Assignment 5: Real-time Task Model in vxWorks (100 points) – CSE 438, Fall 2012

Assignment Objectives
1. To learn the development of thread-safe library for periodic and sporadic tasks in vxWorks
2. To learn the hook mechanism on context switch events in vxWorks
3. To integrate execution time measurement and overrun handling for task management.

Project Assignment

For real-time systems, tasks are modeled as periodic and aperiodic (sporadic) and are with deadlines and worst-case execution times. However, most RTOS with priority scheduling don’t support the notion of periodicity and the bound of execution time. For instance, in vxWorks, a task can be created with a call to taskSpawn():

```c
id = taskSpawn( name, priority, options, stacksize, main, arg1, ...arg10 );
```

The `taskSpawn()` routine creates the new task context, which includes allocating the stack and setting up the task environment to call the main routine (an ordinary subroutine) with the specified arguments. Thus, for real-time tasks, the `main` routine must be programmed to consist of an iteration of computation triggered by a periodic timer or by an event flag. However, this approach is short of any control for task overrun (executing for a duration longer than the task’s WCET).

In this project, you are requested to design a wrapper that consists of a set of routines to enable the model of real-time tasks. The first core API is the `pRTtaskSpawn()` routine:

```c
id = pRTtaskSpawn( name, priority, options, stacksize, period, WECT, main, arg1, ...arg10 );
```

The `pRTtaskSpawn()` routine will set up necessary structures for a real-time task and spawn the task (by calling `taskSpawn`) which invokes the `main` procedure periodically. Therefore, there is no need to have any real-time task control in the `main` procedure and it can be a simple invocation of computation. The parameters of `pRTtaskSpawn` are:

- `name`, `priority`, `options`, `stacksize`, `main`, `arg1`, ...`arg10`: same as `taskSpawn`
- `period`: the time separation between two consecutive invocations of periodic tasks or the period
- `WECT`: worst-case execution time of each invocation

Besides the `pRTtaskSpawn` routine, the control for task overrun should be in place such that, if the execution time of the current invocation has exceeded WCET, the priority of the overrunning task will be downgraded to the lowest priority level. Note that the reduced priority should be brought to the normal level once the next period starts.

The second core API is to set up a sporadic server for aperiodic tasks in a message queue:

```c
id = sRTtaskSpawn( name, priority, options, stacksize, period, budget, msgQid, main, arg1, ...arg10 );
```

The `sRTtaskSpawn` initiates a simple sporadic server task with a period and a budget and invokes `main` procedure to serve aperiodic requests queued in `msgQid`. You may assume that each message represents an aperiodic request and is with the type `Ap_Req_type`. We will ignore the content of the message. The sporadic server task will be scheduled according to its priority by vxWorks when there is any execution budget, and will be run in the background when the budget is exhausted.

For time management, including periods, minimal separation gaps, and the detection of task overruns, you are required to use timer or watchdog timer libraries in vxWorks. A timer can be
configured to indicate the beginning of a new period and the occurrence of a task overrun. You can also read the timer value to get the elapsed time.

To detect task overruns, you will need to implement a measurement of task execution time at run time. In real-time systems, a task may be preempted (by high priority tasks) or blocked (due to unavailable resources locked by low priority tasks). The measurement of execution time at user application level is difficult, since we cannot predict the instances of preemption and blocking. On the other hand, it can be done easily at kernel level. As shown in the following diagram, a task is created, dispatched to run, switched off to waiting or ready state, and deleted (terminated). The measurement of execution time can be done once we mark the instances it is dispatched and accumulate the time it has spent when it is switched off or terminated.

To keep track of the execution time of a task which is preempted in multiple times, we can use vxWorks' task hooks which allow additional routines to be invoked whenever a task is created, switched and deleted. The hooks can be added and deleted using taskCreateHookAdd(), taskCreateHookDelete(), taskSwitchHookAdd(), taskSwitchHookDelete(), taskDeleteHookAdd(), and taskDeleteHookDelete() which are included in taskHookLib. When a hook is called, the pointer to the task’s TCB is passed as a parameter. Thus, we can access TCB and use the spare space in TCB to record additional task information (such as the accumulated execution time). Also, at a task switching instant, you can set up a timer which will be expired after \( WCET - \text{accumulated execution time} \). A task overrun can then be detected when the timer expires first before the next task switch.

Due Date
This is assignment is due on Friday, Dec. 7, at 11:59pm.

What to Turn in for Grading
- Create a working directory to include your source and object files, and makefiles for the wrapper library, and an application to test the functions. A short description (in pdf) on your test results should be included to show that the functions work properly.
- Comment your source files properly and rewrite the readme file to describe the functions of each routine.
- Compress the directory into a zip archive file named cse438-lastname-assgn05.zip. Please make sure you have the proper name in the zip file. Otherwise, 10 point will be deducted.
- Submit the zip archive to Blackboard by the due date and time.
- Failure to follow these instructions may cause an annoyed and cranky TA or instructor to deduct points while grading your assignment.

Hints on the Assignment
1. The project is not about schedulability analysis. Periodic tasks and sporadic servers are executed according to their assigned priorities when there is no overrun.

2. Let’s consider periodic tasks. Most RTOS’s don’t have any notion of period nor WCET. Thus, we program a periodic computation as a task which contains a while loop such as

```c
void taskentry()
begin
    while (,)
    {
        perform computation;
        sleep until next period;
    }
end;
```

Note that this is no overrun control either. When we create this task, we call `taskSpawn` with the function point `taskentry`.

3. For the assignment, we need to add an API `pRTtaskSpawn` for periodic computation. This is basically a wrapper function and should call `taskSpawn` to create a task with period and overrun control. You can use a skeleton such as

```c
void taskentry_periodic()
begin
    while (,)
    {
        start overrun control;
        perform computation;
        stop overrun control;
        sleep until next period;
    }
end;
```

where the computation is in a given function and the period is an input parameter to `pRTtaskspawn`. When `pRTtaskSpawn` is called, it can set up proper data structures, and eventually call `taskSpawn` to create a task with the entry `taskentry_periodic`.

4. The detection of task overruns and the budget replenishment are driven by timed events which are managed by a timer. For task overruns, we need to accumulate the execution times of all tasks. The way we do this is to utilize the switch hook provided by RTOS.

5. Note that the wrapper functions may be called multiple times. Thus, multiple threads will be created and run the while loops concurrently. If there are any global variables shared by these threads, proper access control to these variables must be in place to ensure the correct operations.