Using Linux Kernel Facilities from RT-threads

Nicholas McGuire
Name of Organization Inc,
A-2020 Mistelbach
der.herr@hofr.at

Abstract

RTLinux has been focused on developing a POSIX compliant RTOS layer that operates below Linux - within this development communication between RT-threads and kernel as well as user-space have been quite limited, in part due to the inherent restrictions of a RTOS and in part due to the restrictions imposed by POSIX, or rather the combination of these two sets of restrictions.

As RT-threads are operating in the same address-space as the Linux kernel itself it seems natural to investigate what capabilities within the Linux kernel could be made available to RT-threads as to enhance communication paths too and from user-space and non-rt kernel-space. In this paper a few of these, very non-portable, absolutely non-POSIX, paths are described. The main resources of interest to RT-threads being: Tasklets, Kernel Threads, Software interrupts, Sharing Memory, Accessing Non-RT facilities in kernel space, 'misusing' System calls.

Presenting a few simple examples the mechanisms and problems with utilizing these capabilities of the Linux kernel from within RT-context are discussed.

1 Introduction

In many realtime applications the main chalange for the programmer is to find the corect splitt between what is to be executed in rt-context and what can be executed in non-rt context. RTLinux has been splitting tasks into hard-realtime rt-context and non-rt user context, in many cases a more fine gain splitt is desired, allowing hard-rt and different levels of non-rt execution. The task of designing this splitt requires a basic understanding of the facilities available on the non-rt side of the system and how to comunicate with these. In this paper the focus is on accessing linux kernel facilities from rt-threads. User-space tasks and comunication with these are neglected as they are concidered sufficently documented in the standard rtlinux documentation.

Generally for all facilities that are available in the linux kernel the prime concern is if these can be safely called from rt-context or not. A simple rule is that anything that only involves bit operations `set_bit`, `test_and_set_bit`, `clear_bit` should be absolutly safe, any functions that require more complex synchronization need close analysis (or brute force testing) befor they can be deployed. As far as the authors analysis goes the kernel functions used in the examples here should be safe from rt-context with rtlinux-3.1 and kernel 2.4.4.

This paper ended up being more of a tutorial and a summary of available kernel facilities, never the less I hope that the concpts and the example codes are directly usable. The full source including the necessary Makefiles can be found on the proceedings CD, aswell as on the ftp site `ftp.fsmlabs.at`.

2 Tasklets

Tasklets are the replacement of the bottom half concept that was in use up to kernel 2.2.X (in 2.4.X BH are still supported - but are implemented via tasklets).

The main propoerties of tasklets:

- tasklets can be scheduled with different priorities in Linux
- tasklets don't need to be reentrant
- the same tasklet will never run in parallel on SMP
- scheduling a tasklet multiple times befor it actually runs does not cause it to run multiple times.

- different tasklets may run on different CPU's at the same time.
- tasklets run in interrupt context - thus with all limitations of an interrupt handler.

These propoerties make it fairly simple to write tasklets. The concept behind them is the same as with the former BH handlers, keep the interrupt or rt-thread small and put all processing steps that may be delayed into a tasklet.
Important for rtlinux is that tasklets are run at every context switch to linux, they are not delayed until the next hardware interrupt. Tasklets will run befor any user-space application will get a chance to run, thus they are a high priority non-rt task that can be easaly scheduled from within a rt-thread by calling `schedule_tasklet()` or `schedule_hi_tasklet`, whereby the later has higher priority than the first.

