtl_pthread_cond_broadcast              pthread_cond_broadcast
#define rtl_pthread_cond_destroy                pthread_cond_destroy
#define rtl_pthread_cond_init                           pthread_cond_init
#define rtl_pthread_cond_signal                         pthread_cond_signal
#define rtl_pthread_cond_timedwait              pthread_cond_timedwait
#define rtl_pthread_cond_wait                           pthread_cond_wait
int rtl_pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);
    // rtl_pthread_delete_np
#define rtl_pthread_detach                              pthread_detach
#define rtl_pthread_equal                       pthread_equal
#define rtl_pthread_exit                        pthread_exit
#define rtl_pthread_getcpuclockid               pthread_getcpuclockid
#define rtl_pthread_getschedparam               pthread_getschedparam
#define rtl_pthread_getspecific                         pthread_getspecific
#define rtl_pthread_idle                        pause
#define rtl_pthread_join                        pthread_join
#define rtl_pthread_key_create                          pthread_key_delete
#define rtl_pthread_key_delete                          pthread_key_delete
#define rtl_pthread_kill                        pthread_kill
#define rtl_pthread_linux                       pause
#define rtl_pthread_mutex_t                             pthread_mutex_t
#define rtl_pthread_make_periodic_np            pthread_make_periodic_np
#define rtl_pthread_mutexattr_destroy           pthread_mutexattr_destroy
#define rtl_pthread_mutexattr_getprioceiling    pthread_mutexattr_getprioceiling
#define rtl_pthread_mutexattr_getprotocol       pthread_mutexattr_getprotocol
#define rtl_pthread_mutexattr_getpshared        pthread_mutexattr_getpshared
#define rtl_pthread_mutexattr_gettype           pthread_mutexattr_gettype
#define rtl_pthread_mutexattr_init              pthread_mutexattr_init
#define rtl_pthread_mutexattr_setprioceiling    pthread_mutexattr_setprioceiling
#define rtl_pthread_mutexattr_setprotocol       pthread_mutexattr_setprotocol
#define rtl_pthread_mutexattr_setpshared        pthread_mutexattr_setpshared
#define rtl_pthread_mutexattr_settype           pthread_mutexattr_settype
#define rtl_pthread_mutex_destroy               pthread_mutex_destroy
#define rtl_pthread_mutex_getprioceiling        pthread_mutex_getprioceiling
#define rtl_pthread_mutex_init                          pthread_mutex_init
#define rtl_pthread_mutex_lock                          pthread_mutex_lock
#define rtl_pthread_mutex_setprioceiling        pthread_mutex_setprioceiling
#define rtl_pthread_mutex_timedlock             pthread_mutex_timedlock
#define rtl_pthread_mutex_trylock               pthread_mutex_trylock
#define rtl_pthread_mutex_unlock                pthread_mutex_unlock
#define rtl_pthread_self                        pthread_self
#define rtl_pthread_setcancelstate              pthread_setcancelstate
#define rtl_pthread_setcanceltype               pthread_setcanceltype
        // rtl_pthread_setfp_np
#define rtl_pthread_setschedparam               pthread_setschedparam
#define rtl_pthread_setspecific                         pthread_setspecific
#define rtl_pthread_spin_destroy                pthread_spin_destroy
#define rtl_pthread_spin_init                           pthread_spin_init
#define rtl_pthread_spin_lock                           pthread_spin_lock
#define rtl_pthread_spin_trylock                pthread_spin_trylock
#define rtl_pthread_spin_unlock                         pthread_spin_unlock
int pthread_suspend_np(pthread_t thread);
#define rtl_pthread_testcancel                          pthread_testcancel
int pthread_wait_np(void);
```

```
int pthread_wakeup_np(pthread_t thread);
	//rtl_put_circle
	//rtl_put_fillcircle
	//rtl_put_fillrect
	//rtl_put_line
	//rtl_put_pixel
	//rtl_put_rect
	//rtl_put_text
	//rtl_register_dev
	//rtl_register_rtldev
	//rtl_request_irq
	//rtl_restore_interrupts
	//rtl_rt_calloc
	//rtl_rt_calloc_pool
	//rtl_rt_free
	//rtl_rt_malloc
	//rtl_rt_malloc_pool
	//rtl_rt_realloc
#define rtl_sched_get_priority_max		get_priority_max
#define rtl_sched_get_priority_min		get_priority_min
#define rtl_sched_yield				sched_yield
#define rtl_sem_t				sem_t
#define rtl_sem_close				sem_close
#define rtl_sem_destroy				sem_destroy
#define rtl_sem_getvalue			sem_getvalue
#define rtl_sem_init				sem_init
#define rtl_sem_open				sem_open
#define rtl_sem_post				sem_post
#define rtl_sem_timedwait			sem_timedwait
#define rtl_sem_trywait				sem_trywait
#define rtl_sem_unlink				sem_unlink
#define rtl_sem_wait				sem_wait
#define	rtl_jmp_buf				jmp_buf
#define rtl_setjmp				setjmp
#define rtl_sigaction				sigaction
		// rtl_stop_interrupts
	// rtl_test_bit_and_clear
	// rtl_test_bit_and_set
#define rtl_timespec				timespec
	// rtl_timespec_add
	// rtl_timespec_add_ns
	// rtl_timespec_eq
	// rtl_timespec_from_ns
	// rtl_timespec_ge
	// rtl_timespec_gt
	// rtl_timespec_le
	// rtl_timespec_lt
	// rtl_timespec_normalize
	// rtl_timespec_nz
	// rtl_timespec_sub
	// rtl_timespec_to_ns
#define rtl_uname				uname
	// rtl_unregister_dev
	// rtl_unregister_rtldev
#define rtl_usleep				usleep
	// sched_get_cpufreq_max
	// sched_get_cpufreq_min


