Realtime Application Interface (RTAI)

The Basics of Real-Time Linux

Real-Time Linux Introduction

- Linux is a free Unix-like operating system that runs on a variety of platforms, including PCs. Numerous Linux distributions such as Red Hat, Debian and Mandrake bundle the Linux OS with tools, productivity software, games, etc.
- The Linux scheduler, like that of other OSes such as Windows or MacOS, is designed for best average response, so it feels fast and interactive even when running many programs.
    - However, it doesn't guarantee that any particular task will always run by a given deadline. A task may be suspended for an arbitrarily long time, for example while a Linux device driver services a disk interrupt.
    - Scheduling guarantees are offered by real-time operating systems (RTOSes), such as QNX, LynxOS or VxWorks. RTOSes are typically used for control or communications applications, not general purpose computing.
- Linux has been adapted for real-time support. These adaptations are termed "Real-Time Linux" (RT Linux).
- Numerous versions of RT Linux are available, free or commercial. Two commonly available free RT Linux versions are
    - the Real-Time Application Interface (RTAI), developed by the Milan Polytechnical University and available at www.aero.polimi.it/~rtai/
    - RTL, developed by New Mexico Tech and now maintained by FSM Labs, Inc., with a free version available at www.rtlinux.org.
- These RT Linux systems are patches to the basic Linux kernel source code. Instructions for building an RT Linux system from a the Linux source code are provided with these RT Linux systems. Briefly the process involves setting up the basic Linux system, getting the latest Linux kernel source code from www.kernel.org, patching the kernel source code, and compiling the patched kernel.
- The Realtime kernel, all their component parts, and the real time application are all run in Linux kernel address spaces as kernel modules.

Objectives

The following are the primary objectives of this exercise:

- To teach the student how to build and run kernel modules.

RT Linux Tasks are Kernel Modules, not Linux Programs

- In both the RTAI and RTL flavors of RT Linux, the principle is the same: a real-time scheduler replaces the original Linux scheduler, and
    - intercepts the timer interrupt and external device interrupts,
    - runs any real-time code associated with these, and
    - runs any normal Linux processes in the time left over.
- RT Linux tasks are "kernel modules," meaning they run as part of the privileged Linux kernel, similar to

device drivers.

- Kernel modules, like device drivers, execute in a primitive environment, without direct access to many user-level Linux facilities like terminal or file I/O.
    - Any reasonably large real-time application will want these features, and these applications typically are split between real-time and non-real-time parts, with the non-real-time parts using all the nice Linux features.
    - Communication between the two parts can be accomplished numerous ways, using shared memory, first-in, first-out (FIFO) queues and other communication pathways we will describe later.
    - This split complicates RT Linux programming, in comparison with other real-time operating systems in which everything runs in real-time.
    - On the other hand, the simpler environment of kernel modules makes RT Linux quite fast.
- Warning! Since your real-time code is running as a kernel module, it is effectively as privileged as the kernel. Unlike with regular Linux processes, coding errors may crash the system, requiring a reboot.

  "With great power comes great responsibility." *Uncle Ben to Peter Parker, aka Spider-Man.*

Kernel Modules are Dynamically Loaded

Kernel modules are dynamically

- loaded using the '**insmod**' insert module program, and
- unloaded (stopped), using the '**rmmod**' remove module program.

These programs are only available to the root user (administrator), although there are ways to avoid giving real-time programmers true root access to the system.

C is the Preferred Language

- Linux is written in the C programming language (with some assembly language), as is RTAI, so C is the preferred language for writing RT Linux programs.
- Other languages (e.g., C++) can be used, but there are caveats that make C the much-preferred language for RTAI Linux development.
- The examples in this tutorial are all written in C.
- C programs are normally compiled into full executable programs, but kernel modules are compiled into object code, with final linking suppressed.
  Instead of a full-blown executable, your code will be a loadable kernel object **'.ko'** file, possibly the result of linking together several other '.o' files if your project is split into numerous files for convenience or clarity.
- In C, a program's "entry point" where execution begins is a function called 'main()'. For a kernel module, this entry point is called '**init_module()**'. 'insmod' looks for this symbol when loading the code.
- A program's "exit point" is a function called '**cleanup_module()**'. This will be called when 'rmmod' removes the kernel module.

Here is the "hello.c" C code that illustrates this:

```
/* hello.c */
#include <linux/kernel.h>
#include <linux/module.h>
MODULE_LICENSE("GPL");
int init_module(void)
{
  printk("Hello world!\n"); /* printk = kernel printf, to the console */
  return 0;
}
void cleanup_module(void)
```

```
{
  printk("Goodbye world!\n");
  return;
}
```

Note that we used the command '**printk()**' instead of the familiar 'printf()' command. This is because modules run in so called kernel space which can not use linux API in the so called user space where you run normal applications. Therefore **the kernel has its own special API** where printk is part of it.

The Mechanics of Compiling and Running a kernel Module

- The mechanics of compiling C code vary depending upon which compiler you use.
- The Free Software Foundation's Gnu C compiler 'gcc' is installed with most Linux distributions.
- In the newer linux versions, kernel modules are build using the kernel build system 'kbuild', which uses a so called '**Kbuild**' makefile to make a module (however 'Makefile' will also be continued to be supported). Creating a kbuild make is very simple and a single line will suffice:

```
# Kbuild file
obj-m := hello.o
```

- This uses the special syntax of the kernel build system and means that there is one module to be built from the object file hello.o
- When building a module module.ko from multiple source files, you would have a Kbuild file like:

```
# Kbuild file
obj-m := module.o
module-objs := file1.o file2.o ...
```

- More extended documentation of this kbuild system can be found in the linux kernel source under the directory **${kernel-src}/Documentation/kbuild/**. (The kernel source can be found at /usr/src/linux)
- To build the module with the kbuild system you need to specify the kernel source directory, because with the current build system (2.6. kernel) one cannot build a module without using the kernel source tree. For instance, the build command for a module in directory /root/exercises/ex01 is:

```
make -f Makefile -C /usr/src/linux M=/root/exercises/ex01
```

- This executes the following steps:
  - The '-C' option is to move to the kernel source directory (/usr/src/linux), where the root Makefile is located.
  - In this kernel directory "make" is executed, i.e. using Makefile
  - The 'M' option informs this make process to only build the module code in the current working directory and 'make' will move back to your module source directory after the build.
- However typing this long line is tiresome. Luckily there is an solution in writing you local makefile, called Kbuild, as follows:

```
# Kbuild file
ifeq ($(KERNELRELEASE),)
   # Assume the source tree is where the running kernel was built
   # You should set KERNELDIR in the environment if it's elsewhere
   KERNELDIR ?= /lib/modules/$(shell uname -r)/build
   # The current directory is passed to sub-makes as argument
   PWD := $(shell pwd)
modules:
   $(MAKE)  -C $(KERNELDIR) M=$(PWD) modules
modules_install:
   $(MAKE) -C $(KERNELDIR) M=$(PWD) modules_install
```

```
clean:
    rm -rf *.o *~ core .depend .*.cmd *.ko *.mod.c .tmp_versions
.PHONY: modules modules_install clean
else
    # called from kernel build system: just declare what our modules are
    obj-m := hello.o
endif
```

- This makefile will first set the correct path and rerun itself with the right options. Now to build the module 'hello' you can simply type :

```
make -f Kbuild
```

- The result will be a kernel module called 'hello.ko'.
- To start to use this module you can load it by running:

```
insmod hello.ko
```

On loading, the init_module() function will be called.

- With the command 'lsmod' one can see if the module is loaded.
- Finally one can unload the module with :

```
rmmod hello.ko
```

On unloading, the cleanup_module() function will be called.

- When running the programs in vmware, the output is displayed, but when running then on embedded processors the output is not immediately visible. This is because for kernel modules all output goes directly to the system log file **/var/log/messages**.
    - One can follow the output in /var/log/messages easily with the 'trace -f /var/log/messages' command. Another option is to use the 'dmesg -c' command to clear the kernel ring buffer contents after printing it, as follows:

```
> dmesg -c >& /dev/null
> insmod hello.ko
> dmesg -c
Hello world!
> rmmod hello.ko
> dmesg -c
Goodby world!
```

Exercises

Exercise 1.

- Have a look at the directory /usr/src/linux, see also the documentation, for instance of 'kbuild'.
- Compile and run the above 'Hello world' example to get some experience in working with kernel modules.

Multi-Tasking

Introduction

Modern real-time systems are based on the complementary concepts of multitasking and intertask communications. A multitasking environment allows real-time applications to be constructed as a set of independent tasks, each with a separate thread of execution and its own set of system resources. The intertask communication facilities allow these tasks to synchronize and coordinate their activities. The RTAI multitasking scheduler, uses **interrupt-driven, priority-based task scheduling**. It features fast context switch time and low interrupt latency.

Objectives

The following are the primary objectives of this exercise:

- To teach the student how to initiate multiple processes using RTAI tasking routines.

Description

Multitasking creates the appearance of many threads of execution running concurrently when, in fact, the kernel interleaves their execution on the basis of a scheduling algorithm. Each apparently independent program is called a *task*. Each task has its own *context*, which is the CPU environment and system resources that the task sees each time it is scheduled to run by the kernel. On a *context switch*, a task's context is saved in the *Task Control Block*(TCB). A task's context includes:

- a thread of execution, that is, the task's program counter
- the CPU registers and floating-point registers if necessary
- a stack of dynamic variables and return addresses of function calls
- I/O assignments for standard input, output, error
- a delay timer
- a timeslice timer
- kernel control structures
- signal handlers
- debugging and performance monitoring values

Scheduling in RTAI Linux

- RTAI uses a MultiUniProcessor (MUP) scheduling scheme. Applications using RTAI schedulers can be timed by three type of hard timers:
  - 8254
  - APIC
  - Real Time Clock (RTC).
- RTC allows just a periodic ticking at a frequency that must be set when configuring RTAI. Its frequency can range from 2 to 8192 Hz in power of 2 always. RTC will be the only available timer and its tick frequency can be modified only by reconfiguring RTAI. RTC constraints:
  1 - be happy of working in periodic mode; moreover
  2 - it must be acceptable to resolve your timing needs with a tick within a frequency that can be

varied just in powers of 2, from 2 to 8192 Hz.

