

CSE4354/5354 Real-time Operating Systems

Fall 2022 Project (RTOS)

1 Overview

The goal of this project is write an RTOS solution for an M4F controller that implements a preemptive RTOS solution with support for semaphores, yielding, sleep, priority scheduling, memory protection, and a shell interface.

A simple framework for building the RTOS is included in the `rtos.c` file.

2 Requirements

Scheduler:

Each time the scheduler is called, it will look at all ready threads and will choose the next task to execute. Modify the task to add prioritization to 8 levels (0 highest to 7 lowest).

Note: The method used to prioritize the relative importance of the tasks is implicit in the assignment of the prioritization when `createThread()` is called.

Kernel Functions:

Add a function `yield()` that will yield execution back to the kernel that will save the context necessary for the resuming the task later.

Add a function `sleep(time_ms)` and supporting kernel code that will mark the task as delayed and save the context necessary for the resuming the task later. The task is then delayed until a kernel determines that a period of `time_ms` has expired. Once the time has expired, the task that called `sleep(time_ms)` will be marked as ready so that the scheduler can resume the task later.

Add a function `wait(semaphore)` and supporting kernel code decrements the semaphore count and returns if a resource is available or, if not available, marks the task as blocked, and records the task in the semaphore process queue.

Add a function `post(semaphore)` and supporting kernel code that increments the semaphore count. If a process is waiting in the queue, decrement the count and mark the task as ready.

Modify the function `createThread()` to store the task name and initialize the task stack as needed. You must design the method for allocating memory space for the task stacks.

Add a function `destroyThread()` that removes a task from the TCB and cleans up all semaphore entries for the task that is deleted. Note the TCB entry will stay in the table.

Add a function `restartThread()` that restarts a program.

In implementing the above kernel functions, code the function `systickIsr()` to handle the sleep timing and kernel functions.. The code to switch task should reside in the `pendSvclsr()` function..

Add a shell process that hosts a command line interface to the PC. The command-line interface should support the following commands (many borrowing from Linux):

`ps`: The PID id, process (actually thread) name, and % of CPU time should be stored at a minimum.

`ipcs`: At a minimum, the semaphore usage should be displayed.

`kill <PID>`: This command allows a task to be killed, by referencing the process ID.

`reboot`: The command restarted the processor.

pidof <Process_Name> returns the PID of a task.

run <Process_Name> starts a task running in the background if not already running. Only one instance of a named task is allowed. You will need to write a directory to allow this to work without hard-coded names. Create thread and delete thread should store this information. The thread should start at the normal process priority (8 for this code).

pmap PID displays memory usage by the process (thread) with the matching PID.

preempt ON|OFF turns preemption on or off. The default is preemption on.

sched PRIO|RR selectes priority or round-robin scheduling. The default is priority scheduling.

3 Hints

You should start with the rtos.c code provided in class and modify the program as appropriate to test operation. If you reference a small snippet of code from a book, you must clearly reference the work and page number. You should not be incorporating code more than a few lines from other sources.

You will need to modify the createThread() function to allocate the proper stack space for each process.

It is recommended that you start a cooperative design with the yield() function only and determine the mechanism required to record the context for the current process, call the scheduler, and then restore the context of a task that was selected by the scheduler.

Once this is complete, code the sleep() function. Also add any interrupt code needed to handle the pending timer(s).

Now the wait() and post() functions can be added, which will also reuse the context saving code above. The post() function needs to be carefully examined when called from an interrupt.

Next, add the UART shell processing from Mini Project 1.

Then, add preemption to the SysTick ISR to complete the project.

Finally, enable the MPU.

A list of detailed steps will be provided in class as summarized below:

Step 0: Build HW

Step 1: Code initHw() amd readPbs()

Step 2: Incorporate your MPU code that allows RW access all memory for unprivileged and privileged code. Note that unprivileged will still not be able to access some areas of memory and some registers.

Step 3: Setup a temporary PSP area at the top of SRAM (will not be used once the OS starts) and set the ASP bit.

Step 4: Incorporate your ISR handling code.

Step 5: Incorporate your malloc and heap code.

Step 6: Modify createThread() to set a process stack and store the thread name

Step 7: Code startRtos() to call the scheduler and launch the first task as unprivileged

Step 8: Code yield() incorporating svclsr() and pendSvlslr()

Step 9: Code pendSvlslr() to push the registers, save PSP, call scheduler, restore PSP, and pop the registers to make a task switch to the same with only idle() task running.

Step 10: Code pendSvlslr() to task between two tasks (the hardest step)

Step 11: Add MPU protection code to the malloc function

Step 12: Add SVC number extraction in svclsr(), add kernel support for sleep() with systicklsr()

Step 13: Add kernel support to wait()
Step 14: Add kernel support for post()
Step 15: Write the priority scheduler
Step 16: Add kernel functions for getting cpu time and privileged information for step 17
Step 17: Write shell task with support for pmap, ipcs, kill, pidof, ps, run, preemp, sched, and reboot
Step 18: Add restartThread() and stopThread()
Step 19: Add preemption in systicklslr()

4 Deadlines and Teams

The project teams consist of 1 or 2 members. All code must be unique and written solely by the students submitting the project.

All work should be completely original and should not contain code from any other source, except the rtos.c file framework. Do not rename any of the existing functions or change priorities as this can affect the grading process, which requires these names be preserved for extraction.

When submitting your project, you should all files in the project.

If any part of your project (even 3 lines of code) is determined to not be unique, your grade will be impacted.

Please include your name clearly at the top of your program files. Please document your program well to maximize credit. Use the notes in the rtos.c file for instructions.

Code submissions are due at the time defenses begin indicated in the class syllabus with a defense. No late projects will be accepted.

Have fun!