#define WAITSIGNAL	(SIGRTMIN+2)

typedef struct
{
		timer_t			timer_key;
		int			timer_sno;
		int			state;
		int			request;
		int			tno;
		pthread_t	pid;
} rtltsd_t;

typedef struct
{
```

```
        int flag;
        char       f_name[256];
        int (*handler)(unsigned int fifo);
        pthread_attr_t     attributes;
#ifdef MESSAGE_QUEUE
    mqd_t f_id;
    struct mq_attr f_attr;
        struct sigevent notification;
#else
    int f_id;
        pthread_t     pid;
        //struct timespec timeout;
        struct timeval    timeout;
        sigset_t sigmask;
        fd_set readfds;
#endif
} rtl_fifo_t;

#define RTL_SUSPEND   (0)
#define RTL_RUN                 (1)

hrtime_t gethrtime(void);
void init_valueue(void);
int pthread_make_periodic_np(pthread_t thread,hrtime_t start_time,hrtime_t period);
int pthread_suspend_np(pthread_t thread);
int pthread_wait_np(void);
int pthread_wakeup_np(pthread_t thread);
int pthread_attr_setcpu_np(const pthread_attr_t *attr,int cpu);
int pthread_attr_getcpu_np(const pthread_attr_t *attr,int *cpu);

#endif /* _RTL_H_ */
```

```
/*
rtl.c
*/
#include <rtl.h>

static rtltsd_t RTL_TSD[_POSIX_THREAD_THREADS_MAX];
static rtl_fifo_t    RTL_FIFO[_POSIX_THREAD_THREADS_MAX];
static int RTL_NUMBER=0;
static int SNO_NUMBER=0;
static int Stop_flag=0;
static int global_setup_signal(int signo, void (*interrupt_handler)(int,siginfo_t *, void *))
{
        static struct sigaction newact;
        static sigset_t set,oset;

        if (sigprocmask(0,NULL,&set)==(-1))
        {
                return(-1);
        }
        sigdelset(&set,signo);
        if (sigprocmask(SIG_SETMASK,&set,&oset)==(-1))
        {
                return(-2);
        }

        sigemptyset(&newact.sa_mask);
        sigaddset(&newact.sa_mask,signo);
        newact.sa_sigaction = interrupt_handler;
        newact.sa_flags = SA_SIGINFO|SA_RESTART;
        if (sigaction(signo, &newact, 0)==(-1))
        {
                return(-3);
        }
        return(0);
}

static void sigint_handler(int signo, siginfo_t *siginfo, void *misc)
{
    if ((siginfo->si_pid==0)||(getpid()==siginfo->si_pid)|| (getppid()==siginfo->si_pid))
    {
        fprintf(stderr,"¥nSIG%d DETECT¥n",signo);
        Stop_flag = 1;
    }
}

void wait_module(void)
{
            while(Stop_flag==0) pause();
}

void init_valueue(void)
{
            int fifo,sno;
            sigset_t diset,oset;
            struct sched_param myparam;

            SNO_NUMBER=WAITSIGNAL+1;
            sigemptyset(&diset);
            sigaddset(&diset,WAITSIGNAL);
            for(sno=WAITSIGNAL+1;sno<=SIGRTMAX;sno++)
            {
                        sigaddset(&diset,sno);
            }
        sigprocmask(SIG_BLOCK,&diset,&oset);
            memset(RTL_TSD,0,sizeof(RTL_TSD));
            memset(RTL_FIFO,0,sizeof(RTL_FIFO));
            RTL_NUMBER=0;
            for(fifo=0;fifo<_POSIX_THREAD_THREADS_MAX;fifo++)
            {
                        RTL_FIFO[fifo].f_id = -1;
            }
        myparam.sched_priority = sched_get_priority_max(SCHED_FIFO);
        sched_setscheduler(getpid(),SCHED_FIFO,&myparam);
```

```
	mlockall(MCL_CURRENT|MCL_FUTURE);
	global_setup_signal(SIGINT,sigint_handler);
}


static int get_sno(void)
{
	int sno;
	sno = SNO_NUMBER;
	if (sno>=SIGRTMAX)	return(-1);
	++SNO_NUMBER;
	return(sno);
}

static int get_tno(pthread_t pid)
{
	int tno;

	if (RTL_NUMBER)
	{
		for (tno=0;tno<RTL_NUMBER;tno++)
		{
			if (pthread_equal(RTL_TSD[tno].pid,pid) )
			{
				return(tno);
			}
		}
	}
	tno = RTL_NUMBER;++RTL_NUMBER;
	RTL_TSD[tno].pid = pid;
	RTL_TSD[tno].tno = tno;
	return(tno);
}

int pthread_wait_np(void)
{
	sigset_t diset,eiset,oset;
	int ret,signo,tno;

	tno = get_tno(pthread_self());
	resume:
	{
		sigfillset(&diset);
		for(signo=WAITSIGNAL;signo<=SIGRTMAX;signo++)
		{
			sigdelset(&diset,signo);
		}
	if (pthread_sigmask(SIG_BLOCK,&diset,&oset)==(-1))
	{
			return (-1);
	}

		sigemptyset(&eiset);
		sigaddset(&eiset,WAITSIGNAL);
		sigaddset(&eiset,RTL_TSD[tno].timer_sno);
		sigaddset(&eiset,SIGINT);
		sigaddset(&eiset,SIGIO);
		RTL_TSD[tno].state=RTL_SUSPEND;
		ret = sigwait(&eiset,&signo);
		RTL_TSD[tno].state=RTL_RUN;
	if (pthread_sigmask(SIG_SETMASK,&oset,&eiset)==(-1))
	{
			return (-1);
	}
	}
//printf("pthread_wait_np(ret)¥n");
	return(ret);
}

hrtime_t clock_gethrtime(clockid_t clock)
{
	struct timespec nowtime;
	long long x;
```