2.1 simple tasklet example

This first example is basically only a slightly modified version of `examples/hello/hello.c` the main change is the introduction of the tasklet code itselfe and the scheduling of the tasklet. Note that tasklets can be scheduled from rt-context and from linux kernel context without any conflict as the scheduling is performed by bit-operations which are atomic.
A tasklet is declared with the `DECLARE_TASKLET()` macro and scheduled with `schedule_tasklet` or `schedule_hi_tasklet`. The tasklet related macros are found in `linux/interrupts.h`.

```
#include <rtl.h>
#include <time.h>
#include <linux/interrupt.h>
    /* for the tasklet macros/functions */
#include <pthread.h>

int myint_for_something=1;
pthread_t thread;

void tasklet_function(unsigned long);

char tasklet_data[64];

DECLARE_TASKLET
    (test_tasklet,
    tasklet_function,
    (unsigned long) &tasklet_data);

void *
start_routine(void *arg)
{
    struct sched_param p;
    p . sched_priority = 1;
    pthread_setschedparam
        (pthread_self(), SCHED_FIFO, &p);

    pthread_make_periodic_np
        (pthread_self(), gethrtime(),
        500000000);

    while (1) {
        pthread_wait_np ();
        rtl_printf("RT-Thread;
            my arg is %x\n",
            (unsigned) arg);
        sprintf(tasklet_data,"%s \"%x\"",
            "Linux tasklet received
            RT-Thread arg",(unsigned) arg);
        tasklet_hi_schedule(&test_tasklet);
    }
    return 0;
}

void
tasklet_function(unsigned long data)
{
    struct timeval now;
    do_gettimeofday(&now);
    printk("%s at %ld,%ld\n",
        (char *) data,now.tv_sec,
            now.tv_usec);
}

int init_module(void) {
    sprintf(tasklet_data,"%s\n",
        "Linux tasklet called in
            init_module");
    tasklet_schedule(&test_tasklet);

    return pthread_create
        (&thread, NULL, start_routine, 0);
}

void
cleanup_module(void)
{
    pthread_delete_np (thread);
}
```

This simple example aside from schowing the basics of implementing a tasklet also allows to see the delay times between rt-threads and tasklets, if run with only one short rt-thread as in this example coupling is naturally very good - to see the realy coupling one could run this module together with the actual target application to get a fairly close picture of the delays introduced.

2.2 scheduling tasklets from rt-context

From `linux/interrupt.h`:

```
/* PLEASE, avoid to allocate new softirqs,
   if you need not _really_ high
   frequency threaded job scheduling.
   For almost all the purposes
   tasklets are more than enough.
   F.e. all serial device BHs et
   al. should be converted to tasklets,
   not to softirqs.
 */
```

The tasklet priority of a tasklet scheduled with `schedule_hi_tasklet` is above the network subsystem, so if you over due it you actually can criple your network performance..., `schedule_tasklet` has a priority just below the network subsystem so a network overload can delay your tasklet substantially.
With the kernel functions `tasklet_disable` and `tasklet_enable` the execution of a tasklet can be suspended. If a tasklet was scheduled and is disable befor it was executed it will be executed when `tasklet_enabled` is called. For the full set of kernel functions available for tasklets check `linux/interrupt.h`, note though that you must check if these are safe to be called from rt-context, for this paper checks were done agains linux 2.4.4.
To ensur synchronization of tasklet scheduling when disabling tasklets within rt-context with `tasklet_disable` one must install a cleanup handler to reenable the tasklet on termination of the thread so that a scheduled tasklets can be executed and the tasklet structure can be removed on module exit.

```
...
void
tasklet_cleanup(void *arg)
{
    tasklet_enable(&test_tasklet);
    rtl_printf("cleanup handler called\n");
}

void *
start_routine(void *arg)
{
    ...
    pthread_cleanup_push
        (tasklet_cleanup,0);

    while (1) {
        pthread_wait_np ();
        ...
        if(i==20){
            tasklet_disable(&test_tasklet);
            rtl_printf
                ("killed tasklet\n");
        }
        tasklet_hi_schedule
            (&test_tasklet);
        i++;
    }
    pthread_cleanup_pop(0);
    return 0;
}
```

This somewhat artificial code shows the basic setup - a cleanup handler to reenable the tasklet is installed and within the main loop of the rt-thread `tasklet_disable` is called to disable the `test_tasklet`, the cleanup handler is executed on termination of the `while(1)` loop and reenables tasklets.

