Real-Time Embedded Systems, 7.5 hp
Assignment 4

Before you start the Assignment

Check the "Practical Assignments and Project Seminar" learning module on the course page on Blackboard. It includes general information about the assignments and the Lab Kit.

After you complete Assignment

Once you have completed the tasks for Assignment 4, one student in the group will submit the results via the course page on Blackboard.

The assignment submission form is found within the learning module "**Submissions**".

Grading and Deadline

- **Grading Session**: Check the date and time in Time edit for "Grading Ass. 4".
- **Deadline**: By the day before the grading session at **11PM**.

Objectives

- Implement Preemptive Multitasking: Develop a preemptive multitasking system by building upon a modified version of the Tinythreads[1] library.
- Add Mutual Exclusion: Integrate mechanisms for mutual exclusion within the modified tinythreads library. This step is essential for managing shared resources among concurrently executing tasks to prevent conflicts and data corruption.
- Implement Scheduling Algorithms: Implement three distinct scheduling algorithms—Round Robin, Rate Monotonic, and Earliest Deadline First. Each algorithm will manage the order and timing of task execution.

Content

Assignment 4 includes three parts:

- Part 1 involves implementing the Round Robin scheduling algorithm and introduce mutual exclusion through mutex variables.
- Part 2 involves implementing the Rate Monotonic scheduling algorithm.
- Part 1 involves implementing the Earliest Deadline First scheduling algorithm.

Q&A

Supervision sessions will address questions, and students are encouraged to share relevant queries or comments on the Discussions forum. **Note that posting source code is not allowed.**

[1] http://www.sm.luth.se/csee/courses/d0003e/labs/lab2.html

Required Equipment and Software:

- Lab kit, including the PiFace Control and Display, along with a breadboard circuit for the LED.
- Raspberry Pi OS (if you are compiling code on it).
- For debugging purposes, consider using a serial console software like PuTTY, which works on Windows, macOS, and Linux.

References

- C reference[2] (Recommended by Wagner)
- Tiny threads[3]

[2] https://en.cppreference.com/w/c
[3] http://www.sm.luth.se/csee/courses/d0003e/labs/lab2.html

PART 1 - Round Robin with Timer Interrupts

According to the Round Robin scheduling algorithm, each task runs for some predetermined time called a time slice. After the time interval elapses, the running task is preempted, and the next task in line gets its chance to run. The preempted task doesn't get to run again until all the other tasks in that round have had their chance. [Barr, M., & Massa, A. (2006)]

In Assignment 3, you implemented a form of Round Robin scheduling, where each task runs and yields voluntarily. As a result, some tasks execute longer than others, i.e., `iexp(n)` in `computeExponential` takes longer than `n*n` in `computePower`.

In Assignment 4, such time-slicing will be achieved using timer interrupts so a scheduler can assign the maximum time "slice" to each task. This approach will give all the ready tasks equal shares of the processor, i.e., a task will run until it blocks or its timeslice expires. Consequently, a task with a longer execution time will not hold the processor longer than other tasks. This technique of enforcing yields at regular intervals is called time-sharing and is used extensively by multi-task and multi-user operating systems.

Assignment Details

Downloading source code

1. Start by downloading a4p1.zip and extracting its contents.

Adding C Libraries

2. Utilizing your results from Assignments 3, proceed to edit the following files located in `a4p1/lib/` and incorporate your own function implementations for:
   - `piface.*`
   - `expstruct.*`
     - NOTE: Use the files from a2p2, which include the iexp() function without the LED blinking interleaving.

Implementing the Round Robin Scheduling Algorithm

Note: Include relevant comments and explanations in your code to support your implementation.

3. In Assignment 3 Part 3, you achieved cooperative multitasking by running tasks that ran and yielded voluntarily. Now, the goal is to implement Round Robin with Timer Interrupts. This will involve replacing explicit calls to `yield()` with periodic interrupts. Consequently, the call to `yield()` within the functions will either be removed or commented out, as it will be substituted by involuntary subroutine calls initiated by the processor hardware.
4. In the `a4p1.c` file, you will find the `initTimerInterrupts()` function, which enables and configures the RPi ARM Timer Interrupt. With the suggested configuration, the timer interrupt will fire approximately every second and trigger the execution of its interrupt handler.
5. You will need to identify the `.c` file in the `lib/` directory that implements the handling of this timer interrupt. Once identified, write the necessary code within the timer interrupt handler function to facilitate the switching of context between running threads. This involves:
   a. Calling the `scheduler()` function, which is defined in `lib/tinythreads.h` and implemented in `lib/tinythreads.c`.
   b. Implementing the actual context switching within the `scheduler_RR()` function, which is also found in `lib/tinythreads.c`.
   c. Placing a call to `scheduler_RR()` within the `scheduler()` function.