```
	clock_gettime(clock,&nowtime);
	x = (long long)nowtime.tv_sec << 32;
	x |= (long long)nowtime.tv_nsec;
	    return(x);
}

int rtl_cpu_exists (int cpu)
{
	cpuset_t* setp;
	char *cpustr;
	int ret,mask;

	setp = cpuset_alloc();
	cpuset_init(setp);
	ret = mpadvise (MPA_CPU_PRESENT,0,0,setp);
	if (ret!=MPA_FAILURE)
	{
		cpustr = cpuset_get_string(setp);
		mask=   strtol(cpustr, (char **)NULL, 16);
		free(cpustr);
			return(mask & cpu);
	}
	free(cpustr);
	    return(-1);
}

int rtl_num_cpus(void)
{
	cpuset_t* setp;
	char *cpustr;
	int ret,cpu,mask;

	setp = cpuset_alloc();
	cpuset_init(setp);
	ret = mpadvise (MPA_CPU_ACTIVE,0,0,setp);
	if (ret!=MPA_FAILURE)
	{
		free(cpustr);
			return ( cpuset_count(setp) );
	}
	free(cpustr);
	    return(-1);
}

hrtime_t gethrtime(void)
{
	struct timespec nowtime;
	long long x;

	clock_gettime(CLOCK_REALTIME,&nowtime);
	x = (long long)nowtime.tv_sec << 32;
	x |= (long long)nowtime.tv_nsec;
	    return(x);
}

int pthread_make_periodic_np(pthread_t thread,hrtime_t start_time,hrtime_t period)
{
	static struct sigevent event;
	static timer_t timer;
	static struct itimerspec itspec;
	    int tno,policy;
	    void * val=NULL;
	    struct sched_param param;

	pthread_getschedparam(pthread_self(),&policy,(struct sched_param *)&param);
	param.sched_priority = sched_get_priority_max(SCHED_FIFO);
	pthread_setschedparam(pthread_self(),SCHED_FIFO,(const struct sched_param *)&param);


	    tno = get_tno(thread);
	/* Create the POSIX timer to generate signum */
	event.sigev_notify = SIGEV_SIGNAL;
```

```
        RTL_TSD[tno].timer_sno = event.sigev_signo = get_sno();
        if (event.sigev_signo < (WAITSIGNAL+1))
        {
                return (-1);
        }
    if (timer_create((clockid_t)CLOCK_REALTIME,&event,&timer)==(-1))
    {
                RTL_TSD[tno].timer_key=(timer_t)-1;
        return (-1);
    }
    /*      Set the initial delay and period of the timer.
    This also arms the timer */
    clock_gettime(CLOCK_REALTIME,&itspec.it_value);
        start_time+=10000LL;
    itspec.it_value.tv_sec = (unsigned long int)(start_time>>32);
    itspec.it_value.tv_nsec = (unsigned long int)(start_time & 0xFFFFFFFFLL);
    itspec.it_interval.tv_sec = (unsigned long int)(period>>32);
    itspec.it_interval.tv_nsec = (unsigned long int)(period & 0xFFFFFFFFLL);
    if (timer_settime( timer, TIMER_ABSTIME, &itspec, NULL)==(-1))
    {
                RTL_TSD[tno].timer_key=(timer_t)-1;
        return (-1);
    }
        RTL_TSD[tno].timer_key=timer;
        RTL_TSD[tno].state=RTL_RUN;
        sched_yield();
    return(0);
}

int pthread_wakeup_np(pthread_t thread)
{
        int tno;
        tno = get_tno(pthread_self());
        RTL_TSD[tno].state=RTL_RUN;
        return(pthread_kill(thread, WAITSIGNAL));
}

int pthread_suspend_np(pthread_t thread)
{
        int tno;
        tno = get_tno(pthread_self());
        if (pthread_equal(pthread_self(),thread) )
        {
                if (RTL_TSD[tno].timer_key!=((timer_t)-1))
                        timer_delete(RTL_TSD[tno].timer_key);
        }
        else
        {
                return(pthread_kill(thread, WAITSIGNAL));
        }
}

static void notification(void *arg) /* スレッド開始関数 */
{
        unsigned int fifo,sno;
        rtl_fifo_t    *tcb;
        int status=0,retval, fd_flags;

        fifo    = (int)arg;
        tcb = &RTL_FIFO[fifo];

        tcb->flag=1;
        while(tcb->flag)
        {
                tcb->timeout.tv_sec = 0;
                tcb->timeout.tv_usec = 500000;
                FD_ZERO(&tcb->readfds);
                FD_SET(tcb->f_id, &tcb->readfds);
                retval = select(FD_SETSIZE,&tcb->readfds, NULL, NULL, &tcb->timeout);
                if (retval>0)
                {
                        if (FD_ISSET(tcb->f_id, &tcb->readfds))
                        {
```

```
                                                tcb->handler(fifo);
                                        }
                                }
                        }
                pthread_exit((void*)&status);
}

int rtf_create_handler(unsigned int fifo,int (*handler)(unsigned int fifo) )
{
                rtl_fifo_t    *tcb;

                tcb = &RTL_FIFO[fifo];

            if (pthread_attr_init(&tcb->attributes))
            {
                        return(-1);
            }
            if (pthread_attr_setscope(&tcb->attributes,PTHREAD_SCOPE_SYSTEM))
            {
                        return(-1);
            }
            if (pthread_attr_setdetachstate(&tcb->attributes,PTHREAD_CREATE_DETACHED))
            {
                        return(-1);
            }
            if (pthread_attr_setinheritsched(&tcb->attributes,PTHREAD_INHERIT_SCHED))
            {
                        return(-1);
            }
                tcb->handler = handler;
                if (pthread_create(&tcb->pid,&tcb->attributes,(void*)notification,(void *)fifo) < 0)
                {
                        return(-1);
                }
                return(0);
}

int rtf_create(unsigned int fifo,int size)
{
                rtl_fifo_t    *tcb;

                tcb = &RTL_FIFO[fifo];
                tcb->flag=0;
            sprintf(tcb->f_name,"/dev/rtf%d",fifo);
                if (size>4096)
                {
                        return(-1);
                }
                if (mkfifo(tcb->f_name, 0666)<0)
            {
                  return(-1);
            }
                tcb->f_id = open(tcb->f_name,O_RDWR|O_NONBLOCK,0666);
            return(tcb->f_id);
                return(0);
}

int rtf_destroy(unsigned int fifo)
{
                rtl_fifo_t    *tcb;

                tcb = &RTL_FIFO[fifo];
            if (tcb->f_id!=(-1))
            {
                    close(tcb->f_id);
                        tcb->f_id = -1;
            }
            sprintf(tcb->f_name,"/dev/rtf%d",fifo);
                if (!access(tcb->f_name,F_OK))
                {
                        unlink(tcb->f_name);
                }
                if (tcb->flag)
```