2.3 naive rt-allocator

As a second somewhat more interesting example of using a tasklet from rt-context a naive `rt_allocator` framework is presented. The tasklet is called from a rt-function that suspends the running thread `rtl_malloc(size)`, this allocator will call a tasklet to do the actuall memory allocation and then signal `RTL_SIGNAL_WAKEUP` back to the rt-thread when the allocator thread is done. The allocation thus is non-realtime and the realtime thread needs to check if memory actually was allocated successfully or not. Note that the call to kmalloc in the tasklet uses the flags `GFP_ATOMIC` which is necessary, if `GFP_KERNELG` were used the tasklet could sleep and thus the system would hang.
This allocator has a automatic initialized array of pointers to char set and will allocate a requested size of memory assigned to these pointers. These are globally available so the tasklet can signal a wackup to the rt-thread by setting the appropriate bit in the threads pending singal mask. instead of setting the bit directly one could also call `pthread_kill (rt_thread,RTL_SIGNAL_WAKEUP)`, if modules are splitt between kernel and rtl context it sometimes is a problem to include rtl API-calls that require rtl-header files so in those cases directly accessing the singal pending mask solves the problem.

```
#include <rtl.h>
#include <time.h>
#include <pthread.h>

pthread_t rt_thread;
```

```
#include <linux/interrupt.h>
    /* for the tasklet
       macros/functions */
#include <linux/slab.h>
    /* kmalloc */

void allocator_function
    (unsigned long arg);
#define BUFFERS 128
static char *iptr[BUFFERS];
    /* static array of pointers
      for the buffers */
static int iptr_idx;
DECLARE_TASKLET
    (allocator_tasklet,
    allocator_function,0);


void
allocator_function(unsigned long arg)
{
    struct timeval now;
    do_gettimeofday(&now);
    printk("tasklet:
        allocating %ld at %ld,%ld\n",
        (unsigned long)arg,
        now.tv_sec,
        now.tv_usec);
    iptr[iptr_idx]=
        kmalloc((unsigned long)arg,
        GFP_ATOMIC);
    if(iptr[iptr_idx] == NULL){
        printk("tasklet:
            Allocation failed -
            out of memory\n");
    }
    else{
        memset(iptr[iptr_idx],
            0,
            (unsigned long)arg);
        printk("tasklet:
            Allocated 0'ed buffer
            %d (%ld bytes)\n",
            iptr_idx,
            (unsigned long)arg);
        iptr_idx++;
    }
    /* wake up the rt-thread that
       requested memory */
    set_bit(RTL_SIGNAL_WAKEUP,
        &rt_thread->pending);
}

unsigned long
rtl_kmalloc(unsigned long size)
{
    int idx;
    pthread_t self = pthread_self();
    RTL_MARK_SUSPENDED (self);
    rtl_printf("rtl_malloc:
        requesting %ld bytes\n",
        (unsigned long)size);
    /* if we are out of buffer
       pointers failed without
       calling the tasklet */
    idx = iptr_idx;
    if(idx < BUFFERS){
        allocator_tasklet.data=size;
        tasklet_hi_schedule
            (&allocator_tasklet);
        rtl_schedule();
        pthread_testcancel();
        if(iptr[idx] == NULL){
            return -1;
        }
        else{
            return idx;
        }
    }
    else{
        return -1;
    }
    return 0;
}

void *
start_routine(void *arg)
{
    struct sched_param p;
    int ret;
    unsigned long i,size,block;
    p . sched_priority = 1;
    pthread_setschedparam (
        pthread_self(),
        SCHED_FIFO,
        &p);

    pthread_make_periodic_np (
        pthread_self(),
        gethrtime(),
        500000000);

    size=0;
    block=128;
    i=1;
    while (1) {
        pthread_wait_np ();
        size=block*i++;
        rtl_printf("RT-Thread;
            requesting %ld bytes
```

```
                of memory\n",
            size);
        ret=rtl_kmalloc(size);
        /* apps must check that they
           actually got something */
        if(ret == -1){
            rtl_printf
                ("No more buffers
                     available\n");
        }
        else{
            rtl_printf
                ("allocated buffer
                    %d\n",ret);
        }
    }
    return 0;
}


int
init_module(void)
{
    int i;
    for(i=0;i<BUFFERS;i++){
        iptr[i] = NULL;
    }
    return pthread_create (
        &rt_thread,
        NULL,
        start_routine,
        0);
}

void
cleanup_module(void)
{
    int i;
    /* free all non-NULL buffers */
    for(i=0;i<BUFFERS;i++){
        if(iptr[i] != NULL){
            kfree(iptr[i]);
            printk("Freeing
                buffer %d\n",i);
        }
    }
    pthread_delete_np (rt_thread);
}
```