- The main advantage of using the RTC, if constraints 1 and 2 can be accepted, is that you will not touch any of the hard timing sources used by Linux (8254 especially and APIC too). In fact if an APIC timer is available (MP and recent machines) it is possible to avoid the most important interference with Linux timing by not using the 8254. However such a possibility is not available on low end CPUs and embedded systems for which the only way to avoid interfering with the Linux timer might be this option.
- 8254 and APIC allow using the oneshot mode also. In oneshot mode the timer is fired aperiodically as needed to wake up any timed test with a priority higher or equal to the running task. Special care is taken in maintaining Linux timing also when the 8254 is used.
- The oneshot mode needs a continuous running timer to be read to compute the aperiodic counts to be loaded into the used timer counter. In the APIC case it is granted that a Time Stamp Clock (TSC), running at the CPU frequency on x86s, is always available and the oneshot mode will be highly effective.
- Conversions from the TSC to the APIC timer frequency apart, the oneshot mode with APIC is the most effective and flexible way to time an RTAI application, mostly at a submicrosec resolution (in theory).
- When the 8254 is used instead and a TSC is at hand the situation is not much different in relation to the resolution, moreover it is more jitter prone than programming the APIC oneshot counter. Nonetheless it is believed that if a TSC can be used the oneshot mode remains a very flexible and useful timing mode even with 8254 timers.
- When a true TSC is not available, as on 486s and false 586s, RTAI uses a form of emulation of the TSC based on counter2 of the 8254. So you can use RTAI also on such machines but must be warned that the oneshot timer on 486s is a bit of a performance killer, because of the need of reading the emulated TSC, i.e. 8254 counter2 in this case.
- Unique feature of RTAI on MP is the possibility of using mixed timers simultaneously, i.e. periodic and oneshot, where periodic timers can be based on different periods.

Starting the Timer and Realtime Scheduler

The first thing we always have to do at the start of a real-time program is to start the real-time timer and the scheduler using the *rt_request_timer* command.

- We start the timer to expire in **pure periodic mode** by calling the functions

  void rt_set_periodic_mode(void);
  int rt_request_timer (void(*)(void) *handler*, unsigned *tick*, int *use_apic*);
  (or)
  RTIME start_rt_timer(RTIME period);

  rt_request_timer requests a timer of period tick ticks, and installs the routine *handler* as a real time interrupt service routine for the timer. If *apic* has a nonzero value the local APIC timer is used. Otherwise timing is based on the 8254.

  The value passed to 'start_rt_timer()' in periodic mode is the desired base period, specified in internal counts (i.e. 8354 ticks). The return value of type RTIME yields the realized number of counts. The function *rt_get_time()* returns the numbers of internal count units (i.e., 8254 ticks) passed since *start_rt_timer* was called.

- We start the timer to expire in **one-shot mode** by calling the functions

  void rt_set_oneshot_mode(void);
  int rt_request_timer (void(*)(void) *handler*, unsigned *tick*, int *use_apic*);
  (or)
  RTIME start_rt_timer(RTIME period);

  The *tick* argument for *rt_request_timer* is set to zero, therefore be ignored.

- The argument for *start_rt_timer* is chosen randomly because we don't have a period, and it will therefore be ignored by *rt_start_timer*. So we can for instance use *start_rt_timer(1)*. Function *rt_get_time()* again returns the numbers of internal count units, now this corresponds to the TSC count.

The function

```
RTIME nano2counts(int nanoseconds);
```

can be used to convert time in nanoseconds to these RTIME internal count units. In periodic mode this is quantized to the resolution of the 8254 chip frequency (1,193,180 Hz).

The Task Function

Each task is associated with a function that is called when the task is scheduled to run. This function is a usual C function running in period mode that typically reads some inputs, computes some outputs and waits for the next cycle. Such a task function should enter an endless loop, in which is does its work, and then calls

```
void rt_task_wait_period(void);
```

to wait for the its next scheduled cycle. Typical code looks like this:

```
void task_function(int arg)
{
  while (1) {
    /*
      Do your thing here
     */

    rt_task_wait_period();
  }

  return;
}
```

Setting Up the Task Structure

- An RT_TASK data structure is used to hold all the information about a task:
  - the task function,
  - any initial argument passed to it,
  - the size of the stack allocated for its variables,
  - its priority,
  - whether or not it uses floating-point math,
  - and a "signal handler" that will be called when the task becomes active.
- The task structure is initialized by calling

```
rt_task_init(RT_TASK *task,
             void *rt_thread, int data,
             int stack_size, int priority,
             int uses_fp, void *sig_handler);
```

  - 'task' is a pointer to an RT_TASK type structure which must have been declared before and whose structure is filled.
  - 'rt_thread' is the entry point of the task function.
  - 'data' is a single integer value passed to the new task.
  - 'stack_size' is the size of the stack to be used by the new task.

    - 'priority' is the priority to be given the task. The highest priority is RT_SCHED_HIGHEST_PRIORITY (which equals 0), while the lowest is RT_SCHED_LOWEST_PRIORITY (which equals 1,073,741,823) (both are defined in rtai_sched.h).
    - 'uses_fp' is a flag. Nonzero value indicates that the task will use floating point, and the scheduler should make the extra effort to save and restore the floating point registers.
    - 'sig_handler' is a function that is called, within the task environment and with interrupts disabled, when the task becomes the current running task after a context switch.
- The newly created real time task is initially in a suspended state. It is can be made active either with 'rt_task_make_periodic()', 'rt_task_make_periodic_relative_ns()' or 'rt_task_resume()'.

Scheduling the Task

periodic mode

- The task can now be started by passing a pointer to the initialized RT_TASK structure to the function

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

    - 'task' is the address of an earlier initialized RT_TASK structure,
    - 'start_time' is the absolute time, in RTIME units, when the task should begin execution. Typically this is "now" (i.e., calling 'rt_get_time()').
    - 'period' is the task's period, in RTIME units, which will be rounded to the nearest multiple of the base period.
- The task will now execute indefinitely every 'period' counts.

one-shot mode

- The task can be simply started immediately by calling the function

```
int rt_task_resume(RT_TASK *task)
```

- However the task can be started after a delay with

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

    - 'task' is the address of the RT_TASK structure,
    - 'start_time' is the absolute time, in RTIME units, when the task should begin execution. Typically this is "now" (i.e., calling 'rt_get_time()').
    - 'period' is now a dummy value which is not used by the scheduler in one-shot mode

Example : Helloworld!

```
#include <linux/kernel.h>   /* decls needed for kernel modules */
#include <linux/module.h>   /* decls needed for kernel modules */
#include <linux/version.h>  /* LINUX_VERSION_CODE, KERNEL_VERSION() */
#include <linux/errno.h>    /* EINVAL, ENOMEM */

/*
  Specific header files for RTAI, our flavor of RT Linux
 */
```

```
#include "rtai.h"       /* RTAI configuration switches */
#include "rtai_sched.h"     /* rt_set_periodic_mode(), start_rt_timer(),
                nano2count(), RT_SCHED_LOWEST_PRIORITY,
                rt_task_init(), rt_task_make_periodic() */
#include <rtai_sem.h>

MODULE_LICENSE("GPL");

static RT_TASK print_task;  /* we'll fill this in with our task */

void print_function(long arg)     /* Subroutine to be spawned */
{
  rt_printk("Hello world!\n"); /* Print task Id */
  return;
}

int init_module(void)
{
   int retval;         /* we look at our return values */

   /*
    Set up the timer to expire in one-shot mode by calling

    void rt_set_oneshot_mode(void);

    This sets the one-shot mode for the timer. The 8254 timer chip
    will be reprogrammed each task cycle, at a cost of about 2 microseconds.
   */
   rt_set_oneshot_mode();

   /*
    Start the one-shot timer by calling

    RTIME start_rt_timer(RTIME count)

    with 'count' set to 1 as a dummy nonzero value provided for the period since
    we don't have one. We habitually use a nonzero value since 0
    could mean disable the timer to some.
   */
   start_rt_timer(1);

   /*
    Create the RT task with rt_task_init(...)
    */
   retval =
   rt_task_init(&print_task,   /* pointer to our task structure */
     print_function, /* the actual  function */
     0,       /* initial task parameter; we ignore */
     1024,        /* 1-K stack is enough for us */
     RT_SCHED_LOWEST_PRIORITY, /* lowest is fine for our 1 task */
     0,       /* uses floating point; we don't */
     0);      /* signal handler; we don't use signals */
   if (0 != retval) {
      if (-EINVAL == retval) {
         /* task structure is already in use */
         printk("task: task structure is invalid\n");
      } else {
         /* unknown error */
```

```
            printk("task: error starting task\n");
        }
        return retval;
    }

    /*
     Start the RT task with rt_task_resume()
     */
    retval = rt_task_resume(&print_task); /* pointer to our task structure */
    if (0 != retval) {
        if (-EINVAL == retval) {
            /* task structure is already in use */
            printk("task: task structure is invalid\n");
        } else {
            /* unknown error */
            printk("task: error starting task\n");
        }
        return retval;
    }

    return 0;
}

void cleanup_module(void)
{
  // task end themselves -> not necessary to delete them
  return;
}
```

Exercises

Exercise 2a.

- Build the Helloworld module above.
- Note that just loading it with insmod leads to errors about unknown symbols.
- Nest try the following sequence: load first the rtai modules with "rtai-insmod.sh", next load the hello module ("insmod hello.ko"), and finally unload all ("rmmod hello.ko", "rtai-rmmod.sh").
- Try the shell script 'run' which combines loading and unloading. Its usage is :

```
> ./run -h

usage:
  ./run [-v|h] [module]

  module : name of module to load (without .ko extension)
  -h : prints this message
  -v : verbose output, useful when not working on a console
```

  remarks :

  - When no module argument is given, it uses its default module which is set with a variable inside the run shell script. For the run scripts coming with the exercises this variable is already set to the right module.