```
        {
                tcb->flag=0;
                if (tcb->pid) pthread_kill(tcb->pid,SIGINT);
                tcb->pid=0;
        }
        return(0);
}

int rtf_put(unsigned int fifo,char *buff,int count)
{
        rtl_fifo_t    *tcb;
        int ret;

        tcb = &RTL_FIFO[fifo];
        ret = write(tcb->f_id,buff,count);
        return(ret);
}

int rtf_get(unsigned int fifo,char *buff,int count)
{
        rtl_fifo_t    *tcb;

        tcb = &RTL_FIFO[fifo];
        return(read(tcb->f_id,buff,count));
}
int rtf_flush(unsigned int fifo)
{
        rtl_fifo_t    *tcb;

        tcb = &RTL_FIFO[fifo];
        return(tcflush(tcb->f_id,TCIOFLUSH));
}


int pthread_attr_setcpu_np(const pthread_attr_t *attr,int cpu)
{
    cpuset_t            *cpuset;
        int ret;

    cpuset = cpuset_alloc();
    cpuset_init(cpuset);
    cpuset_set_cpu(cpuset,cpu,1);/* CPU set */

        ret = pthread_attr_setaffinity_np(attr, sizeof(cpuset), cpuset);
    cpuset_free(cpuset);
        return(ret);
}

int pthread_attr_getcpu_np(const pthread_attr_t *attr,int *cpu)
{
    cpuset_t            *cpuset;
        char *str;
        int i,ret;


    cpuset = cpuset_alloc();
    cpuset_init(cpuset);

        ret = pthread_attr_getaffinity_np(attr, sizeof(cpuset), cpuset);
        *cpu=-1;
        for(i=cpuset_min_cpu();i<cpuset_max_cpu();i++)
        {
                        if (cpuset_get_cpu(cpuset,i)) *cpu = i;
        }
    cpuset_free(cpuset);
        return(ret);
}

int rtl_pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)
{
        struct sched_param param;
        pthread_attr_setschedpolicy(attr,SCHED_FIFO);
    param.sched_priority = sched_get_priority_max(SCHED_FIFO);
```

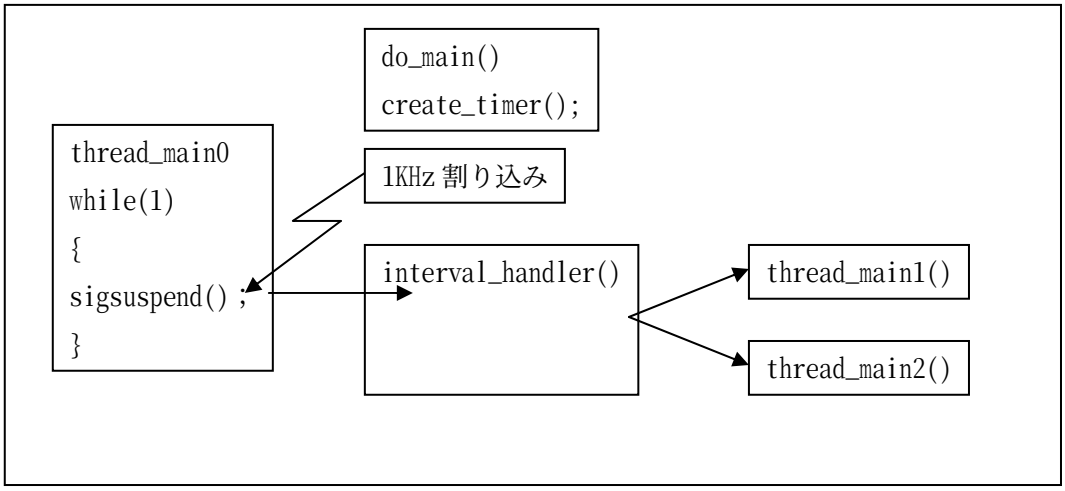
```
                pthread_attr_setschedparam(attr,&param);
                return(pthread_create(thread, attr, start_routine, arg));
}

int rtl_gpos_mknod(const char *pathname, int mode, int major, int minor)
{
        return(mknod(pathname,(mode_t)(mode|S_IFCHR),(dev_t)((major<<16)|(minor))));
}
```

標準 pthread によるミニスケジューラーの実装

前節の frank_module.c,frank_app.c の実装は、周期の異なるスレッドを制御する RTLinux のプログラム例でした。しかし、RedHawk では、標準 API だけで、このプログラムモデルを構築できます。

下記のプログラムは、POSIX リアルタイムタイマーを使って、複数のスレッドをスケジュールするプログラム例です。

tswitch.c では、２つのスレッドを生成し、１つの POSIX タイマーをつかって制御します。
POSIX タイマーは、1ms のインターバル割り込みを生成し、sigsuspend();に割り込む形で interval_handler()が実行されます。



interval_handler()では、割り込みの数を GLOBAL_COUNTER でカウントし、任意のメジャー周期を作り出し、pthread_mutex(),pthread_cond()によって thread_main1(),thread_main2()に対して起動を繰り返しています。スレッドを起動停止しているのは、pthread_mutex と pthread_cond 関数で、この２種類は、以下の様にペアで使用します。

スレッド側　　シグナルハンドラー側

```
pthread_mutex_lock (&Mutex);
if(Event==0) pthread_cond_wait(&Condition,&Mutex);
Event--;
pthread_mutex_unlock(&Mutex);
```