3 Kernel threads

kernel threads are a mechanism in the linux kernel that allow threads of execution to run in the kernels memory space (kernel context) but be visible as regular tasks that can receive signals and execute user-space calls with cirtain limitations/provisions. Here we are not so much interested with the details of kernel threads within the linux kernel itselfe but rather with how to interface rt-threads via kthreads to non-rt kernel-apce and user-space.

3.1 simple example

This first example is not rt-spceific, it only should give a framework of a kthread, this module declares a kernel function `exec_cmd` that is local to this module, a kernel thread is initiated passing this function as the routine to execute and a string via the arg pointer. The call to `kernel_thread()` initializes a task structure that is visible from user space (the pid of the process is printk'ed) and the thread routine (`exec_cmd`) is executed once. As we did not set up a specific context for this thread it runs in the inherited context of insmod and thus prints to the current console via the echo command. The thread routine is comparable to a regular user-space function that would call execve except for the privileges and the enabling of the kernels data section to store command arguments in `set_fs(KERNEL_DS)`. This also shows one clear danger of kernel threads - if they are not set up carfully with respect to privileges they can result in a serious security problem - for details on this give the kmod `kernel_thread` implementation in `kernel/kmod.c` a look.

```
#define __KERNEL_SYSCALLS__

#include <linux/config.h>
#include <linux/module.h>
#include <linux/sched.h>
#include <linux/unistd.h>
#include <linux/kmod.h>
#include <linux/errno.h>
#include <linux/unistd.h>
#include <linux/smp_lock.h>

#include <asm/uaccess.h>

int errno;

char cmd_path[256] = "/bin/echo";

static int
exec_cmd(void * kthread_arg)
{
    struct task_struct *curtask = current;
    /* we set up a minimum environment but
       note that we still inherit the
       environment of who ever launched
       insmod of this module sounds
       dangorous ? - it is !
```

```
     */
    static char *envp[] = {"HOME=/root ",
        "TERM=linux ",
        "PATH=/bin",
        NULL };
    char *argv[] = {
        cmd_path,
        kthread_arg,
        NULL };
    int ret;

    /* Give the kthread all effective
       privileges.. */
    curtask->euid = curtask->fsuid = 0;
    curtask->egid = curtask->fsgid = 0;
    cap_set_full(curtask->cap_effective);

    /* Allow execve args to be in
       kernel space. */
    set_fs(KERNEL_DS);

    printk("calling execve for
        %s \n",cmd_path);
    ret = execve(cmd_path, argv, envp);

    /* if we ever get here -
       execve failed */
    printk(KERN_ERR "failed to exec %s,
        ret = %d\n",
        cmd_path,ret);
    return -1;
}

int
init_module(void)
{
    pid_t pid;
    char kthread_arg[]=
        "Hello Kernel World !";

    pid = kernel_thread
        (exec_cmd,(void*)kthread_arg,0);
    if (pid < 0) {
        printk(KERN_ERR "fork failed,
            errno %d\n", -pid);
        return pid;
    }
    printk("fork ok, pid %d\n",pid);
    return 0;
}

void
leanup_module(void)
{
    printk("module exit\n");
}
```