-
-

-

Exercise 2b.

Write a program which runs ten tasks in one-shot mode running the following function :

```
-------------------------------------------------------------------------
void print(long arg)     /* Subroutine to be spawned */
{
   rt_printk("Hello, I am task %d\n", rt_whoami()); /* Print task Id */
}
-------------------------------------------------------------------------
```

Exercise 2c.

A long integer argument can be passed to a task during the call to *rt_task_init()*. Pass a unique argument to each of the ten tasks and have them print it. For example, pass an increment variable "i".

Exercise 2d.

Assign each task a unique priority such that the task which is started first gets the lowest priority and tasks started later get higher priority. Is there any difference in output? Has the order in which the ten tasks print changed? If not, explain why.

Semaphores

Introduction

Semaphores permit multitasking applications to coordinate their activities. The most obvious way for tasks to communicate is via various shared data structures. Because all tasks in RTAI exist in a single linear address space, sharing data structures between tasks is trivial. Global variables, linear buffers, ring buffers, link lists, and pointers can be referenced directly by code running in a different context. However, while shared address space simplifies the exchange of data, interlocking access to memory is crucial to avoid contention. Many methods exist for obtaining exclusive access to resources, and one of them is semaphores.

Objectives

The following are the primary objectives of this exercise:

- To demonstrate the use of RTAI semaphores.

Description

RTAI semaphores provide fast intertask communication in RTAI. Semaphores are the primary means for addressing the requirements of both mutual exclusion and task synchronization. In general we can say:

- Semaphores are data structures with a count and two associated operations, *give* and *take*.
- A '**give**' (signal) operation increments the count and returns immediately.
- A '**take**' operation decrements the count and returns immediately, unless the count is already zero. In this case the operation blocks until another process gives the semaphore.
- Semaphores have the semantics to determine which of the possibly many blocked processes attempting to take a semaphore will awaken first when it is given, for example first-in first-out (FIFO) or priority-based (PRIO).

There are three types of semaphores, optimized to address different classes of problems:

Binary

The fastest, most general purpose semaphore which is optimized for synchronization.
A binary semaphore is a semaphore with a maximum count of 1. Binary semaphores are useful to enforce mutual exclusion: only one process can have the semaphore at any point of time, and then other takers will block until the holder returns it. Shared data will remain consistent, since there is no possibility of being interrupted during access.

Resource

A special binary semaphore optimized for problems inherent in mutual exclusion: priority inheritance and recursion.
Resource semaphores are special binary semaphores suitable for managing resources. The task that acquires a

resource semaphore becomes its owner, also called resource owner, since it is the only one capable of manipulating the resource the semaphore is protecting. The owner has its priority increased to that of any task blocking on a wait to the semaphore. Such a feature, called *priority inheritance*, ensures that a high priority task is never slaved to a lower priority one, thus allowing to avoid any deadlock due to priority inversion.
Note that resource semaphores will enforce a priority queuing policy and can never use FIFO queuing policy.

Counting

A counting semaphore is similar to the binary semaphore, it but keeps track of the number of times the semaphore is given. Optimized for guarding multiple instances of a resource.

Semaphore API

In kernel mode you can use

- void rt_typed_sem_init (SEM *sem, int value, int type)

to initialize a specifically typed (counting, binary, resource) semaphore. Type is the semaphore type and queuing policy:

  - semaphore kind: CNT_SEM for counting semaphores, BIN_SEM for binary semaphores, RES_SEM for resource semaphores.
  - queuing policy: FIFO_Q, PRIO_Q for a fifo and priority queuing respectively.

- void rt_sem_init (SEM *sem, int value)

to initialize a counting semaphore.

In both user and kernel space:

- int rt_sem_delete(SEM *sem) /* Delete a semaphore. */
- int rt_sem_signal (SEM *sem) /* Signaling (giving) a semaphore. */
- int rt_sem_broadcast (SEM *sem) /* Signaling a semaphore, unblocks all tasks waiting on it. */
- int rt_sem_wait (SEM *sem) /* Take a semaphore. */
- int rt_sem_wait_if (SEM *sem) /* Take a semaphore, only if the calling task is not blocked. */
- int rt_sem_wait_until (SEM *sem, RTIME time) /* Wait a semaphore with timeout. */
- int rt_sem_wait_timed (SEM *sem, RTIME delay) /* Wait a semaphore with timeout.*/

Example: Binary Semaphore

A binary semaphore can be declared as follows:

```
static SEM semBinary;
rt_typed_sem_init(&semBinary, 1, BIN_SEM | FIFO_Q );
```

It is taken by the following statement:

```
rt_sem_wait(&semBinary);
```

A binary semaphore can be viewed as a flag that is available or unavailable. When a task takes a binary semaphore, using *rt_sem_wait()*, the outcome depends on whether the semaphore is available or unavailable at

the time of the call. If the semaphore is available, then the semaphore becomes unavailable and then the task continues executing immediately. If the semaphore is unavailable, the task is put in a queue of blocked tasks and enters a state of pending on the availability of the semaphore. When a task gives a binary semaphore, using *rt_sem_signal()*, the outcome depends on whether the semaphore is available or unavailable at the time of the call. If the semaphore is already available, giving the semaphore has no effect at all. If the semaphore is unavailable and no task is waiting to take it, then the semaphore becomes available. If the semaphore is unavailable and one or more tasks are pending on its availability, then the first task in the queue of pending tasks is unblocked, and the semaphore is left unavailable.

Example: Changing a global variable by two tasks

In the example below, two tasks (taskOne and taskTwo), are competing to update the value of a global variable, called "global." The objective of the program is to toggle the value of the global variable (1s and 0s). taskOne increments the value of "global" and taskTwo decrements the value.

```
--------------------------------------------------------------------------------
/*
   file global.c
   update global variable by two tasks
*/
/* includes */
#include <linux/kernel.h>       /* decls needed for kernel modules */
#include <linux/module.h>       /* decls needed for kernel modules */
#include <linux/version.h>      /* LINUX_VERSION_CODE, KERNEL_VERSION() */
#include <linux/errno.h>        /* EINVAL, ENOMEM */
/*
  Specific header files for RTAI, our flavor of RT Linux
 */
#include "rtai.h"               /* RTAI configuration switches */
#include "rtai_sched.h" /* RTAI scheduling */
#include <rtai_sem.h>   /* RTAI semaphores */
MODULE_LICENSE("GPL");
/* globals */
#define ITER 10
static RT_TASK  t1;
static RT_TASK  t2;
/* function prototypes */
void taskOne(long arg);
void taskTwo(long arg);
int global = 0;
void tasks(void)
{
   int retval;
   /* init the two tasks */
   retval = rt_task_init(&t1,taskOne, 0, 1024, 0, 0, 0);
   retval = rt_task_init(&t2,taskTwo, 0, 1024, 0, 0, 0);
   /* start the two tasks */
   retval = rt_task_resume(&t1);
   retval = rt_task_resume(&t2);
}
void taskOne(long arg)
{
   int i;
   for (i=0; i < ITER; i++)
   {
```

```
        rt_printk("I am taskOne and global = %d.....\n", ++global);
    }
}
void taskTwo(long arg)
{
    int i;
    for (i=0; i < ITER; i++)
    {
        rt_printk("I am taskTwo and global = %d----------\n", --global);
    }
}
int init_module(void)
{
    printk("start of init_module\n");
    rt_set_oneshot_mode();
    start_rt_timer(1);
    tasks();
    printk("end of init_module\n");
    return 0;
}
void cleanup_module(void)
{
    // task end themselves -> not necessary to delete them
    return;
}
```

Exercise

Exercise 3a.

Run the example program above, observe what happens and explain the resulting output.
Also note that no toggling between 0 and 1 of the global value happens. Why is this?

Exercise 3b.

We can conclude that the example program does not toggle the value of the global variable between 1 and 0. Change the program by using a binary semaphore so that it does reach this objective. Describe why your solution works.

Exercise 3c.

Use semaphores to adapt the program from exercise 2d such that, with the same priority assignment, the tasks are executed in the order of their priority (instead of the order in which they are started). Try a solution which uses *rt_sem_signal*, and another with *rt_sem_broadcast*.

Periodic Scheduling

Introduction

In exercise 2 we already discussed periodic scheduling and looked at an example of one-shot scheduling. In this exercise we consider periodic scheduling.

Objectives

The following are the primary objectives of this exercise:

- To demonstrate the use of periodic scheduling.
- Get some experience with the interface to PC ports.

Description

In Exercise 2 the main constructs for working with periodic tasks have been explained. A few additional remarks:

- Any number of tasks may be scheduled, up to memory limits (there is no predefined maximum)
- Tasks may share variables, since they share the same address space. In general, some care must be taken to ensure mutual exclusion, to avoid that one task's actions on shared data are interrupted by another's.
    - Small data types (char, int) are changed atomically, i.e., reads and writes cannot be interrupted.
    - For large types (structures, arrays), semaphores or other mechanisms are needed to avoid simultaneous access to shared data.
- With two or more periodic tasks, you have to determine the base period, multiples of which will be allowable for each task's period.
    - E.g., if we have a 200 microsecond task and a 300 microsecond task, the base period can be 100 microseconds.
    - In general, you might use the greatest common divisor of all your desired task periods.
    - Practically speaking, the lower bound on the base period is about 10 microseconds.
- The tasks are scheduled according to their priority. If a task is ready to run (its period has expired), it can only interrupt tasks of lower priority. A higher priority task will keep executing to completion.

Example code

In this example we will do the following:

- Set up the timer with a base period, in our case 1 millisecond (100 000 nanoseconds).
- Define the task function, in our case toggling the PC speaker port by alternately setting or clearing bit 2 of the speaker port address, 61 hex (0x61 in C). A web search on "PC port addresses" is a good source for programming PC resources (parallel port, serial port, joystick, LEDs, etc.).
- Set up the task structure and fill in fields for the task code, its priority, size of stack and other

information.

