CSE4354/5392 Real-time Operating Systems
Fall 2025 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 mutexes, 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 that 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.

Add a function lock(mutex) and supporting kernel code locks the mutex and returns if a resource is available or, if not available, marks the task as blocked on a mutex, and records the task in the mutex process queue.

Add a function unlock(mutex) and supporting kernel code unlocks the mutex. Only the thread that locked a mutex can unlock it.

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

Add a function killThread() that marks the task as killed in the TCB. Memory must be freed.

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 pendSvcIsr() 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 UNIX):

ps: The PID id, process (actually thread) name, sleep time remaining is applicable, blocked resource if applicable, and % of CPU time should be stored at a minimum.

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

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

pkill <Process_Name>: This command kills a task with the matching process name.

reboot: The command restarts 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. The stack will be recreated for a task if is was not already running.

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

sched PRIO|RR selects 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.

Start 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 lock(), unlock(), wait(), and post() functions can be added, which will also reuse the context saving code above.

Then, add priority scheduling.

Next, add the shell task incorporating the user interface from the Nano Project.

Then, add preemption to the SysTick ISR.

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() and readPbs()
Step 2: Bring in the exception/fault handling routines from the Mini Project
Step 3: Create a heap area aligned to a 512- or 1024-byte boundary
Step 4: Modify createThread() to set a process stack, store the thread name, and store the srd bits based on the stack space allocated
Step 5: Code startRtos() to call the scheduler and launch the first task as unpriviledged
Step 6: Code yield() incorporating svcIsr() and pendSvIsr()

Step 7: Code pendSvIsr() to push the registers, save PSP, call scheduler, restore PSP, restore SRD bits, and pop the registers to make a task switch to the same with only idle() task running.
Step 8: Code pendSvIsr() to task between two tasks (the hardest step)
Step 9: SVC number in svcIsr(), add kernel support for sleep() with systickIsr()
Step 10: Add kernel support to lock()
Step 11: Add kernel support for unlock()
Step 12: Add kernel support to wait()
Step 13: Add kernel support for post()
Step 14: Write the priority scheduler
Step 15: Write shell task based on your Nano proect.
Step 16: Add kernel functions for getting cpu time and privileged information
Step 17: Add killThread() and resumeThread()
Step 18: Add preemption/priority inheritance in systickIsr()
Step 19: Add MPU support based on your Mini Project

4 Deadlines and Teams

The project teams consist of 1 person. All code must be unique and written solely by the student 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. Libraries from the instructor may be freely used in this project.

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!