CS 4390/5372: Specification and Design of Real-Time Systems
Lab2: Task Control

Introduction: Modern real-time systems are based on the complementary concepts of multitasking and inter-task 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 inter-task communication facilities, to be practiced later, allow these tasks to synchronize and coordinate their activities. The VxWorks multitasking kernel, *wind*, uses interrupt-driven, priority- based task scheduling. It features fast context switch time and low interrupt latency.

The following are the primary objectives of this experiment:

1. To understand how to initiate multiple threads using Vxworks tasking routines.
2. To understand how to pass parameters to spawned tasks
3. To introduce various methods of running functions as tasks from the WindSh (*repeat, period, sp*)
4. To identify running tasks and the task states (using *i* and *taskShow*)
5. To experiment with task delays (*taskDelay* and *nanosleep*)
6. To introduce Windriver's Workbench Debugger

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:

- 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
- delay timer
- times slice timer
- kernel control structures
- signal handlers
- debugging and performance monitoring values

Task Creation and Activation: The routine *taskSpawn* creates the new task context, which includes allocating and setting up the task environment to call the entry routine with the specified arguments. The arguments to *taskSpawn()* are the new task's name (an ASCII string), priority, an "options" word (also hex value), stack size (int), the entry routine address (routine name), and ten integer arguments to be passed to the entry routine as startup parameters. Task can be spawned from WindShell command line using *sp* (which is a shell version of *taskSpawn*). For a repeated execution of a routine *repeat* or *period* from the command line can be used.

Syntax

id = taskSpawn(name,priority,options,stacksize,function,arg1,..,arg10);

Requirements:

We shall use Windriver's Workbench and the hardware target. However, you may also experiment with using simulated target environment using vxsim. ***Use one-side paper and single spacing for the report***.

1. Complete the described experiment. Record all pertinent commands that you have executed and their results. Attempt to understand and explain the

significance of each step. As part of this experiment, you will have to use the debugger provided by workbench to set breakpoints and examine task status. For additional information refer to the vxWorks and workbench help content in workbench.

2. Prepare lab report using the prescribed format. Include what you performed at each phase of the lab and what results you received. In the report identify by each step of the experiment letter and number (e.g. A3, A5) and respond to all underlined questions. Include descriptions of all procedures/activities, results, and observations, specifically the shell commands and their outcomes.

Task Priorities and Execution Order

A1. Follow the procedure from previous lab to create a new project and add the source code below to the project.

```
#include <vxWorks.h> /* include this as the first thing in every program */
#include <stdio.h> /* we use printf */
#include <taskLib.h> /* we use taskSpawn */
#include <sysLib.h> /* we use sysClk... */
#include <private\trgLibP.h>
int global_howmany=-1;


int printing(int arg) /* subroutine to be spawned */
{
  int k; /* counter to repeat printing "arg" times */
  for (k=0;k<arg;k++)
    {
     printf("I am task %d, iter: #%d\n",taskIdSelf(), k); /* Print task Id */
     taskDelay(sysClkRateGet()/60); /* delay for 1/60 of second */
    }
/*trigger user-event 10 to make WV stop logging when the last
task has finished printing. Assuming all tasks are spawned
with the same priority level. If this is not the case, a
different triggering method needs to be used to stop WV.*/

if (arg==global_howmany)
  {
    trgEvent(40010);
    global_howmany=-1;
  }
return(0);
}

void spawn_tasks(int howmany) /* subroutine to perform the spawning */
  {
   int i, taskId;
   global_howmany=howmany;
   for(i=1; i <= howmany; i++)  /* Creates "howmany" tasks */
     {
       taskId = taskSpawn(NULL,90,0x100,2000,(FUNCPTR)printing,i,0,0,0,0,0,0,0,0,0);
     }
}
```

A2. Execute *printing* from the shell. **How do you do it, where are the results shown?**

A3. Spawn the *printing* from the shell using *sp, repeat,* and *period* commands. Use WindSh *help* function to find out the syntax of these commands. **Observe the output and the status of executing tasks. Note the naming convention for the spawned tasks. Where are the results shown?**

A4. Execute *spawn_tasks(50).* **Observe currently executing tasks in various state using *i* shell function (you need to do it very fast, immediately after pressing <enter> when executing *spawn_tasks*, why?). Experiment with *taskShow*, what do you see?**

A5. Control task execution using suspend, resume, delete (*ts, tr, td*) from the shell. **Observe and comment on the produced output.**

A6. Use the Debugger to execute *spawn_tasks(N). E*xperiment with attaching and detaching running tasks (*printing*), breakpoints and single-stepping. Refer to the Debugger on-line manuals. **Comment on how easy/hard was it to use debugger.**

A7. The tasks execute in such order that the one with less printing lines will be executing before the one with more iterations (see example below for a case with three tasks):

I am task 73326808 <--- task 1
I am task 73225848 <--- task 2
I am task 73214976 <--- task 3
I am task 73225848 <--- task 2
I am task 73214976 <--- task 3
I am task 73214976 <--- task 3

> Modify the task priorities to reverse this order and to get printing as on the example below. **Explain the source code changes. What instruction(s) must be changed and how? (Attach your modified source code with your deliverable)**
>
> *I am task 73214976 <--- task 3*
> *I am task 73225848 <--- task 2*
> *I am task 73326808 <--- task 1*
> *I am task 73214976 <--- task 3*
> *I am task 73225848 <--- task 2*
> *I am task 73214976 <--- task 3*

A8. Attempt use POSIX delay function *nanosleep* to replace *taskDelay*. Look in the documentation to find out what happens (the header file 'time.h' will need to be included). Attach modified source code with highlighted/commented new lines. **Comment on the results of this experiment**