3.2 comunicating with rt-threads

3.3 buddy thread concept

One of the many traditional communication mechanisms are singals. As rt-threads are operating in kerenl memory space and are not available via the linux kernel task-structure direct unix-signals from user-space applications to rt-threads are not posible. Posibilities shown in rtlinux examples where to install rt-handlers for fifos and trigger signals via these rt-fifos. In the folowing code an alternative concept that is intended to be expanded in the future is shown. This concept introduces a buddy-thread to each rt-thread that runs in kernel space as a kernel thread and thus is reachable directly from user-space via regular unix-signals. The signal is stll a two hopp job, a signal is set to the kthread indentified by the pid of the kernel process and passed on to the rt-thread via directly modifying the pending singals mask of the rt-thread structure or by using the rtlinux-API `pthread_kill` and `pthread_delete_np`.

```
#include <rtl.h>
#include <time.h>
#include <pthread.h>
#include <rtl_signal.h>
    /* RTL_SIGNAL_WAKEUP */
#include <linux/sched.h>
    /* flush_signals() */
#include <linux/init.h>

static pid_t kthread_id=0;
static wait_queue_head_t wait;
static int rt_thread_state=1;
    /* got to initialize it to != 0 */
#define ACTIVE 1
#define TERMINATED 0
static int state=ACTIVE;

#define NAME_LEN 16
static pthread_t rt_thread;

static void *
rtthread_code(void *arg)
{
    struct sched_param p;
    p . sched_priority = 1;
    pthread_setschedparam
        (pthread_self(), SCHED_FIFO, &p);

    while (1) {
        rtl_printf("RT-Thread woke up\n");
```

```
        pthread_suspend_np
            (pthread_self());
    }
    return 0;
}

static int
kthread_code( void *data )
{
    struct task_struct *kthread=current;
    char thread_name[NAME_LEN];

    memset(thread_name,0,NAME_LEN);
    daemonize();

    /* wait for pthread_create of
        the finisch so
        we are in sync */
    while (!rt_thread_state) {
        current->state =
            TASK_INTERRUPTIBLE;
        schedule_timeout(1);
    }

    /* take the address of the rt-thread
        as the uniq name */
    sprintf(thread_name,
        "rtl_%lx",
        (unsigned long)&rt_thread);
    strcpy(kthread->comm, thread_name);

    /* make it low priority */
    kthread->nice=20;

    /* clear all pending signals */
    spin_lock_irq(&kthread->sigmask_lock);
    sigemptyset(&kthread->blocked);
    flush_signals(kthread);
    recalc_sigpending(kthread);
    spin_unlock_irq
        (&kthread->sigmask_lock);

    /* wait for signals to pass on in
       an endless loop */
    while(1){
        interruptible_sleep_on(&wait);
        /* if we got a SIGKILL
           terminate the rt-thread and
         * exit the loop
         */
        if(sigtestsetmask
            (&kthread->pending.signal,
            sigmask(SIGKILL))  ){
            pthread_delete_np(rt_thread);
            break;
        }
        /* else send a RTL_SIGNLA_WAKEUP
           to the rt-thread and sleep on
         */
        else{
        pthread_kill
            (rt_thread,RTL_SIGNAL_WAKEUP);
            spin_lock_irq
                (&kthread->sigmask_lock);
            sigemptyset(&kthread->blocked);
            flush_signals(kthread);
            recalc_sigpending(kthread);
            spin_unlock_irq
                (&kthread->sigmask_lock);
        }
    }
    /* so cleanup module knows when to
       safely exit */
    state=TERMINATED;
    return(0);
}

int
init_module(void)
{
    init_waitqueue_head(&wait);
    kthread_id=kernel_thread(kthread_code,
        NULL,
        CLONE_FS|CLONE_FILES|
        CLONE_SIGHAND );
    printk("rt_sig_thread launched
        (pid %d)\n",
        kthread_id);
    rt_thread_state =
        pthread_create (&rt_thread,
        NULL,
        rtthread_code,
        0);
    return 0;
}

void
cleanup_module(void)
{
    int ret;

    /* delete the rt-thread */
    pthread_delete_np (rt_thread);

    /* send a term signal to the kthread */
    ret = kill_proc
        (kthread_id, SIGKILL, 1);
    if (!ret) {
        int count = 10 * HZ;
        /* wait for the kthread to exit
```

```
           before terminating */
        while (state && --count) {
            current->state =
                 TASK_INTERRUPTIBLE;
            schedule_timeout(1);
        }
    }
    printk("rt_sig_thread exit\n");
}
```