- Schedule the task to run with the specified base period.

```
/*
  periodic_task.c

  Sets up a task in pure periodic mode that toggles the speaker. In
  pure periodic mode, the timer is programmed once to expire at a
  given period, and all tasks must run at multiples of this period.
  This is useful when such a fundamental period can be identified,
  in which case the overhead of timer reprogramming is avoided.

*/


#include <linux/kernel.h> /* decls needed for kernel modules */
#include <linux/module.h> /* decls needed for kernel modules */
#include <linux/version.h>        /* LINUX_VERSION_CODE, KERNEL_VERSION() */

/*
  Specific header files for RTAI, our flavor of RT Linux
 */
#include "rtai.h"               /* RTAI configuration switches */
#include "rtai_sched.h"         /* rt_set_periodic_mode(), start_rt_timer(),
                                   nano2count(), RT_LOWEST_PRIORITY,
                                   rt_task_init(), rt_task_make_periodic() */

/*
  Some newer versions define RT_SCHED_LOWEST_PRIORITY instead, so we'll
  get that if necessary
 */
#if ! defined(RT_LOWEST_PRIORITY)
#if defined(RT_SCHED_LOWEST_PRIORITY)
#define RT_LOWEST_PRIORITY RT_SCHED_LOWEST_PRIORITY
#else
#error RT_SCHED_LOWEST_PRIORITY not defined
#endif
#endif

/*
  Some RTAI functions return standard Linux symbolic error codes, so
  we'll include them
 */
#include <linux/errno.h>  /* EINVAL, ENOMEM */

/*
  Include declarations for inb() and outb(), the byte input and output
  functions for port I/O
 */
#include <asm/io.h>             /* may be <sys/io.h> on some systems */

/*
  Linux kernel modules in kernel versions 2.4 and later are asked to
  state their license terms.  "GPL" is the usual, for software
  released under the Gnu General Public License. The Linux kernel
  module loader 'insmod' will complain if it's anything else.
 */

/*
```

```
  THIS SOFTWARE WAS PRODUCED BY EMPLOYEES OF THE U.S. GOVERNMENT AS PART
  OF THEIR OFFICIAL DUTIES AND IS IN THE PUBLIC DOMAIN.

  When linked into the Linux kernel the resulting work is GPL. You
  are free to use this work under other licenses if you wish.
*/
#if LINUX_VERSION_CODE > KERNEL_VERSION(2,4,0)
MODULE_LICENSE("GPL");
#endif

static RT_TASK sound_task;      /* we'll fill this in with our task */
static RTIME sound_period_ns = 1000000; /* timer period, in nanoseconds */

#define SOUND_PORT 0x61         /* address of speaker */
#define SOUND_MASK 0x02         /* bit to set/clear */

/*
  sound_function() is our task code, executed each period. RTAI requires
  all tasks to be of this form, taking an int argument and returning nothing.
  We'll ignore our argument.

  This simple task just toggles the speaker port.
 */
void sound_function(int arg)
{
  unsigned char sound_byte;
  unsigned char toggle = 0;

  while (1) {
    /*
      Toggle the sound port
     */
    sound_byte = inb(SOUND_PORT);
    if (toggle) {
      sound_byte = sound_byte | SOUND_MASK;
    } else {
      sound_byte = sound_byte & ~SOUND_MASK;
    }
    outb(sound_byte, SOUND_PORT);
    toggle = ! toggle;

    /*
      Wait one period by calling

      void rt_task_wait_period(void);

      which applies to the currently executing task, and used the period
      set up in its task structure.
    */
    rt_task_wait_period();
  }

  /* we'll never get here */
  return;
}

/*
   All Linux kernel modules must have 'init_module' as the entry point,
```

```
  similar to the C language 'main' for programs. init_module must return
  an integer value, 0 signifying success and non-zero signifying failure.

  The Linux kernel module installer program 'insmod' will look at this
  and print out an error message at the console.
*/
int init_module(void)
{
 RTIME sound_period_count;     /* requested timer period, in counts */
 RTIME timer_period_count;     /* actual timer period, in counts */
 int retval;                   /* we look at our return values */

 /*
  Set up the timer to expire in pure periodic mode by calling

  void rt_set_periodic_mode(void);

  This sets the periodic mode for the timer. It consists of a fixed
  frequency timing of the tasks in multiple of the period set with a
  call to start_rt_timer. The resolution is that of the 8254
  frequency (1193180 hz). Any timing request not an integer multiple
  of the period is satisfied at the closest period tick. It is the
  default mode when no call is made to set the oneshot mode.
 */
 rt_set_periodic_mode();

 /*
  Start the periodic timer by calling

  RTIME start_rt_timer(RTIME period);

  This starts the timer with the period 'period' in internal count units.
  It's usually convenient to provide periods in second-like units, so
  we use the nano2count() conversion to convert our period, in nanoseconds,
  to counts. The return value is the actual period set up, which may
  differ from the requested period due to roundoff to the allowable
  chip frequencies.

  Look at the console, or /var/log/messages, to see the printk()
  messages.
 */
 sound_period_count = nano2count(sound_period_ns);
 timer_period_count = start_rt_timer(sound_period_count);
 printk("periodic_task: requested %d counts, got %d counts\n",
        (int) sound_period_count, (int) timer_period_count);

 /*
  Initialize the task structure by calling

  rt_task_init(RT_TASK *task, void *rt_thread, int data, int stack_size,
  int priority, int uses_fpu, void *signal);

  This structure will be passed to rt_task_init() later to start the
  task.

  'task' is a pointer to an RT_TASK type structure whose space must
  be provided by the application. It must be kept during the whole
  lifetime of the real time task and cannot be an automatic
```

```
  variable.  'rt_thread' is the entry point of the task
  function. The parent task can pass a single integer value data to
  the new task.  'stack_size' is the size of the stack to be used by
  the new task. See the note below on computing stack size.
  'priority' is the priority to be given the task. The highest
  priority is 0, while the lowest is RT_LOWEST_PRIORITY (0x3fffFfff,
  or 1073741823).  'uses_fpu' is a flag. Nonzero value indicates
  that the task will use the floating point unit.  'signal' is a
  function that is called, within the task environment and with
  interrupts disabled, when the task becomes the current running
  task after a context switch.

  The newly created real time task is initially in a suspend
  state. It is can be made active either with
  rt_task_make_periodic(), rt_task_make_periodic_relative_ns() or
  rt_task_resume().
*/
retval =
  rt_task_init(&sound_task,       /* pointer to our task structure */
               sound_function, /* the actual timer function */
               0,              /* initial task parameter; we ignore */
               1024,           /* 1-K stack is enough for us */
               RT_LOWEST_PRIORITY, /* lowest is fine for our 1 task */
               0,              /* uses floating point; we don't */
               0);             /* signal handler; we don't use signals */
if (0 != retval) {
  if (-EINVAL == retval) {
    /* task structure is already in use */
    printk("periodic task: task structure already in use\n");
  } else if (-ENOMEM == retval) {
    /* stack could not be allocated */
    printk("periodic task: can't allocate stack\n");
  } else {
    /* unknown error */
    printk("periodic task: error initializing task structure\n");
  }
  return retval;
}

/*
  Start the task by calling

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

  This marks the task 'task', previously created with
  rt_task_init(), as suitable for a periodic execution, with period
  'period'.  The time of first execution is given by start_time.
  start_time is an absolute value measured in clock
  ticks. After the first task invocation, it should call
  rt_task_wait_period() to reschedule itself.
 */
retval =
  rt_task_make_periodic(&sound_task, /* pointer to our task structure */
                        /* start one cycle from now */
                        rt_get_time() + sound_period_count,
                        sound_period_count); /* recurring period */
if (0 != retval) {
  if (-EINVAL == retval) {
```

```
      /* task structure is already in use */
      printk("periodic task: task structure is invalid\n");
    } else {
      /* unknown error */
      printk("periodic task: error starting task\n");
    }
    return retval;
  }

  return 0;
}

/*
  Every Linux kernel module must define 'cleanup_module', which takes
  no argument and returns nothing.

  The Linux kernel module installer program 'rmmod' will execute this
  function.
 */
void cleanup_module(void)
{
  int retval;

  retval = rt_task_delete(&sound_task);

  if (0 !=  retval) {
    if (-EINVAL == retval) {
      /* invalid task structure */
      printk("periodic task: task structure is invalid\n");
    } else {
      printk("periodic task: error stopping task\n");
    }
  }

  /* turn off sound in case it was left on */
  outb(inb(SOUND_PORT) & ~SOUND_MASK, SOUND_PORT);

  return;
}
```

Exercise

Exercise 4.

Run the example above.

This program runs only one periodic task that toggles the speaker port. Write another periodic task that varies the frequency of toggling, according to the following specification:

- Let task one run at a fixed 100 microsecond period, and let it toggle the speaker port at intervals determined by a delay count, that is as long the delay count is not passed, we skip the toggle. Thus the larger the delay count, the lower the audible frequency.
- A second task runs at a fixed 1 second period, and changes the delay count each time, so you will hear the frequency change each second.
- The delay count is a shared variable, an 'int', which is only changed by the second task, and only read by

the first task. So we don't have to protect it with a semaphore/critical section because it cannot get corrupted by interrupted reads/writes because only the second task changes it.

- Since the task timing itself doesn't need to vary, the tasks both run in pure periodic mode, using the shortest period of the two tasks as the base.
- The frequency of the sound will continue to drop indefinitely as the delay count increases.

FIFOs & Messages & Mailboxes

Introduction

In RTAI, the primary intertask communication mechanisms between tasks are FIFOs, mailboxes and messages. They all have the same API for both user space as kernel space. This fulfills RTAI's promise for symmetric APIs for user and kernel space (however only for FIFOs the symmetry is not totally perfect).