```
pthread_mutex_lockMUTEX_LOCK(&Mutex);
Event++;
pthread_cond_signal(&Condition);
pthread_mutex_unlock(&Mutex);
```

pthread_mutex_lock()/pthread_mutex_unlock()は、mutex 変数 Mutex で、クリティカルセクションを作っています。

この例でのクリティカルセクションデータは Event で、この値を操作する間、Mutex で保護されています。今、スレッド側で、Evant の値が０である（つまりイベントが無い）とき、スレッドは、pthread_mutex_cond_wait()で、Mutex を解放し、Condition 変数で、眠りにつきます。シグナルハンドラーは、POSIX タイマーにより起動され、スレッドが

Mutex を解放している間に、Event を加算し、pthread_cond_signal()を発行します。

休眠状態にあった、スレッドは、シグナルハンドラーが Mutex を解放した時に、pthread_cond_wait()から目覚め、そのとき再び Mutex をロックします。

この動作で、Mutex 変数 Event はイベントカウンターとして動作するため、イベントを失うことなく、スレッドの実行ー停止を繰り返すことができます。

したがって、スレッド側の完全な記述は以下のようになります。

```
do
{
        pthread_mutex_lock(&Mutex[1]);
        if(Event[1]==0)
        {
                pthread_cond_wait(&Condition[1],&Mutex[1]);
        }
        else
        {
                /* オーバーラン状態 */
        }
        Event[1]--;
        pthread_mutex_unlock(&Mutex[1]);

        /*ここで必要な動作を行う; */

} while (loop_flag);
```

tswitch.c のリスト

```
#include <stdio.h>
#include <errno.h>
#include <sys/types.h>
#include <time.h>
#include <signal.h>
#include <string.h>
#include <unistd.h>
#include <mpadvise.h>
#include <pthread.h>
#include <locale.h>
#include <cpuset.h>
#include <stdlib.h>
#include <math.h>

#define CYCLE                1000
#define THREAD_NO        3

char         *progname,*lang;

volatile sig_atomic_t loop_flag=1,GlobalCounter=0;
sigset_t diset,oset,eiset;

#define TESTNUMBER        10000
#define MAXJITTER        (30)

pthread_cond_t Condition[THREAD_NO];
#define COND_SIGNAL        pthread_cond_signal
#define COND_WAIT        pthread_cond_wait
pthread_mutex_t Mutex[THREAD_NO];
#define MUTEX_LOCK        pthread_mutex_lock
#define MUTEX_UNLOCK        pthread_mutex_unlock

int Event[THREAD_NO]={0,0,0};
float SwitchTime1[TESTNUMBER/2];
float SwitchTime2[TESTNUMBER/2];
float IntervalTime[TESTNUMBER];
struct timespec StartTime={0,0},EndTime={0,0};
struct timespec CurrrentTime={0,0};


void
readtime(struct timespec *nowtime)
{
        clock_gettime(CLOCK_REALTIME,nowtime);
}

void
adjust(struct timespec *start,struct timespec *finish, float *realtime)
{
        int     sec,nsec;

        sec = finish->tv_sec - start->tv_sec;
        nsec = finish->tv_nsec - start->tv_nsec;
        if (nsec<0)
        {
                sec --;
                nsec += 1000000000;
        }
        *realtime = (float)(nsec/1000)+(float)(sec*1000000);
}


timer_t
create_posix_timer(int signum,int sec,int nsec)
{
        static struct sigevent event;
        static timer_t timer;
        static struct itimerspec itspec;

        /* Create the POSIX timer to generate signum */
        event.sigev_notify = SIGEV_SIGNAL;
```

```
        //event.sigev_notify = SIGEV_THREAD;
        event.sigev_signo = signum;
        if (timer_create((clockid_t)CLOCK_REALTIME,&event,&timer)==(-1))
        {
                fprintf(stderr, "create_posix_timer() Cannot create timer - %s¥n",
                strerror(errno));
                return ((timer_t)-1);
        }
        /*      Set the initial delay and period of the timer.
        This also arms the timer */
        clock_gettime(CLOCK_REALTIME,&itspec.it_value);
        itspec.it_value.tv_sec += 1;
        itspec.it_interval.tv_sec = sec;
        itspec.it_interval.tv_nsec =  nsec;
        if (timer_settime( timer, TIMER_ABSTIME, &itspec, NULL)==(-1))
        {
                return ((timer_t)-1);
        }
        return(timer);
}


void interval_hadler(int sig)
{
#ifdef DEBUG
        printf("interval_hadler(%d)¥n",GlobalCounter);
#endif
        if(Event[0]!=THREAD_NO) return;

        readtime(&StartTime);
        if(CurrrentTime.tv_sec)
        {
                adjust(&CurrrentTime,&StartTime,&IntervalTime[GlobalCounter]);
        }
        CurrrentTime.tv_sec = StartTime.tv_sec;
        CurrrentTime.tv_nsec = StartTime.tv_nsec;
        switch(GlobalCounter%10)
        {
                case 0:
                        MUTEX_LOCK(&Mutex[1]); /* lock */
                                Event[1]++;
                                COND_SIGNAL(&Condition[1]);
                        MUTEX_UNLOCK(&Mutex[1]); /* unlock */
                break;
                case 1:
                        MUTEX_LOCK(&Mutex[2]); /* lock */
                                Event[2]++;
                                COND_SIGNAL(&Condition[2]);
                        MUTEX_UNLOCK(&Mutex[2]); /* unlock */
                break;
                case 2:
                        MUTEX_LOCK(&Mutex[1]); /* lock */
                                Event[1]++;
                                COND_SIGNAL(&Condition[1]);
                        MUTEX_UNLOCK(&Mutex[1]); /* unlock */
                break;
                case 3:
                        MUTEX_LOCK(&Mutex[2]); /* lock */
                                Event[2]++;
                                COND_SIGNAL(&Condition[2]);
                        MUTEX_UNLOCK(&Mutex[2]); /* unlock */
                break;
                case 4:
                        MUTEX_LOCK(&Mutex[1]); /* lock */
                                Event[1]++;
                                COND_SIGNAL(&Condition[1]);
                        MUTEX_UNLOCK(&Mutex[1]); /* unlock */
                break;
                case 5:
                        MUTEX_LOCK(&Mutex[2]); /* lock */
                                Event[2]++;
                                COND_SIGNAL(&Condition[2]);
                        MUTEX_UNLOCK(&Mutex[2]); /* unlock */
```