4 Shared memory

Many rt-processes need to share data with non-rt processes or the non-rt Linux kernel. For this purpos the rt-extensions to Linux made use of a shared memory module contributed by Tomas Motylevsky. In this section we are not concerned with this module which is part of RTAI and RTLinux, but rather with sharing memory via mechanisms available from the Linux kenrel.
The one way to share memory with rt-space is to add a character device that need not provide more than the open/releas and mmap function in the fops an use a kmalloc'ed area that then can be shared, alternatively one can make use of the memory devices in Linux, mmaping `/dev/mem`.
Simple mmap driver:
The simplest method of having shared memory for your RTLinux system is to set up a dummy character device (or drop it into any real device that you need for your system) and provide a mmap call allowing to access a kmalloc'ed area via the mmap system call.

```
#include <rtl.h>
#include <time.h>
#include <pthread.h>
#include <rtl_signal.h>
    /* RTL_SIGNAL_WAKEUP */
#include <linux/sched.h>
    /* flush_signals() */
#include <linux/module.h>
#include <linux/version.h>
#include <linux/init.h>

#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/errno.h>
#include <linux/mm.h>
#include <linux/malloc.h>
#include <linux/mman.h>
#include <linux/slab.h>
#include <linux/wrapper.h>

#include <asm/io.h>
#include <asm/uaccess.h>

static pthread_t rt_thread;

#define DRIVER_MAJOR 17

#define LEN 4096
static char *kmalloc_area;

static void *
rtthread_code(void *arg)
{
    struct sched_param p;
    p . sched_priority = 1;
    pthread_setschedparam
        (pthread_self(), SCHED_FIFO, &p);

    pthread_make_periodic_np
        (pthread_self(),
            gethrtime(), 500000000);

    while (1) {
        pthread_wait_np();
        rtl_printf ("RT-Thread
            current buffer=%s\n",
            kmalloc_area);
    }
    return 0;
}

static int
driver_open(struct inode *inode,
    struct file *file )
{
    MOD_INC_USE_COUNT;
    return 0;
}

static int
driver_close(struct inode *inode,
    struct file *file)
{
    MOD_DEC_USE_COUNT;
    return 0;
}

static int
driver_mmap(struct file *file,
    struct vm_area_struct *vma)
{
    vma->vm_flags |= VM_SHARED|VM_RESERVED;

    if(remap_page_range(vma->vm_start,
        virt_to_phys(kmalloc_area),
        LEN,
        PAGE_SHARED))
```

```
        {
            printk("mmap failed\n");
            return -ENXIO;
        }
    return 0;
}


static struct file_operations simple_fops={
    mmap:    driver_mmap,
    open:    driver_open,
    release:    driver_close,
};

static int __init simple_init(void)
{
    struct page *page;
    int ret;
    kmalloc_area=kmalloc(LEN,GFP_USER);
    if(!kmalloc_area){
        printk("kmalloc failed -
            exiting\n");
        return -1;
    }
    page = virt_to_page(kmalloc_area);
    mem_map_reserve(page);
    memset(kmalloc_area,0,LEN);

    if(register_chrdev
        (DRIVER_MAJOR,"simple-driver",
        &simple_fops) == 0) {
        printk("driver for major %d
            registered successfully\n",
            DRIVER_MAJOR);
        ret = pthread_create
            (&rt_thread,
            NULL,
            rtthread_code,
            0);
        return 0;
    }
    printk("unable to get major %d\n",
        DRIVER_MAJOR);
    return -EIO;
}

static void __exit simple_exit(void)
{
    pthread_delete_np (rt_thread);

    unregister_chrdev
        (DRIVER_MAJOR,"simple-driver");
    kfree(kmalloc_area);
}

module_init(simple_init);
module_exit(simple_exit);
```