Objectives

The following are the primary objectives of this exercise:

- To demonstrate the use of FIFOs, messages and message queues in RTAI.

FIFOs

FIFOs are simple queues of raw bytes, that is, a FIFO can only access data in a specified linear sequence. Typically, fixed-sized data structures are written to a FIFO, so that the reader does not need to worry about message boundaries. In some cases variable-sized messages are written, in which case the reader must search for the message boundary, often a special character.

Creating a FIFO

- On the real-time side, a FIFO is created by calling

  ```
  int rtf_create (unsigned int fifo, int size);
  ```

  'rtf_create()' creates FIFO number 'fifo', of initial size 'size' bytes. 'fifo' is an integer ranging from 0 to RTF_NO (currently 63), that identifies the fifo on further operations. 'fifo' may refer an existing FIFO and then the size is adjusted (if necessary).
- On the Linux side, FIFO numbers 0 to 63 are associated with character devices /dev/rtf0 through /dev/rtf63. These device files are created when you install RTAI, so they are always visible. To "create" a FIFO on the Linux side, use the standard Unix function 'open()', e.g.,

  ```
  file_descriptor = open("/dev/rtf0", O_RDONLY);
  ```

  The integer 'file_descriptor' is returned to you, and used to identify the FIFO during subsequent read or write operations. 'O_RDONLY' is a constant signifying that you'll only read the FIFO. Other possibilities are O_WRONLY and O_RDWR, for write-only and read-write, respectively. These are from standard Unix.

Accessing a FIFO

- On the real-time side, reading and writing are done using the 'rtf_get()' and 'rtf_put()' functions, e.g.,

  ```
  num_read = rtf_get(0, &buffer_in, sizeof(buffer_in));
  ```

```
num_written = rtf_put(1, &buffer_out, sizeof(buffer_out));
```

In the 'rtf_get' function the last parameter is the size of the input buffer, which also specifies the maximum number of bytes to be read. You can check the return values for the actual number of bytes read or written, where negative numbers signify an error.

- On the Linux side, reading and writing are done using the standard Unix 'read()' and 'write()' functions, e.g.,

```
num_read = read(read_descriptor, &buffer_in, sizeof(buffer_in));
num_written = write(write_descriptor, &buffer_out, sizeof(buffer_out));
```

where 'read_descriptor' and 'write_descriptor' are the file descriptors returned to you on the previous call to 'open()'. Negative return values signify an error.

Blocking versus Non-Blocking Reads

- In Unix, devices can be opened for either blocking- or non-blocking reads.
  - With a blocking read, the 'read()' function does not return if there is no data to be read. The calling process goes to sleep ("blocks") until the 'read()' function returns later when some data is available. This is the default behavior.
  - In some situations this blocking behavior is not desirable, and the device can be configured so that reads are non-blocking. In this case, 'read()' returns 0 immediately if no characters are available.
  - In a real-time application, blocking is typically not desired. 'rtf_get()' returns immediately with a 0 return value if no characters are available.

Messages

The basic message passing mechanism allows sending a four byte message from a sender to a receiver. Both on the real-time side and the Linux side, the following functions can be used.

Sending and receiving a message

- Send a message to a task synchronously. Blocks until the receiver is ready to get the message.

```
RT_TASK * rt_send (RT_TASK *task, unsigned int msg)
```

- Receive a message from a task (if the task parameter is 0 it accepts from any task). Blocks until a message is available.

```
RT_TASK * rt_receive (RT_TASK *task, unsigned int *msg)
```

The functionality above can also be used with a timeout, by calling *rt_send_timed* and *rt_receive_timed.* When using extension "_if", the message is only passed if the whole message can be passed immediately, i.e., the caller of the function will not be blocked.
More recent versions of RTAI allow so-called extended messages. They are less efficient than the four byte messages, but are more flexible in that they allow messages of arbitrary size. The main primitives are:

- Send a message buffer "msg" with size "size" to a task.

```
RT_TASK * rt_sendx (RT_TASK *task, void *msg, int size)
```

- Receive a message from a task; "msg" points to the message to be received, "size" its size, and "len" a pointer to the actual length of the received message.

RT_TASK * rt_receivex (RT_TASK *task, void *msg, int size, int *len)

Mailboxes

A mailbox is a buffer managed by the operating system. Mailboxes allow a variable number of messages, each of variable length, to be queued. Any task can send a message to a mailbox and any task can receive a message from a mailbox. Multiple tasks can send to and receive from the same mailbox. A receiving task reads the messages in the order of arrival. (Of course, variations on this policy exist.) Two-way communication between two tasks generally requires two mailboxes, one for each direction.

The original implementation uses a FIFO (First In, First Out) policy; a recent addition are typed mailboxes, that have a priority message delivery option. Sending and receiving messages with "typed" mailboxes can be done with several policies:

- Unconditionally: the task blocks until the whole message has passed. (default)
- Best-effort: only pass the bytes that can be passed immediately (using command extension '_wp').
- Conditional on availability: only pass a message if the whole message can be passed immediately (by using command extension '_if').
- Timed: with absolute or relative time-outs (using extensions '_up' and'_timed').

Creating a mailbox (in kernel space)

- Initialize a mailbox

  int rt_mbx_init (MBX *mbx, int size)

- Initialize a fully typed mailbox queuing tasks according to the specified type.

  int rt_typed_mbx_init(MBX *mbx, int size, int type)

Sending and receiving from mailbox

- Send a message unconditionally.

  int rt_mbx_send (MBX *mbx, void *msg, int msg_size)

- Receive a message unconditionally (similar to FIFOs, the 'size' parameter specifies the maximum number of bytes to be read from the mailbox).

  int rt_mbx_receive(MBX *mbx, void *msg, int msg_size)

  Important: the msg_size should exactly match the size of the message send!

These two functions can be applied in the different policies mentioned above using the mentioned extension on the command (_wp,_if,_up or _timed).

Example: Mailbox

In the example program below, taskOne sends a string message ("Received message from taskOne") to taskTwo using a (FIFO) mailbox. Next taskTwo prints the message it received from taskOne.

/*

```
  Example Mailbox
*/

/* includes */
#include <linux/kernel.h>       /* decls needed for kernel modules */
#include <linux/module.h>       /* decls needed for kernel modules */
#include <linux/version.h>      /* LINUX_VERSION_CODE, KERNEL_VERSION() */
#include <linux/errno.h>        /* EINVAL, ENOMEM */

/*
  Specific header files for RTAI, our flavor of RT Linux
 */

#include <rtai.h>
#include <rtai_sched.h>
#include <rtai_mbx.h>

MODULE_LICENSE("GPL");

static RT_TASK  t1;
static RT_TASK  t2;
/* function prototypes */
void taskOne(long arg);
void taskTwo(long arg);


/* defines */
#define MAX_MESSAGES 100
#define MAX_MESSAGE_LENGTH 50

/* globals */
static MBX mailboxId;

void message(void) /* function to create the message queue and two tasks */
{
   int retval;

   /* create FIFO mailbox */
   retval = rt_typed_mbx_init (&mailboxId, MAX_MESSAGES,  FIFO_Q);
   if (0 != retval) {
      if (-ENOMEM == retval) {
        printk("ENOMEM: Space could not be allocated for the mailbox buffer.");
      } else {
        /* unknown error */
        printk("Unknown error creating message queue\n");
      }
   }

   /* spawn the two tasks that will use the mailbox */

   /* init the two tasks */
   retval = rt_task_init(&t1,taskOne, 0, 1024, 0, 0, 0);
   retval = rt_task_init(&t2,taskTwo, 0, 1024, 0, 0, 0);

   /* start the two tasks */
   retval = rt_task_resume(&t1);
   retval = rt_task_resume(&t2);
```

```
}

void taskOne(long arg) /* task that writes to the mailbox */
{
   int retval;
   char message[] = "Received message from taskOne";

   /* send message */
   retval = rt_mbx_send(&mailboxId, message, sizeof(message));
   if (0 != retval) {
      if (-EINVAL == retval) {
        rt_printk("mailbox is invalid\n");
      } else {
        /* unknown error */
        rt_printk("Unknown mailbox error\n");
      }
   } else
   {
      rt_printk("taskOne sent message to mailbox\n");}

}

void taskTwo(long arg) /* tasks that reads from the mailbox */
{
   int retval;
   char msgBuf[MAX_MESSAGE_LENGTH];

   /* receive message */
   retval = rt_mbx_receive_wp(&mailboxId, msgBuf, 50);
   if (-EINVAL == retval) {
        rt_printk("mailbox is invalid\n");
   } else {
      rt_printk("taskTwo received message : %s\n",msgBuf);
      /* retval gives number of not received bytes from 50 */
      rt_printk("with length %d\n",50-retval);
   }


   /* delete mailbox */
   rt_mbx_delete(&mailboxId);
}

int init_module(void)
{

   printk("start of init_module\n");

   rt_set_oneshot_mode();
   start_rt_timer(1);

   message();

   printk("end of init_module\n");
   return 0;
}

void cleanup_module(void)
{
```

```
    stop_rt_timer();
    rt_task_delete(&t1);
    rt_task_delete(&t2);

   return;
}
```

Exercises

Exercise 5a.

1. Modify the example program such that also taskTwo first sends a string message ("Received message from taskTwo") to taskOne (before receiving the message from taskOne) via a mailbox, and have taskOne receive it (after it has sent a message) and print the message received from taskTwo. Try the use of one mailbox.
2. Use two mailboxes and describe the difference with the program that uses one mailbox.
3. Send several messages from one task to another (and back). Is it possible to send all messages before receiving any messages?

Exercise 5b.

1. Modify the example program such that it uses integer messages instead of mailboxes. Let taskOne send 112 to taskTwo which returns 221.
2. Send several messages from one task to another (and back). Is it possible to send all messages before receiving any messages?

Exercise 5c.

1. Modify the example program such that it uses FIFOs instead of mailboxes and sends string messages "Received message from taskOne" to taskTwo and "Received message from taskTwo" to taskOne.
2. Send several messages from one task to another (and back). Is it possible to send all messages before receiving any messages?