6. **Question**: Given that the enabled interrupts may occur at any time during the execution of a given task, it is essential to study and understand the purpose of the macros `DISABLE()` and `ENABLE()`, as well as their placement within the functions in `lib/tinythreads.c`. Please create a file called `Answers.txt` and provide answers to the following questions:
    a. What is the purpose of the macros DISABLE() and ENABLE()?
    b. What potential issues could arise if a `yield()` call were injected at the least suitable location within functions like `dispatch()` or `enqueue()`? How about injecting such a call into `yield()` itself?

Generating a Kernel using the Round Robin Scheduling Algorithm

7. The `a4p1.c` file contains an example program that executes tasks concurrently. Compile the code and boot the RPi using the newly created kernel, `a4p1.img`.
    - Successful kernel creation and installation will enable you to observe the alternating execution of different threads on the PiFace Display, similar to `a4p1_without_mutex.img`.

Mutual Exclusion

After booting the RPi with the latest kernel you generated, you may have noticed that a task displays its results in different segments on the display. This indicates that the display is a shared resource among the executing threads, and calls to `print_at_seg()` in `a4p1.c` have become critical sections that must be protected.

Mutual exclusion is necessary to prevent unwanted interleaving of the execution of methods in a multithreaded environment [Course book]. To address this, you will implement the functionality of mutex variables.

Note: Include relevant comments and explanations in your code to support your implementation.

8. Implement the bodies of the functions `lock()` and `unlock()` in `lib/tinythreads.c`. The provided C code in `lib/tinythreads.h` defines a structure named mutex_block and then defines a new type alias named `mutex` for this structure. The `mutex` type is used in `lib/tinythreads.c`, already contains the fields necessary for implementing a mutex variable.
    a. The implementation for the `lock()` function must acquire the `mutex` if it's available by setting the locked flag of the mutex. If the mutex was previously locked, the running thread should be placed in the waiting queue of the mutex, and a new thread should be dispatched from the ready queue.
    b. The implementation for the `unlock()` function must activate a thread in the waiting queue of the mutex if it is non-empty. Otherwise, the locked flag shall be reset.
9. Apply `lock()` and `unlock()` in a4p1.c to achieve well-behaved printouts.
10. Compile the code and boot the RPi using the newly created kernel.
    - A successful kernel creation and installation will allow you to observe the alternating execution of different threads on the PiFace Display, similar to `a4p1_with_mutex.img`.

Note: Protecting the bodies of exported operations from interrupts is just as important here as in other parts of the kernel. Prioritize understanding what is happening over hasty coding.

Deliverables

For Assignment 4 Part 1, one student in the group must upload the following individual files:

- `Answers.txt`
- `a4p1.c`
- `The .c file containing the ISR.`
- `tinythreads.c`

Note:

- All students in the group are equally responsible for the submitted source code.
- The group ensures that the submitted code does not include cheating and plagiarism issues.

PART 2 - Rate Monotonic Scheduling

The Rate Monotonic (RM) scheduling algorithm is used to schedule real-time periodic tasks in a real-time operating system (RTOS). Tasks are assigned priorities based on their periods, with shorter periods resulting in higher task priorities. To represent these real-time periodic tasks, modifications have been made to the original `thread_block` structure in `tinythreads.c`. These modifications include:

- `` `unsigned int Period_Deadline` ``: Represents the absolute period and deadline of a task (since period and deadline are equal).
- `` `unsigned int Rel_Period_Deadline` ``: Represents the relative period and deadline of the task.

Assignment Details

Source code

1. Create a copy of the `a4p1` directory and rename it to `a4p2`.
2. Download the attached `a4p2.c` and `Makefile` into the `a4p2` directory.

Exploring a4p2.c

- In the updated `a4p2.c`, note that the `main()` function now includes calls to `spawnWithDeadline`. These calls include parameters for the start routine, its argument, period, and relative deadline for each task.
- Consider the `computeSomething` function, which has a fixed execution time of 1 tick. For example, a call to `spawnWithDeadline(computeSomething, 0, 3, 3)` spawns a task that runs `computeSomething`, taking 1 tick to complete. The deadline and relative deadline parameters indicate that this task has a period of 3 ticks, meaning it will be activated every 3 ticks.
- Three tasks are spawned when the system starts in `main()`, and the timer interrupt is initialized.
    - When the timer interrupt handler function executes and calls the `scheduler()` function, the task with a period equal to 3 will run because it has the highest priority (shorter period).
    - In the subsequent timer interrupt (i.e., ticks), the task with a period equal to 5 will run because it has the highest priority among the remaining tasks to execute.
    - Since this task also completes during its ticks, in the next ticks, the task with a period equal to 7 will run and complete its execution. Each finished task must be respawned in multiples of the `Rel_Period_Deadline` (i.e., the period).

Implementing the Rate Monotonic Scheduling Algorithm

Note: Include relevant comments and explanations in your code to support your implementation.