Using /dev/mem

The POSIX way of sharing memory is via /dev/mem - you can pass it an offset of 0 and let the kernel select where to place the shared buffer, or you can allocate a buffer and pass the address and size to the user-space side and then use /dev/mem to mmap it to the user-space app. In the given example we simply pass 0 and let the kernel take care of it.

```
#include <rtl.h>
#include <time.h>
#include <rtl_debug.h>
#include <errno.h>
#include <pthread.h>

#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>

pthread_t thread;

struct shared_mem_struct
{
    int some_int;
    char ready;
};

int memfd;

#define MEMORY_OFFSET 0

struct shared_mem_struct* shared_mem;

void
cleanup(void *arg)
{
    printk("Cleanup handler called\n");
}

void * start_routine(void *arg)
{
    struct sched_param p;
    p . sched_priority = 1;
    pthread_setschedparam
        (pthread_self(), SCHED_FIFO, &p);

    pthread_make_periodic_np
        (pthread_self(),
        gethrtime(), 500000000);
    pthread_cleanup_push(cleanup,0);

    while (1) {
        hrtime_t now;
        pthread_wait_np ();
```

```
        now = gethrtime();
        rtl_printf("I'm here;
            my shared mem=%d\n",
            shared_mem->some_int);
    }
    pthread_cleanup_pop(0);
    return 0;
}

int
init_module(void)
{
    int ret;

    memfd = open("/dev/mem", O_RDWR);
    if (memfd){
        shared_mem =
            (struct shared_mem_struct*)
             mmap(0,
             sizeof
            (struct shared_mem_struct),
            PROT_READ | PROT_WRITE,
            MAP_FILE | MAP_SHARED,
            memfd, MEMORY_OFFSET);
        if(shared_mem != NULL){
            printk("Dev mem available\n");
        }
        else{
            printk("Failed to map
                memory\n");
            close (memfd);
            return -1;
        }
    }
    else{
        printk("Failed to open memory
            device file\n");
        return -1;
    }
    ret=pthread_create
        (&thread, NULL, start_routine, 0);
    return ret;
}

void cleanup_module(void) {
    pthread_delete_np (thread);
    close(memfd);
}
```

The user space side simply opens `/dev/mem/` an mmaps the offset 0 address.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include "device_common.h"

int
main(void)
{
    int fd;
    char msg[LEN];
    unsigned int *addr;

    if((fd=open
        (SIMPLE_DEV, O_RDWR|O_SYNC))<0)
    {
        perror("open");
        exit(-1);
    }
    addr = mmap(0, LEN,
        PROT_READ|PROT_WRITE,
        MAP_SHARED, fd, 0);

    printf("enter a short test:");
    scanf("%s",&msg);