Exercise 5d.

Compare the three mechanisms (FIFO, Messages, and Mailbox), what are the differences, when to use which mechanism?

Round-Robin Task Scheduling

Introduction

Task scheduling is the assignment of starting and ending times to a set of tasks, subject to certain constraints. Constraints are typically either time constraints or resource constraints. A time-sharing operating system runs each active process in turn for its share of time (its "timeslice"), thus creating the illusion that multiple processes are running simultaneously on a single processor.

Beginning from release 24.1.6, RTAI schedulers offers the possibility of choosing between First In First Out (FIFO) priority-based preemptive scheduling and time slicing Round Robin (RR) scheduling policies, on a per task basis, the default being FIFO. The implementation of RR scheduling can be taken away by simply commenting out the macro ALLOW_RR in RTAI schedulers, to avoid a (very) small scheduling overhead if RR is not used (3 do nothing "if"s at most).

Objectives

The following are the primary objectives of this exercise:

- To demonstrate the use of round-robin task scheduling facilities.

Description

Round-Robin Scheduling

A round-robin scheduling algorithm attempts to share the CPU fairly among all ready tasks of the same priority. Without round-robin scheduling, when multiple tasks of equal priority must share the processor, a single task can usurp the processor by never blocking, thus never giving other equal priority tasks a chance to run.

Round-robin scheduling achieves fair allocation of the CPU to tasks of the same priority by an approach known as *time slicing*. Each task executes for a defined interval or *time slice*; then another task executes for an equal interval, in rotation. The allocation is fair in that no task of a priority group gets a second slice of time before the other tasks of a group are given a slice.

Round-robin scheduling can be enabled with a call to the following function, defined in rtai_sched.h,

```
void rt_set_sched_policy(struct rt_task_struct *task,
                         int policy,
                         int rr_quantum_ns);
```

This sets the time slice interval value to *rr_quantum_ns*, which is the amount of time each task is allowed to run before relinquishing the processor to another equal priority task. Scheduling policy RT_SCHED_RR yields round-robin scheduling. The other, default, policy is RT_SCHED_FIFO (and then *rr_quantum_ns is not used)*. The scheduling policy (and time slice) can be assigned to each task separately.

Example: Round-robin Scheduling

In the example below, three tasks with the same priority print their task ids and task names on the console.

- The tasks are started from the init_module() function which runs in the linux scheduler. However the tasks started run in the real-time scheduler and will therefore be immediately be running in real-time whereas the init_module() function has to wait for completion. Hence, we have added a semaphore on which a started task will block and the init_module() function can continue spawning new tasks. Finally, it will signal all the task blocking on the semaphore that they can continue.
- Each task terminates after EXECTIME time units. This is programmed as follows:

```
starttime = rt_get_cpu_time_ns();
while(rt_get_cpu_time_ns() < (starttime + EXECTIME));
```

- The signal handler *signal*, a parameter of *rt_task_init*, counts the number of context switches of each task, using global array *switchesCount*. In general, the signal handler is a function called within the task environment and with interrupts disabled, when the task becomes the current running task after a context switch. Note however that signal is not called at the very first scheduling of the task.

```
--------------------------------------------------------------------------------
#include <linux/kernel.h>
/*
  Example, exercise 6 (to add round robin)
  Three tasks with equal priority
*/
#include <linux/kernel.h>
#include <linux/module.h>
#include <rtai.h>
#include <rtai_sched.h>
#include <rtai_sem.h>
MODULE_LICENSE("GPL");
#define STACK_SIZE 2000
#define EXECTIME   400000000
/* proposed time-slice, not yet used here */
#define RR_QUANTUM  10000000
#define NTASKS 3
#define PRIORITY 100
/* globals */
static SEM sync;
static RT_TASK tasks[NTASKS];
static int switchesCount[NTASKS];
static void fun(long indx)
/* function executed by all tasks */
{
      RTIME starttime;
   RTIME endtime;
      rt_printk("Resume task #%d (%p) on CPU %d.\n", indx, &tasks[indx], hard_cpu_id());
      rt_sem_wait(&sync);
      rt_printk("Task #%d (%p) starts executing on CPU %d.\n", indx, &tasks[indx], hard_cpu_id());
   /* execute until EXECTIME time units have elapsed */
      starttime = rt_get_cpu_time_ns();
      while(rt_get_cpu_time_ns() < (starttime + EXECTIME));
   /* ready, check time and top signalling context switches */
   endtime = rt_get_cpu_time_ns()-starttime;
      tasks[indx].signal = 0;
      rt_printk("Task #%d (%p) terminates after %d.\n", indx, &tasks[indx],endtime);
}
static void signal(void)
```

```
/* signal handler, executed when a context switch occurs */
{
	RT_TASK *task;
	int i;
	for (i = 0; i < NTASKS; i++) {
		if ((task = rt_whoami()) == &tasks[i]) {
			switchesCount[i]++;
			rt_printk("Switch to task #%d (%p) on CPU %d.\n", i, task, hard_cpu_id());
			break;
		}
	}
}
int init_module(void)
{
	int i;
	printk("INSMOD on CPU %d.\n", hard_cpu_id());
	rt_sem_init(&sync, 0);
	rt_set_oneshot_mode();
	start_rt_timer(1);
	for (i = 0; i < NTASKS; i++) {
		rt_task_init(&tasks[i], fun, i, STACK_SIZE, PRIORITY, 0, signal);
	}
	for (i = 0; i < NTASKS; i++) {
		rt_task_resume(&tasks[i]);
	}
	rt_sem_broadcast(&sync);
	return 0;
}
void cleanup_module(void)
{
	int i;
	stop_rt_timer();
	rt_sem_delete(&sync);
	for (i = 0; i < NTASKS; i++) {
		printk("number of context switches task # %d -> %d\n", i, switchesCount[i]);
		rt_task_delete(&tasks[i]);
	}
}
--------------------------------------------------------------------------------
```

Exercises

Exercise 6a.

Run the program above and observe its behaviour. Next schedule all three tasks according to the round robin policy and explain the output.

Exercise 6b.

Add a fourth task which is the same as the other three tasks, but has a priority of 80. Describe and explain the output from running the program. Does it make a difference whether task four uses the round-robin policy or not? Explain why.

Preemptive Priority Based Task Scheduling

Introduction

The default scheduling policy in RTAI is preemptive and based on priorities. Hence, execution of a low-priority task is interrupted when a high-priority is ready to execute.

Objectives

The following are the primary objectives of this exercise:

- To demonstrate the use of RTAI preemptive priority based task scheduling facilities.

Description

Preemptive Priority Based Scheduling

With a preemptive priority based scheduler, each task has a priority and the kernel insures that the CPU is allocated to the highest priority task that is ready to run. This scheduling method is *preemptive* in that if a task that has a higher priority than the current task becomes ready to run, the kernel immediately saves the current tasks's context and switches to the context of the higher priority task. The *RTAI* kernel has priority levels between the highest priority RT_SCHED_HIGHEST_PRIORITY (which equals 0) and the lowest priority RT_SCHED_LOWEST_PRIORITY (which equals 1,073,741,823) (both are defined in rtai_sched.h).

When creating a task with *rt_task_init(..)* one of the arguments is the priority at which the task is to execute. By varying the priority of the task spawned, you can affect the priority of the task. The task priority number itself has no particular significance by itself only the value relative to other tasks is important. In addition a task's priority can be changed after its spawned using the following routine :

int rt_change_prio(RT_TASK **task*, int*priority*);

RTAI uses preemptive priority based scheduling as its default scheduling policy. However it can be explicitly enabled (e.g. when switch back from round robin scheduling) with a call to the following function, defined in rtai_sched.h,

```
void rt_set_sched_policy(struct rt_task_struct *task,

                         int policy,

                         int rr_quantum_ns);
```

The scheduling policy (and time slice) can be assigned to each task separately. Scheduling policy RT_SCHED_FIFO yields preemptive priority based scheduling. The other policy RT_SCHED_RR yields round-robin scheduling which is discussed in the previous exercise. In the case of preemptive priority based scheduling the time slice interval value rr_quantum_ns has no meaning and can just be set to 0.

Exercises

Exercise 7a.

Rewrite the program of Exercise 6 "Round-Robin Task Scheduling"
so that it uses pure priority based scheduling without any time-slicing. Use the following priorities :

```
#define HIGH 100 /* high priority */
#define MID 101 /* medium priority */
#define LOW 102 /* low priority */
```

where the first task gets priority LOW, the second one priority MEDIUM and the third one priority HIGH.

Run the program and describe its output.

Experiment 7b.

Modify the program of 7a. such that first task still has the lowest priority, but the other two tasks are running at the same high priority. Compare the output with that of Experiment 7a.

Experiment 7c.

Modify the program of 7a. such that the highest-priority task half-way its execution:

- first raises the priority of the middle-priority task with 10 and
- next immediately also raises the priority of the low-priority task with 10.

Explain what happens.
Note 1: Raising priority means making it higher, but the number describing the priority should be lowered, because 0 is the highest priority in RTAI.
Note 2: RTAI version 3.3. contains a buggy version of the 'rtai_change_prio' function. For instance, lowering the priority of a running task, has no effect. However, it is possible to change the priority of a not running task (as proposed in exercise 7c) which will have the desired effect.

Priority Inversion

Introduction

Priority inversion occurs when a higher-priority task is forced to wait an indefinite period for the completion of a lower priority task. For example, suppose **prioHigh, prioMedium,** and **prioLow** are tasks of high, medium, and low priority, respectively. **prioLow** has acquired a resource by taking its associated binary semaphore. When **prioHigh** preempts **prioLow** and contends for the resource by taking the same semaphore, it becomes blocked. If **prioHigh** would be blocked no longer than the time it normally takes **prioLow** to finish with the resource, there would be no problem, because the resource cannot be preempted. However, the low priority task is vulnerable to preemption by the medium priority task, **prioMedium**, which could prevent **prioLow** from relinquishing the resource. This condition could persist, blocking **prioHigh** for an extensive period of time.