```
                break;
                case 6:
                        MUTEX_LOCK(&Mutex[1]); /* lock */
                                Event[1]++;
                                COND_SIGNAL(&Condition[1]);
                        MUTEX_UNLOCK(&Mutex[1]); /* unlock */
                break;
                case 7:
                        MUTEX_LOCK(&Mutex[2]); /* lock */
                                Event[2]++;
                                COND_SIGNAL(&Condition[2]);
                        MUTEX_UNLOCK(&Mutex[2]); /* unlock */
                break;
                case 8:
                        MUTEX_LOCK(&Mutex[1]); /* lock */
                                Event[1]++;
                                COND_SIGNAL(&Condition[1]);
                        MUTEX_UNLOCK(&Mutex[1]); /* unlock */
                break;
                case 9:
                        MUTEX_LOCK(&Mutex[2]); /* lock */
                                Event[2]++;
                                COND_SIGNAL(&Condition[2]);
                        MUTEX_UNLOCK(&Mutex[2]); /* unlock */
                break;
        }
        GlobalCounter++;
}



void *thread_main0(void *arg)
{
        int class = 3;                                  /* FIFO Schedule */
        int priority = 1;                           /* RealTime Priority min */
        struct sched_param param;       /* RealTime Priority min */
        cpuset_t        *cpumask;
        cpu_t                cpu = 1;
        volatile int thread_no;
        int i;
        float min,max,avr;
        struct timespec w0 = {0,100 * 1000* 1000};
        struct timespec wait = {0,200* 1000};

        if (pthread_getschedparam(pthread_self(),&class,&param))
        {
                fprintf(stderr, "pthread %d Cannot get priority %s¥n",
                pthread_self(),strerror(errno));
        }
        param.sched_priority = priority;
        if (pthread_setschedparam(pthread_self(),SCHED_FIFO,(const struct sched_param *)&param))
        {
                fprintf(stderr, "pthread %d Cannot set priority %s¥n",
                pthread_self(),strerror(errno));
        }
#ifdef CCURRT
        cpumask = cpuset_alloc();
        cpuset_init(cpumask);
        cpuset_set_cpu(cpumask,cpu,1);/* 2nd CPU set */
        if ( mpadvise(MPA_PRC_SETBIAS,MPA_TID,0,cpumask)<0)
        {
                fprintf(stderr, "thread %d Cannot bind processor %s¥n",
                pthread_self(),strerror(errno));
        }
        if (mpadvise (MPA_PRC_GETRUN,MPA_TID,0,cpumask)!=MPA_FAILURE)
        {
                char * cpustr;
                cpustr = cpuset_get_string(cpumask);
                printf("Thread %d is ¥"%s¥" running¥n",*(int*)arg,cpustr);
                free(cpustr);
        }
        cpuset_free(cpumask);
```

```
#endif
        MUTEX_LOCK(&Mutex[0]);
        Event[0]++;
        MUTEX_UNLOCK(&Mutex[0]);
        do
        {
                MUTEX_LOCK(&Mutex[0]);
                thread_no = Event[0];
                MUTEX_UNLOCK(&Mutex[0]);
                nanosleep(&w0,0);
        } while (thread_no<THREAD_NO);
        if (pthread_sigmask(SIG_SETMASK,&eiset,&oset)==(-1))
        {
                fprintf(stderr, "Cannot set sigprocmask - %s¥n",
                strerror(errno));
        }
        do
        {
                sigsuspend(&eiset);
                if(GlobalCounter>=TESTNUMBER) loop_flag=0;
        } while (loop_flag);
        COND_SIGNAL(&Condition[1]);
        COND_SIGNAL(&Condition[2]);
        min=99999999999.0,max=0,avr=0;
        for(i=0;i<TESTNUMBER/2;i++)
        {
                if(min>SwitchTime1[i]) min = SwitchTime1[i];
                if(max<SwitchTime1[i]) max = SwitchTime1[i];
                avr += SwitchTime1[i];
        }
        printf("Switch1 %10.3f: %10.3f : %10.3f¥t",min,avr/(float)(TESTNUMBER/2),max);
        if (fabs(max-min)<=MAXJITTER)
        {
                printf("GOOD(fabs(%f)<%d micro sec)¥n",max-min,MAXJITTER);
        }
        else
        {
                printf("FAIL(fabs(%f)>%d micro sec)¥n",max-min,MAXJITTER);
        }

        min=99999999999.0,max=0,avr=0;
        for(i=0;i<TESTNUMBER/2;i++)
        {
                if(min>SwitchTime2[i]) min = SwitchTime2[i];
                if(max<SwitchTime2[i]) max = SwitchTime2[i];
                avr += SwitchTime2[i];
        }
        printf("Switch2 %10.3f: %10.3f : %10.3f¥t",min,avr/(float)(TESTNUMBER/2),max);
        if (fabs(max-min)<=MAXJITTER)
        {
                printf("GOOD(fabs(%f)<%d micro sec)¥n",max-min,MAXJITTER);
        }
        else
        {
                printf("FAIL(fabs(%f)>%d micro sec)¥n",max-min,MAXJITTER);
        }



        min=99999999999.0,max=0,avr=0;
        for(i=1;i<TESTNUMBER;i++)
        {
                if(min>IntervalTime[i]) min = IntervalTime[i];
                if(max<IntervalTime[i]) max = IntervalTime[i];
                avr += IntervalTime[i];
        }
        printf("Interval %10.3f: %10.3f : %10.3f¥t",min,avr/(float)(TESTNUMBER-1),max);
        if (fabs(max-min)<=MAXJITTER)
        {
                printf("GOOD(fabs(%f)<%d micro sec)¥n",max-min,MAXJITTER);
        }
        else
        {
```

