Realtime Linux (RTAI) 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.

Description

Multitasking creates the appearance of many threads of execution running concurrently when, in fact, the kernel interleaves their execution on a 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 time slice timer
- kernel control structures
- signal handlers
- debugging and performance monitoring values

Scheduling in RTAI Linux

- In RTAI Linux, the real-time scheduler is driven by timer interrupts from the PC's 8254 Programmable Interval Timer (PIT) chip.
- The real-time scheduler is started at the same time with the real-time timer with the *start_rt_timer*.
- Each separate CPU has its own real-time timer where each can run in periodic of one-shot mode independent of all the other CPU's
- In *pure periodic mode*, the timer is programmed to expire at a fixed period. All tasks will run at a multiple of this period.
    - The advantage is that timer reprogramming is avoided, which saves about 2 microseconds.
    - The disadvantage is that tasks are forced to run at multiples of the fundamental period.
    - The internal count units used are the number of periods passed since *start_rt_timer* was called. The function *rt_get_time()* then returns the numbers of periods passed since *start_rt_timer* was called.
- In *one-shot mode*, the timer is reprogrammed at the end of each task cycle, so that it expires exactly when the next task is scheduled to run.

    - The internal count units used are directly the TSC count for CPUs having a time stamp clock (TSC).The function *rt_get_time()* returns it.
    - The advantage is that tasks timing can vary dynamically, and new tasks can be added without regard to any fundamental period.
    - The disadvantage is that timer reprogramming overhead recurs continually. Notice that the timer will always run on a frequency lower than the time stamp clock (TSC) of the CPU. Therefore it has to precisely programmed so that a tick of the timer happens when a specific process should be spawned at some precise TSC clock value. Then when e.g. a task is programmed to be launched single shot at time t1 and a call is made to launch another task to be launched at time t2 than the timer has to be reprogrammed so that it ticks at both t1 and t2. Thus at every timing request in TSC units the timer should be reprogrammed to met the demands.
- Each separate CPU has its own real-time timer where each can run in periodic of one-shot mode independent of all the other CPU's

Starting the Timer and Real-time Scheduler

- The first thing we always have to do at the start of a real-time program is to start the real-time timer with the scheduler using the *start_rt_timer* command.
    - We start the timer to expire in pure periodic mode by calling the functions

```
void rt_set_periodic_mode(void);
RTIME start_rt_timer(RTIME period);
```

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

```
void rt_set_oneshot_mode(void);
RTIME start_rt_timer(1);
```

      The argument for *start_rt_timer* is chosen randomly because we don't have a period an it will be therefore be ignored by *rt_start_timer*.
- The value passed to 'start_rt_timer()' in periodic mode is the period of type RTIME "internal counts".
- The function

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

  can be used to convert time in nanoseconds to these RTIME internal count units.
- The requested period in periodic mode is quantized to the resolution of the 8254 chip frequency (1,193,180 Hz). Any timing request not an integer multiple of the period is satisfied at the closest period tick.

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, 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 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.
  - '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 (see Example 9 for information 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 (1,073,741,823).
  - '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 its filled-in RT_TASK structure to the function

```
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" ('rt_get_time()').
- 'period' is the task's period, in RTIME units, which will be rounded to the nearest multiple of the fundamental timer period.

- The task will now execute indefinitely every 'period' counts.

one-shot mode

- The task can be simple start immediately with

```
rt_task_resume()
```

- 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" ('rt_get_time()').
  - 'period' is now a dummy value which is not used by the scheduler in one-shot mode

Example : Hello world!

```
#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!"); /* 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 */
       0, /* lowest is fine for our 1 task */
       0,       /* uses floating point; we don't */
       0);      /* signal handler; we don't use signals */
       //10-i, /* lowest is fine for our 1 task */
       //RT_SCHED_LOWEST_PRIORITY-i, /* lowest is fine for our 1 task */
    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;
}
```

Assignment #2

1. Compile the *Helloworld* example above using *Makefile* taken from SLAX-RT tutorial.

   - Put the *Helloworld* source code in a new folder together with the *Makefile*
   - Edit the *Makefile* by changing the name of object file (.o) in the line "**obj-m := ….....**" according to your preference
   - Run make command to build the module
   - Insert the module to kernel and report the result
   - If there is an error about inserting the module, you might need to insert ***rtai_hals.ko*** and ***rtai_lxrt.ko*** located at ***/usr/realtime/modules*** then try to insert your module again.
   - Report the result

2. Write a program which runs 5 tasks in one-shot mode running the following function :

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

Show the result and explain how your program works.