To address the problem of priority inversion, RTAI provides an special semaphore called a **RESOURCE semaphore**. When this semaphore is used, a priority inheritance algorithm is used. This algorithm insures that the task that owns a resource (i.e., takes the RESOURCE semaphore) executes at the priority of the highest priority task blocked on that resource. When execution is complete, the task relinquishes the resource and returns to its normal priority. Therefore, the inheriting task is protected from preemption by an intermediate priority task.

Objectives

The following are the primary objectives of this exercise:

- To demonstrate RTAI's priority inversion avoidance mechanisms.

Exercises

Exercise 8a.

Program an example program in which priority inversion takes place. Use in the program three tasks for which the following behaviour happens :

1. **prioLow** task locks the semaphore.
2. **prioLow** task gets preempted by **prioMedium** task which runs for a long time which results in the blocking of **prioLow**.
3. **prioHigh** task preempts **prioMedium** task and tries to lock the semaphore which is currently locked by **prioLow**.
4. Since both **prioLow** and **prioHigh** are both blocked, **prioMedium** runs to completion(a very long time).
5. By the time **prioHigh** runs it is likely that it has missed its timing requirements.

The program's printout should look like :

Low priority task locks semaphore
Medium task running
High priority task trys to lock semaphore
Medium task running
Medium task running
Medium task running

Medium task running
Medium task running
Medium task running
Medium task running
Medium task running
Medium task running
----------Medium priority task exited
Low priority task unlocks semaphore
High priority task locks semaphore
High priority task unlocks semaphore
High priority task trys to lock semaphore
High priority task locks semaphore
High priority task unlocks semaphore
High priority task trys to lock semaphore
High priority task locks semaphore
High priority task unlocks semaphore
.....High priority task exited
Low priority task locks semaphore
Low priority task unlocks semaphore
Low priority task locks semaphore
Low priority task unlocks semaphore
.....Low priority task exited

Exercise 8b.

Modify the program of 8a such that the problem with priority inversion is eliminated and the printout from the program looks like the following:

Low priority task locks semaphore
Medium task running
High priority task trys to lock semaphore
Low priority task unlocks semaphore
High priority task locks semaphore
High priority task unlocks semaphore
High priority task trys to lock semaphore
High priority task locks semaphore
High priority task unlocks semaphore
High priority task trys to lock semaphore
High priority task locks semaphore
High priority task unlocks semaphore
.....High priority task exited
Medium task running
Medium task running
Medium task running
Medium task running
Medium task running
Medium task running
Medium task running
Medium task running
Medium task running
----------Medium priority task exited
Low priority task locks semaphore
Low priority task unlocks semaphore
Low priority task locks semaphore
Low priority task unlocks semaphore
.....Low priority task exited

RMS & EDF

Note: Excercise a,b,c can be done in a simulated realtime linux os in vmware, but the timing values will not be correct. Exercises d,e,f and g must be done on a real hardware platform.

Introduction

Besides preemptive priority-based FIFO and round robin scheduling, RTAI also supports Rate Monotonic Scheduling (RMS) and Earliest Deadline First (EDF) scheduling.

Objectives

The following are the primary objectives of this exercise:

- To get experience with the RMS and EDF possibilities in RTAI.

Description

Rate Monotonic Scheduling (RMS)

Rate Monotonic Scheduling (RMS) is a simple scheduling policy for real time periodic tasks that assigns task priorities in the order of the highest task frequencies, i.e. the shortest periodic task gets the highest priority, then the next with the shortest period get the second highest priority, and so on. So RMS could be easily implemented by yourself simply by assigning appropriate priorities and using the standard prioritized preemptive First In First Out (FIFO) schedulers found in RTAI.

However there are situations, such as dynamic task creation of periodic tasks, that can make it annoying to keep track of all periods and corresponding priorities. Thus some form of minimal support could be of help. RTAI allows you to be sure to comply with MS by calling the function

```
void rt_spv_RMS(int cpuid)
```

This should be done after the operating system knows the timing information of all your tasks. That is, after you have made all of your tasks periodic at the beginning of your application, or after you create a periodic task dynamically, or after changing the period of a task. The cpuid parameter of the function rt_spv_RMS() is only used by the multi uni-processor scheduler.

Note that the function only makes sure that priorities of periodic tasks are assigned accordingly to their periods. The RTAI schedular does not check if this leads to a feasible schedule, where all tasks meet their deadlines. This would also be impossible for the schedular, because it only knows the periods of tasks and not their execution times and deadlines. Hence, the programmer has to ensure schedulability, for instance, by using Rate Monotonic Analysis (RMA) as can be found in any textbook on OSes or scheduling.

Earliest Deadline First (EDF)

Another scheduling policy supported is Earliest Deadline First (EDF). Such a policy schedules always first the task with the earliest deadline. Hence, besides information about when to resume the task (e.g., by defining a

period), the scheduler needs to know the deadline of each period. To realize this in RTAI, a task must call at the beginning of every run of one instance of the task, the function

```
void rt_task_set_resume_end_times(RTIME resume_time, RTIME end_time);
```

Here *resume_time* specifies the next release time of the task and *end_time* the next deadline. Hence, no specific periodicity is assumed. If the arguments are positive they denote absolute times. If the *resume_time* is negative, it means that it is relative to its previous *resume_time* and, if negative, also *end_time* is taken as relative to the previous *resume_time*, i.e., for negative values we obtain new absolute values as follows:

```
rt_task.resume_time -= resume_time;
rt_task.end_time = rt_task.resume_time - end_time;
```

It is possible to mix up all the scheduling policies, i.e., the ones above and SCHED_FIFO and RR. It is up to the programmer to insure RMS behaves correctly, e.g., by assigning lower priorities (higher numbers) to non periodic tasks. When EDF tasks come into the game a design choice has to be made for the case they mix up with tasks based on other policies. For instance, once could decide to run EDF tasks as the highest priority ones, i.e., any EDF task will run ahead of any other non-EDF task when its resume time expires.

Example EDF

Since there is little documentation about the use of EDF in RTAI, it is useful to study an example of EDF usage:

```
/*

This example features NTASKS realtime tasks running periodically in EDF mode.
The task are given a priority ordered in such a way that low numbered tasks
have the lowest priority. However the tasks execute for a duration proportional
to their number so that, under EDF, the lowest priority tasks run first.
So if they appear increasingly ordered on the screen EDF should be working.

*/

#include <linux/module.h>

#include <asm/io.h>

#include <asm/rtai.h>
#include <rtai_sched.h>

#define ONE_SHOT

#define TICK_PERIOD 10000000

#define STACK_SIZE 2000

#define LOOPS 3
//#define LOOPS 1000000000

#define NTASKS 8

static RT_TASK thread[NTASKS];

static RTIME tick_period;

static int cpu_used[NR_RT_CPUS];

static void fun(long t)
```

```
{
	unsigned int loops = LOOPS;
	while(loops--) {
		cpu_used[hard_cpu_id()]++;
		rt_printk("TASK %d with priority %d in loop %d \n", t, thread[t].priority,loops);
		rt_task_set_resume_end_times(-NTASKS*tick_period, -(t + 1)*tick_period);
	}
	rt_printk("TASK %d with priority %d  ENDS\n", t, thread[t].priority);
}


int init_module(void)
{
	RTIME now;
	int i;

#ifdef ONE_SHOT
	rt_set_oneshot_mode();
#endif
	for (i = 0; i < NTASKS; i++) {
		rt_task_init(&thread[i], fun, i, STACK_SIZE, NTASKS - i - 1, 0, 0);
	}
	tick_period = start_rt_timer(nano2count(TICK_PERIOD));
	now = rt_get_time() + NTASKS*tick_period;
	for (i = 0; i < NTASKS; i++) {
		rt_task_make_periodic(&thread[NTASKS - i - 1], now, NTASKS*tick_period);
	}
	return 0;
}


void cleanup_module(void)
{
	int i, cpuid;

	stop_rt_timer();
	for (i = 0; i < NTASKS; i++) {
		rt_task_delete(&thread[i]);
	}
	printk("\n\nCPU USE SUMMARY\n");
	for (cpuid = 0; cpuid < NR_RT_CPUS; cpuid++) {
		printk("# %d -> %d\n", cpuid, cpu_used[cpuid]);
	}
	printk("END OF CPU USE SUMMARY\n\n");
}
```

In fact, part of the information given above is derived by looking into the source of RTAI in /usr/src/, such as the definition of
rt_task_set_resume_end_times:

```
void rt_task_set_resume_end_times(RTIME resume, RTIME end)
{
  RT_TASK *rt_current;
  long flags;

  flags = rt_global_save_flags_and_cli();
  rt_current = RT_CURRENT;
```

```
    rt_current->policy   = -1;
    rt_current->priority =  0;
    if (resume > 0) {
        rt_current->resume_time = resume;
    } else {
        rt_current->resume_time -= resume;
    }
    if (end > 0) {
        rt_current->period = end;
    } else {
        rt_current->period = rt_current->resume_time - end;
    }
    rt_current->state |= RT_SCHED_DELAYED;
    rem_ready_current(rt_current);
    enq_timed_task(rt_current);
    rt_schedule();
    rt_global_restore_flags(flags);
}
```

Example Task Definition

In this exercise we want to schedule sets of tasks, each with some computation time C and period P. Instead of using real tasks, we simulate task timing by a means of a dummy task which runs periodically with period P and keeps the processor busy during time C, we call this a *busy sleep for time C*.

How to do a busy sleep?

In the API of RTAI linux we have the function

```
void rt_busy_sleep(int ns)
```

with the following specification: Delay/suspend execution for a while. *rt_busy_sleep* delays the execution of the caller task without giving back the control to the scheduler. This function burns away CPU cycles in a busy wait loop so it should be used only for very short synchronization delays.