```
                printf("FAIL(fabs(%f)>%d micro sec)¥n",max-min,MAXJITTER);
        }


        return(0);
}


void *thread_main1(void *arg)
{
        static int err=-1;
        int class = 3;                /* FIFO Schedule */
        int priority = 2;                        /* RealTime Priority min */
        static struct sched_param param;        /* RealTime Priority min */
        cpuset_t        *cpumask;
        cpu_t cpu = 1;                                /* boot PROCESSOR */

        if (pthread_getschedparam(pthread_self(),&class,&param))
        {
                fprintf(stderr, "thread %d Cannot get priority %s¥n",
                pthread_self(),strerror(errno));
        }
        param.sched_priority = priority;
        if (pthread_setschedparam(pthread_self(),SCHED_FIFO,(const struct sched_param *)&param))
        {
                fprintf(stderr, "thread %d Cannot set priority %s¥n",
                pthread_self(),strerror(errno));
        }
        if (pthread_sigmask(SIG_SETMASK,&diset,&oset)==(-1))
        {
                fprintf(stderr, "Cannot set sigprocmask - %s¥n",
                strerror(errno));
                err = errno;
                pthread_exit((void*)&err);
        }
#ifdef CCURRT
        cpumask = cpuset_alloc();
        cpuset_init(cpumask);
        cpuset_set_cpu(cpumask,cpu,1);/* CPU set */
        if ( mpadvise(MPA_PRC_SETBIAS,MPA_TID,0,cpumask)<0)
        {
                fprintf(stderr, "thread %d Cannot bind processor %s¥n",
                pthread_self(),strerror(errno));
        }
        if (mpadvise (MPA_PRC_GETRUN,MPA_TID,0,cpumask)!=MPA_FAILURE)
        {
                char * cpustr;
                cpustr = cpuset_get_string(cpumask);
                printf("Thread %d is ¥"%s¥" running¥n",*(int*)arg,cpustr);
                free(cpustr);
        }
        cpuset_free(cpumask);
#endif
        MUTEX_LOCK(&Mutex[0]);
        Event[0]++;
        MUTEX_UNLOCK(&Mutex[0]);
        do
        {
                MUTEX_LOCK(&Mutex[1]); /* lock */
                        if(Event[1]==0) COND_WAIT(&Condition[1],&Mutex[1]);
                        readtime(&EndTime);
                        if (GlobalCounter<TESTNUMBER)
adjust(&StartTime,&EndTime,&SwitchTime1[GlobalCounter/2]);
                        Event[1]--;
#if 0
                        printf("¥t¥tthread_main1 %d¥n",Event[1]);
#endif
                MUTEX_UNLOCK(&Mutex[1]); /* unlock */
                /* event1_exec(); */
        } while (loop_flag);
```

```
        err=0;
        pthread_exit((void *)err);
        return(0);
}
void *thread_main2(void *arg)
{
        static int err=-1;
        int class = 3;                                  /* FIFO Schedule */
        int priority = 3;                             /* RealTime Priority min */
        struct sched_param param;        /* RealTime Priority min */
        cpuset_t        *cpumask;
        cpu_t cpu = 1;                                  /* 2nd PROCESSOR */

        if (pthread_getschedparam(pthread_self(),&class,&param))
        {
                fprintf(stderr, "thread %d Cannot get priority %s¥n",
                pthread_self(),strerror(errno));
        }
        param.sched_priority = priority;
        if (pthread_setschedparam(pthread_self(),SCHED_FIFO,(const struct sched_param *)&param))
        {
                fprintf(stderr, "thread %d Cannot set priority %s¥n",
                pthread_self(),strerror(errno));
        }
        if (pthread_sigmask(SIG_SETMASK,&diset,&oset)==(-1))
        {
                fprintf(stderr, "Cannot set sigprocmask - %s¥n",
                strerror(errno));
                err = errno;
                pthread_exit((void*)&err);
        }
#ifdef CCURRT
        cpumask = cpuset_alloc();
        cpuset_init(cpumask);
        cpuset_set_cpu(cpumask,cpu,1);/* CPU set */
        if ( mpadvise(MPA_PRC_SETBIAS,MPA_TID,0,cpumask)<0)
        {
                fprintf(stderr, "thread %d Cannot bind processor %s¥n",
                pthread_self(),strerror(errno));
        }
        if (mpadvise (MPA_PRC_GETRUN,MPA_TID,0,cpumask)!=MPA_FAILURE)
        {
                char * cpustr;
                cpustr = cpuset_get_string(cpumask);
                printf("Thread %d is ¥"%s¥" running¥n",*(int*)arg,cpustr);
                free(cpustr);
        }
        cpuset_free(cpumask);
#endif
        MUTEX_LOCK(&Mutex[0]);
        Event[0]++;
        MUTEX_UNLOCK(&Mutex[0]);
        do
        {
                MUTEX_LOCK(&Mutex[2]); /* lock */
                        if (Event[2]==0) COND_WAIT(&Condition[2],&Mutex[2]);
                        readtime(&EndTime);
                        if (GlobalCounter<TESTNUMBER)
adjust(&StartTime,&EndTime,&SwitchTime2[GlobalCounter/2]);
                        Event[2]--;
#if 0
                        printf("¥t¥t¥t¥tthread_main2 %d¥n",Event[2]);
#endif
                MUTEX_UNLOCK(&Mutex[2]); /* unlock */
                /* event2_exec(); */
        } while (loop_flag);


       pthread_exit((void *)err);
        return(0);
}

void    *(*thread_main[THREAD_NO])(void *arg) =
```

```
{
        thread_main0,
        thread_main1,
        thread_main2
};



do_main(progname,fdin,filename,opt)
        char    *progname;
        FILE    *fdin;
        char    *filename;
        int opt;
{
        static int        no,thread_status[THREAD_NO],thread_arg[THREAD_NO];
        static pthread_t        tid[THREAD_NO];
        static pthread_attr_t        attributes[THREAD_NO];
        void *status_p;
        int signum;
        static struct sigaction newact;
        static sigset_t set,oset;
        static timer_t timer_id;

printf("************************************************* ¥n");
printf("**** posix timer & thread contexst switch test **** ¥n");
printf("************************************************* ¥n");