    if(!addr)
    {
        perror("mmap");
        exit(-1);
    }
    else
    {
        memset(addr,0,LEN);
        strncpy(addr,msg,sizeof(msg));
        printf("Put: %s\n",addr);
    }
    munmap(addr,LEN);
    close(fd);
    return 0;
}
```

Using reserved 'raw'-memory

You can map reserved physical memory by passing the kernel a mem=126m line at the boot: prompt and then mmap'ing it via `/dev/mem` (this assumes you have 128m of physical memory installed and want to dedicate 2MB to RTLinux). Not a very elegant way to do it - but a very simple way if you need large blocks of contingous memory. Linux's kmalloc, that provides contingous memory, is limited to 128kB as maintaining a buddy-system up to 2MB would be a tremendous waste of resources, so contingous memory is limited to de-facto 128kB if you use the Linux kernel memory functions to allocate memory (vmalloc is non-contingous - and not limited to 128kB). There is no need to do any magic for the kernel side

to access this area, simply user the physical address of 126*0x100000 as the base address of the 126th MB and manage it on your own.

5 Accessing kernel facilities

6 System calls

This is a sample implementation of a system call - system calls are fairly fast compared to device open/read/close operations that need to traverse the VFS and execute a few system calls sequescially, but it is the mosts non-portable and the most dangorous solution to a problem posible, changing a system call or introducing a new one makes your system as a whole incompatible to all other linux system. Adding a system call can introduce a serious security problem in your system. Adding a system call will require you to patch every kernel releas when updating. So the best solution is not to write your own system calls.... but they solve problems some times ;)
The actual syscall code is quite simple, and placed in `/usr/src/linux/arch/i386/kernel/sys_i386.c` for our purposes, naturally if your system call code is more elaborate then you should put it into an independant file.

```
asmlinkage int sys_test_call(void)
{
    /* do something usefull in kernel
        space - like a printk */
    printk("Test System Call called \n");
    return 0;
}
```

This system call will only produce a printk output and thats it - system calls have a fixed number of parameters and types that must be declared, in the above case the system call takes no arguments at all. The number of arguments not only needs to be given with the declaration of the system call but also with the prototype declaration which is a littl bit different than regular prototype declarations (see below).
The kernel has a "jump-matrix" for the system calls - the position of a system call in the syscall table is absolut so you can't add in your system call at the beginning or in the middle or your will break the entire system, if at all add it at the end of the syscall table. The position in the syscall table is the syscall number. So put it into the syscall table like:

```
/usr/src/linux/arch/i386/kernel/entry.S
    ...
    .long SYMBOL_NAME(sys_getdents64)
        /* 220 */
    .long SYMBOL_NAME(sys_fcntl64)
    .long SYMBOL_NAME(sys_test_call)
```

Note that this system call table may change over time - so you will have to patch newer kernels with your system call and modify the code that is calling the syscall since the number may have changed - it is up to you to maintain your system call.
If you want to put your syscall at a position beond the last current system you must fill up the system call table with empty system calls:

```
    .long SYMBOL_NAME(sys_ni_syscall)
```

after recompiling your kernel you could now call it with the absolute system call number, to be a bit more user frindly you need to add some entries to make it available to user space apps via `asm/unistd.h`:

```
/usr/include/asm/unistd.h
#define __NR_test_call        222
     /* this number better be the same
        as the position in entry.S !! */
```

Now a regular system call like `open` is simply called by

```
   fd=open(".....
```

our system call could also be called in this way but that would require recompiling glibc aswell, as during the build process of glibc the kernels syscall table is read - if you do recompile glibc then you have reached the maximum posible incompatibility to any other linux system. If you don't want to recompile glibc, which is probably a good idea, then you need to put the prototype declaration for your system call into the source file.
So assuming we did not recompile libc, call it in in a c-source file like:

```
---syscall.c---
#include <asm/unistd.h>
#include <errno.h>

_syscall0(int,test_call);

main(){
    syscall(222);
        /* call it via syscal
           (SYSCAL_NUMBER) */
    test_call();  /* call it by name */
    return 0;
}
```

compile with simple gcc syscall.c -o syscall and run this program as `./syscall`. To check kernel output (the printk that our syscall is to do) use the dmesg command - it should have produced:
Test System Call called
Test System Call called
the two calls are via syscall(222) and `test_call()` - note that you don't need the headerfiles errno.h and asm/unistd.h to use syscall(222) but you do need these includes for the named call `test_call();` Using syscall(222) can be very confusing as it says nothing about what you are trying to do, so give your system call a meaningfull name.