With this function we can easily make a dummy task which runs this *rt_busy_sleep* for a certain number of nanoseconds for each time it is called.
However, after some testing, it turns out that it does a busy sleep till some time is reached. When another task interrupts it for some time, it will still finish at the same moment as when it was not interrupted. So it is not suitable for our task simulation.

Hence, we decide to implement the busy sleep by running an empty for loop for some counter value *loop_size*:

```
for (i = 0; i < loop_size ; i++) {}
```

The main point is to determine the value of *loop_size,* e.g., such that it occupies the processor for 1 microsecond. This is done by running the for loop for some initial counter value and record the time it takes. Using this measured time we can calculate the counter value for 1 microsecond. The following code implements this :

```
    loop_size=1e9;
    rt_printk("Do counter loop from 0 to %lld.\n",countersleep);
    starttime = rt_get_time_ns();
    for (i = 0; i < loop_size ; i++) {}
    sleeptime_ns=rt_get_time_ns()-starttime;
    rt_printk("TIME PASSED in ns %lld.\n",sleeptime_ns);
```

```
   sleeptime_in_us = sleeptime_ns/1000;
   loop_size_need_for_one_us_busysleep=loop_size/sleeptime_in_us;
   rt_printk("loop size needed for 1 us : %lld.\n",loop_size_need_for_one_us_busysleep);
```

With the calculated loop_size_needed_for_one_us_busysleep value we can implement our desired busysleep function. In the busysleep.c example below we implemented a busysleep function and we use it to make a dummy task witch uses the processor for 200 microseconds :

```
/*
   busysleep.c
       runs in a dummy_task a busy_sleep for 200 microseconds
*/


#include <linux/kernel.h>    /* decls needed for kernel modules */
#include <linux/module.h>    /* decls needed for kernel modules */
#include <linux/version.h>    /* LINUX_VERSION_CODE, KERNEL_VERSION() */
#include <linux/errno.h>    /* EINVAL, ENOMEM */

/*
  Specific header files for RTAI, our flavor of RT Linux
 */
#include "rtai.h"
#include "rtai_sched.h"
#include <rtai_sem.h>

MODULE_LICENSE("GPL");


/* globals */

#define CALIBRATION_PERCENTAGE  100
#define CALIBRATION_LOOP_SIZE  1e5
                        // or 1000000000LL

#define STACK_SIZE 2000
#define NTASKS 1
#define HIGH 100
#define MID 101
#define LOW 102

static RTIME count_need_for_one_us_busysleep;
static SEM sync;
static RT_TASK tasks[NTASKS];
static int switchesCount[NTASKS];

static RT_TASK init_task_str;

// executes a loop from 0 till count
//  returns : time in ns to execute the loop
inline RTIME loop(RTIME count) {
//RTIME loop(RTIME count) {
   //RTIME i;
   unsigned long int i;
   RTIME starttime;
   RTIME sleeptime_ns;
```

```
    //rt_printk("Do counter loop from 0 to %lld.\n",count);
    starttime = rt_get_time_ns();
    for (i = 0; i < count ; i++) {}
    sleeptime_ns=rt_get_time_ns()-starttime;

    return sleeptime_ns;
}

/* function to callibrate busysleep function */
void calibrate_busysleep(void)
{
    RTIME sleeptime_ns;
    RTIME x;


    volatile RTIME loop_size=CALIBRATION_LOOP_SIZE;
    //RTIME loop_size=CALIBRATION_LOOP_SIZE;

    // loop with global CALIBRATION_LOOP_SIZE
    sleeptime_ns=loop(loop_size);
    //sleeptime_ns=loop(CALIBRATION_LOOP_SIZE);
    rt_printk("sleeptime_ns=%lld\n",sleeptime_ns);

    // EXPLANATION CALCULATION :
    //    sleeptime_in_us = sleeptime_ns/1000
    //    count_need_for_one_us_busysleep=calibration_loop_size/sleeptime_in_us;
    //     -> add calibration factor to sleeptime_in_us :
    //          sleeptime_in_us -> sleeptime_in_us *  calibrationpercentage/100
    //     -> combine everything
    //       counterbusysleepns= calibration_loop_size*10*calibrationpercentage/sleeptime_ns
    x=CALIBRATION_LOOP_SIZE*10*CALIBRATION_PERCENTAGE;
    do_div(x,sleeptime_ns);


    // set global calibration value
    count_need_for_one_us_busysleep =x;
}


/*
    RTIME busysleep(RTIME sleeptime_us )

      usage :
         sleeptime_ns busysleep(sleeptime_us)

      description :
         keep the processor busy in a loop for a sleeptime_us amount
         of microseconds

      returns :
         the real time passed during this loop  (if other tasks get
         scheduled during this function this time can become much larger than
         the time this function keeps the cpu busy)
*/
RTIME busysleep(RTIME sleeptime_us ) {
    RTIME sleeptime_ns;
```

```
    //volatile RTIME sleep_count;
    RTIME sleep_count;

    sleep_count= count_need_for_one_us_busysleep*sleeptime_us;
    sleeptime_ns=loop(sleep_count);
    return sleeptime_ns;
}

// calibrate busy sleep
static void init(long arg)
{
    rt_printk("------------------------------------------ init task  started\n",arg);

    // first calibrate busysleep for the speed of the current machine
    // no other tasks are allowed to run to garantee optimal calibration
    calibrate_busysleep();
    rt_printk("count_need_for_one_us_busysleep=%lld\n", count_need_for_one_us_busysleep);

    rt_printk("------------------------------------------ init task  ended\n",arg);
    return;
}


// task which calibrates busysleep and tests it
static void dummy_task(long arg)
{
    RTIME sleeptime_ns;
    RTIME sleeptime_us;

    rt_printk("RESUMED TASK #%d (%p) ON CPU %d.\n", arg, &tasks[arg], hard_cpu_id());
    rt_sem_wait(&sync);

    rt_printk("------------------------------------------ task %d started\n",arg);

    //  do a test sleep
    rt_printk("\nTEST SLEEP for 1000 us\n");
    sleeptime_us=1000;
    sleeptime_ns=busysleep(sleeptime_us);
    //  as long nothing else is happening on this machine, the returned
    //  realtime should match the  time the processor is kept busy
    rt_printk("RESULT SLEEPTIME in ns : %lld.\n\n",sleeptime_ns);

    rt_printk("END TASK #%d (%p) ON CPU %d.\n", arg, &tasks[arg], hard_cpu_id());
    rt_printk("------------------------------------------ task %d ended\n",arg);
    return;
}


static void signal(void)
/* signal handler, executed when a context switch occurs */
{
    RT_TASK *task;
    int i;
    for (i = 0; i < NTASKS; i++) {
        if ((task = rt_whoami()) == &tasks[i]) {
            switchesCount[i]++;
            rt_printk("Switch to task #%d (%p) on CPU %d.\n", i, task, hard_cpu_id());
            break;
```

```
        }
    }
}


// start the realtime tasks with a specific scheduling
int init_module(void)
{

    printk("Start of init_module\n");

    rt_set_oneshot_mode();
    start_rt_timer(1);

    printk("INSMOD on CPU %d.\n", hard_cpu_id());
    rt_sem_init(&sync, 0);

    // calibrate busy sleep
    rt_task_init(&init_task_str, init, 0, STACK_SIZE, LOW, 0, 0);
    rt_task_resume(&init_task_str);

    // start busysleep task
    rt_task_init(&tasks[0], dummy_task, 0, STACK_SIZE, LOW, 0, 0);
    rt_task_resume(&tasks[0]);
    rt_sem_broadcast(&sync);

    printk("End of init_module\n");

    return 0;
}


void cleanup_module(void)
{
    int i;

    stop_rt_timer();
    rt_sem_delete(&sync);
    for (i = 0; i < NTASKS; i++) {
        printk("number of context switches task # %d -> %d\n", i, switchesCount[i]);
        rt_task_delete(&tasks[i]);
    }
    rt_task_delete(&init_task_str);
}
```

NOTES :

- The CALIBRATION_LOOP_SIZE variable which is used for calibrating the busysleep is set to 1e9, however for slow machine this may be to big. In that case change it to a lower value.
- Normal division is not allowed in kernel code, therefore division in the above code is done with the function do_div. From the linux kernel source we get :

```
/*
 * do_div(dividend,divisor)
 *
 * do_div() is NOT a C function. It wants to return
```

```
 * two values (the quotient and the remainder), but
 * since that doesn't work very well in C, what it
 * does is:
 *
 * - modifies the 64-bit dividend _in_place_ with the
     quotient
 * - returns the 32-bit remainder
 *
 * This ends up being the most efficient "calling
 * convention" on x86.
 */
```

Exercises

Exercise 9a.

Implement using the above busysleep.c a program which runs 3 dummy tasks with priority 1 and which are scheduled with standard FIFO scheduling.

Exercise 9b.

Implement a program which runs 3 dummy tasks with priority 1,2,3 and which are scheduled with standard FIFO scheduling.
(in fact this is already RMS scheduling)

Exercise 9c.

Build and run the edf example given in this exercise. Explain what is happening.
How could we change we program so that edf scheduling starts immediately?
Do this and test the resulting code.

Exercise 9d.

For the task given below simulate the schedules using RMS.
Use the code given in simulate.c

Task	Computation time	Period
τ1	C1 = **3**	T1 = **6**
τ2	C2 = **2**	T2 = **9**
τ3	C3 = **2**	T3 = **11**

Exercise 9e.

Rewrite the RMS program of 9d. so you can use it for EDF scheduling.
For the task given in 9d. simulate the schedules using EDF.

Exercise 9f.

For the task given below simulate the schedules using RMS and EDF.

Task	Priority	Computation time	Period
τ1	**3**	C1 = **2**	T1 = **7**
τ2	**2**	C2 = **4**	T2 = **16**
τ3	**1**	C3 = **7**	T3 = **31**

Exercise 9g.

For the tasks given in 9f. how much can you increase C3 theoretically so that EDF scheduling is still possible? Simulate EDF with these values. Is it also schedable in the simulation? If not, why not?