        if (opt)
        {
                system("/usr/bin/shield -a 1");
                system("/usr/bin/shield -c");
        }

        if (sigprocmask(0,NULL,&set)==(-1))
        {
                fprintf(stderr, "%s: Cannot get sigprocmask - %s¥n",
                progname,strerror(errno));
                exit(1);
        }
        sigaddset(&set,SIGRTMIN+2);
#if 1
        if (sigprocmask(SIG_SETMASK,&set,&oset)==(-1))
        {
                fprintf(stderr, "%s: Cannot set sigprocmask - %s¥n",
                progname,strerror(errno));
                exit(1);
        }
#endif
        signum = SIGRTMIN+2;
        printf("Create POSIX timer %d Hz¥n",CYCLE);
        timer_id = create_posix_timer(signum,0,1000000000/CYCLE);
        if (timer_id<0)
        {
                      fprintf(stderr, "%s: Cannot set posix timer - %s¥n",
                progname,strerror(errno));
                exit(1);
        }
        /* Set up the signal handler using sigaction(2) or sigset(2) */
        sigemptyset(&newact.sa_mask);
        sigaddset(&newact.sa_mask,signum);
        newact.sa_handler = interval_hadler;
        newact.sa_flags = SA_SIGINFO|SA_RESTART;
        if (sigaction(signum, &newact, 0)==(-1))
        {
                fprintf(stderr, "%s: Cannot set sigaction - %s¥n",
                progname,strerror(errno));
                exit(1);
        }

        sigfillset(&diset);
         sigfillset(&eiset);
        sigdelset(&eiset,SIGRTMIN+2);
```

```
        sigdelset(&eiset,SIGALRM);
        sigdelset(&eiset,SIGINT);
#if 0
        do
        {
                sigsuspend(&eiset);
                if(GlobalCounter>=TESTNUMBER) loop_flag=0;
        } while (loop_flag);
#endif

        printf("Create %d thread¥n",THREAD_NO);
        for (no=0;no<THREAD_NO;no++)
        {
                if (pthread_cond_init(&Condition[no],0)<0)
                {
                        fprintf(stderr, "%s: Cannot create condition - %s¥n",
                        progname,strerror(errno));
                        exit(1);
                }
                if (pthread_mutex_init(&Mutex[no],0)<0)
                {
                        fprintf(stderr, "%s: Cannot create condition - %s¥n",
                        progname,strerror(errno));
                        exit(1);
                }
                Event[no]=0;
        }
        for (no=0;no<THREAD_NO;no++)
        {
                if (pthread_attr_init(&attributes[no]))
                {
                    fprintf(stderr, "%s: Cannot set attributes - %s¥n",
                    progname,strerror(errno));
                    exit(1);
                }
                if (pthread_attr_setscope(&attributes[no],PTHREAD_SCOPE_SYSTEM))
                {
                    fprintf(stderr, "%s: Cannot set attributes - %s¥n",
                    progname,strerror(errno));
                    exit(1);
                }
                if (pthread_attr_setdetachstate(&attributes[no],PTHREAD_CREATE_JOINABLE))
                {
                    fprintf(stderr, "%s: Cannot set attributes - %s¥n",
                    progname,strerror(errno));
                    exit(1);
                }
                if (pthread_attr_setinheritsched(&attributes[no],PTHREAD_INHERIT_SCHED))
                {
                    fprintf(stderr, "%s: Cannot set attributes - %s¥n",
                    progname,strerror(errno));
                    exit(1);
                }
        }
        MUTEX_LOCK(&Mutex[0]); /* lock */
        for (no=1;no<THREAD_NO;no++)
        {
                thread_arg[no] = no;
                thread_status[no] = pthread_create(&tid[no],&attributes[no],thread_main[no],(void
*)&thread_arg[no]);
                if (thread_status[no])
                {
                        fprintf(stderr, "%s: Cannot create thread - %s¥n",
                        progname,strerror(errno));
                        exit(1);
                }
        }
        MUTEX_UNLOCK(&Mutex[0]); /* unlock */
        thread_arg[0] = 0;
        thread_main[0]((void *)&thread_arg[0]);
        timer_delete(timer_id);
        for (no=1;no<THREAD_NO;no++)
        {
```

```
                pthread_join(tid[no],&status_p);
        }
}


main(argc, argv)
        int argc;
        char **argv;
{
            extern int errno;
        extern int optind;
            extern char *optarg;
            register int i,c, opt = 0;


            progname = argv[0];

            while((c = getopt(argc, argv, "b")) != EOF){
                switch(c) {
                        case 'b':
                                opt =1;
                                break;
                            default:
                                fprintf(stderr,
                                "usage: %s [-b]¥n",
                                progname);
                                exit(1);
                    }
        }
#if 0
            if (optind != argc) {
                for (c = optind; c < argc; c++) {
                            if (access(argv[c], 4)) {
                                    fprintf(stderr,
                                "%s: ¥"%s¥" %s¥n",
                                 progname,argv[c],strerror(errno));
                                exit(1);
                            }
                }
            }
#endif
            if (optind == argc) {
                do_main(progname,stdin, "(stdin)",opt);
            } else {
                register FILE *fd;
                for (c = optind; c < argc; c++) {
                            if ((fd = fopen(argv[c], "w")) == NULL) {
                                    fprintf(stderr,
                                "%s: ¥"%s¥" %s¥n",
                                 progname,argv[c],strerror(errno));
                                exit(1);
                            }
                            do_main(progname,fd,argv[c],opt);
                        fprintf(fd,"Interval,Switch1,Switch2¥n");
                        for(i=0;i<TESTNUMBER/2;i++)
                        {
                                fprintf(fd,"%f,%f,%f¥n",IntervalTime[i+1], SwitchTime1[i], SwitchTime2[i]);
                        }
                        for(i=TESTNUMBER/2;i<TESTNUMBER-1;i++)
                        {
                                fprintf(fd,"%f,,¥n",IntervalTime[i+1]);
                        }
                            fclose(fd);
                }
            }
}
```