3. In `lib/tinythreads.c`, implement `void spawnWithDeadline(void (* function)(int), int arg, unsigned int deadline, unsigned int rel_deadline)`
    - This function spawns a real-time task, including the deadline (i.e., `Period_Deadline`) and relative deadline (i.e., `Rel_Period_Deadline`) parameters. It will be similar to the `spawn` function but with the addition of setting `Period_Deadline` and `Rel_Period_Deadline` attributes.
4. In `lib/tinythreads.c`, implement `void respawn_periodic_tasks()`

- This function spawns real-time periodic tasks that have completed execution according to their respective periods. It should be called by the `scheduler()` function at every timer interrupt and before the actual scheduling.

5. In `lib/tinythreads.c`, implement void `scheduler_RM()`
   - This function implements the RM scheduler, determining which task will run based on priority (the smallest period has the highest priority). You may need to keep the `readyQ` sorted, with the highest priority task at the head of the queue.
   - The `scheduler_RM()`function must be called by the `scheduler()` function.

Generating a Kernel using the Rate Monotonic Algorithm

6. Compile the code and boot the RPi using the newly created kernel, `a4p2.img`. Remember to adjust the Make file to use `a4p2`.
   - A successful kernel creation and installation will allow you to visualize the alternating execution of different threads, similar to `a4p2.expected.img`.

Note: It's crucial to protect the bodies of exported operations from interrupts as in other parts of the kernel. Understanding what is happening is more important than quick hacking.

7. **Question**: Consider the `computeSomething` function, which has a fixed-length execution time slightly equal to 1 tick. Remove the code `while(t==ticks);`, compile the code, and boot the RPi using the newly created kernel.
   - Create a file called `Answers.txt` and add the answer to the following question: What is the result and implication of removing the code `while(t==ticks);`? Elaborate on your answer.

Deliverables

For Assignment 4 Part 2, one student in the group must upload the following individual files:

- `Answers.txt`
- `a4p2.c`
- `tinythreads.c`

Note:

- All students in the group are equally responsible for the submitted source code.
- The group ensures that the submitted code does not include cheating and plagiarism issues.

PART 3 - Earliest Deadline First Scheduling

The Earliest Deadline First (EDF) scheduling algorithm is a priority-based scheduler that calculates the deadlines of real-time tasks dynamically, e.g., at each interrupt or timer interrupt, and adjusts priorities as appropriate. The closest deadline results in a higher task priority. When new priorities are assigned, a new or existing task might be assigned the highest priority, necessitating a context switch. As expected, EDF scheduling incurs a significant computational overhead.

Assignment Details

Source code

1. Create a copy of the `a4p2` directory and rename it to `a4p3`.
2. Download the attached `a4p3.c` and `Makefile` into the `a4p3` directory.

Exploring a4p3.c

- In the `main()` function of `a4p3.c`, there are calls to `spawnWithDeadline`, including parameters for the start routine, its argument, period, and relative deadline for each task.
- Note that, according to the EDF scheduling algorithm, task priorities must change dynamically based on their deadlines. Therefore, the scheduler will adjust the deadline (`Period_Deadline`) attribute.
- The `computeSomething` function has a fixed execution time of `1` tick. Thus, `spawnWithDeadline(computeSomething, 0, 3, 3)` spawns a task with a 3-tick period, executing `computeSomething`.

Implementation of Earliest Deadline First Scheduling

Note: Include relevant comments and explanations in your code to support your implementation.

3. The `respawn_periodic_tasks()` function remains as implemented in Part 2 and is called by the `scheduler()` function at each timer interrupt.
4. Implement the `scheduler_EDF()` function, which schedules tasks based on the EDF algorithm by selecting the task with the closest deadline.
5. Locate the code segment where you need to adjust the `Period_Deadline` of all tasks ready to execute.
6. Ensure that `readyQ` remains sorted, with the highest priority task at the head of the queue, sorting based on deadlines.

Generating a Kernel using the Earliest Deadline First Algorithm

7. Compile the code and boot the RPi using the newly created kernel, `a4p3.img`. Remember to adjust the `Make` file to use `a4p3`.
    - A successful kernel creation and installation will allow you to visualize the alternating execution of different threads, similar to `a4p3.expected.img`.

Schedulability Analysis

8. In `main()` in `a4p3.c`, change the task parameters of `spawnWithDeadline(computeSomething, 2, 7, 7)` to `spawnWithDeadline(computeSomething, 2, 4, 4)`.
9. Generate a new kernel file and boot the RPi. Observe the outcome, should be similar to `a4p3_5-3-7.expected.img`, and determine if any task misses its deadline.
    - Completing these steps will provide insight into how the EDF scheduling algorithm operates and whether tasks meet their deadlines.
10. **Question**: Does any task miss its deadline? If so, which one?

Deliverables

For Assignment 4 Part 3, one student in the group must upload the following individual files:

- `a4p3.c`
- `tinythreads.c`

Note:

- All students in the group are equally responsible for the submitted source code.
- The group ensures that the submitted code does not include cheating and plagiarism issues.