ME241 – Introduction to Computer Structures & Real-Time Systems

Lab Project 3: Timing

Prepared by:

Nathan Tang (20890522)

Eddy Zhou (20873917)

Submitted on:

2 November 2022

Part 1: Thread Modified Implementation

To allow gizmOS to support pre-emptive and cooperative context switching, modifications were made to the original thread struct node and linked list structure. The original members of the struct can be found in the *APPENDIX* section of this report. The original members of the `thread struct` node include `threadStackP` which stores the thread stack pointer, `threadFunc` which stores the function pointer to the function the thread will run, `status`, and `next` which stores a pointer to the next node in the linked list. New members added can be seen below in *Tables 1 and 2*.

`struct threadNode (added members)`	
`uint32_t runTimer;`	Stores the time before the thread will preemptively yield. It's initial value should be set to the period of the the task.
`uint32_t sleepTimer;`	Stores the time left that the thread must sleep for

Table 1. Additional thread struct members for Linked List Node and its characteristics

The new members added above, `runTimer` and `sleepTimer`, allow the OS to keep track of the amount of time a particular thread must sleep for and the amount of time before a thread must yield preemptively respectively. They are used in many of the functions mentioned later in this report.

`struct threadLinkedList (added members)`	
`threadNode* tail;`	Stores a pointer to the last thread (node) in the linked list

Table 2. Additional linked list member and its characteristic

The new member added above, `tail`, keeps track of the last thread in the node. In the last version of the lab, suggestions for an array implementation prompted for the use of a task number variable to access the threads by index. gizmOS previously used a task number to access its threads as well using iteration. However, it was realized that using a tail pointer which would point back to the beginning of the head of the linked list and storing a global pointer to the current thread that is running would be much more efficient. This will also help the OS efficiently implement Round-Robin scheduling later on.

The following addition of a tail pointer also prompted a change in `newThread()` in `_threadsCore.c` which updates the tail pointer accordingly. On the first addition of a thread, the tail and head pointer point to the same thread. On sequential additions, the head pointer is updated to point to the newly added node and the tail pointer is updated to point back to the head, making a circularly singly linked list.

Part 2: The new osYield() and scheduler()

In the new version of `osYield()`, the currently running thread has its `runTimer` reset in preparation for the next time that it runs. It then sets the current thread's status to `ACTIVE` before making space to push 16 registers. Once done, it calls on `scheduler()` which determines the next thread to run. In `scheduler()`, the `currentNode` pointer (global variable which stores the pointer to the currently running thread) simply points to the next `ACTIVE` thread in the linked list in order to implement Round-Robin scheduling. It then sets the new thread's status stored in `currentNode` to `RUNNING`. Note that this process is enveloped in a mutex, `flag`, which indicates when `osYield()` is currently in the critical section and should not be interrupted by `SysTick_Handler` to avoid inconsistent state (explained later in the report).

Part 3: osThreadSleep()

In the case that a user would like to make a thread sleep, they can do so by calling `osThreadSleep()` (found in `_kernelCore.c`) in their thread function. This sets the currently running thread's status to `SLEEPING` and sets its `sleepTimer` according to the user passed parameter. The thread will then yield by calling `osYield()`.

Part 3: SysTick_Handler

SysTick is configured in `p1_main.c` and runs the core frequency of the OS at 1 interrupt/ms. The `SysTick_Handler()` first checks to see if `osYield()` is in the critical section using the mutex boolean variable, flag. If this `osYield()` is in the critical section, return and do nothing.

`SysTick_Handler()` (found in `kernelCore.c`) then decrements the `runTimer` of the current thread and loops through the entire linked list to identify which threads are `SLEEPING` to decrement their sleepTimer. In this process, `SysTick_Handler()` also checks if a sleeping thread's `sleepTimer` is 0, in which case the thread is to wake and be put into an `ACTIVE` state (which allows it to be scheduled by `scheduler()`).

In the case that `runTimer` is 0 and the current thread is not finished its task, its timer is reset and it preemptively yields for the next thread to proceed. Note that this process is similar to `osYield()` in that it sets the `currentNode` to `ACTIVE`, makes space for the registers, calls `scheduler()` to determine the next thread, and sets the new thread to be `RUNNING`. The minor difference is that the space made is only for 8 registers to save due to context switch instead of 16 since the 8 hardware-saved registers have already been pushed due to the first interrupt.

Part 4: Testing

Testing the first of the two test cases, running three threads concurrently, one of which sleeps, one that yields, and one that is forced to context switch preemptively yields results that can be seen in *Figure 1*. The looping thread is preemptively context switched (after 5ms) to run the yielding thread. The

yielding thread instantly yields after it prints one statement. The sleeping thread is then put to sleep for 20ms allowing the context switch only to be done between both the looping and yielding threads before the sleeping thread wakes again for execution.

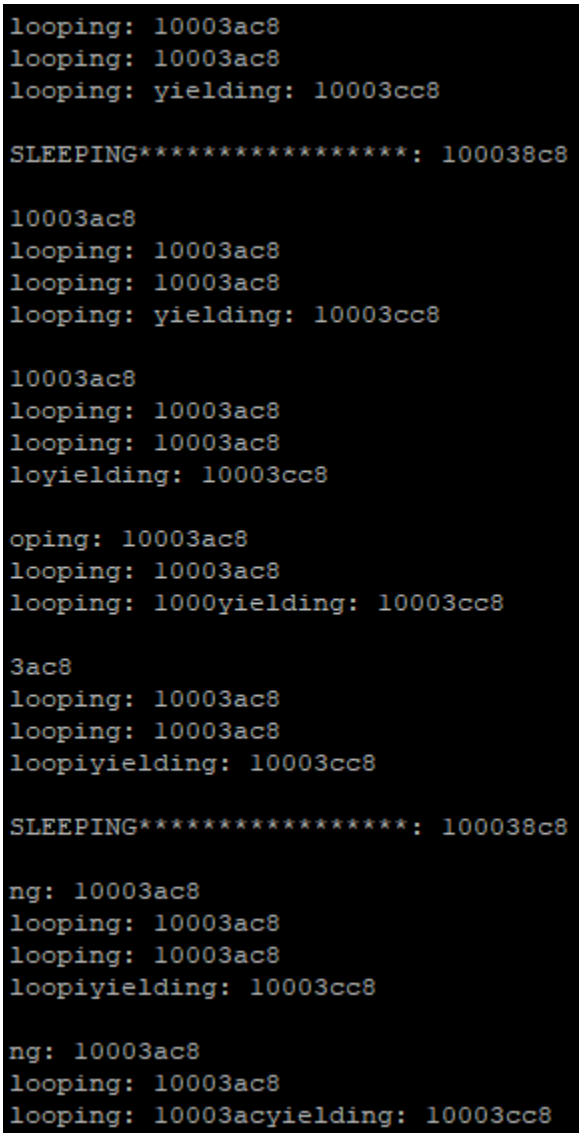

Figure 1. Running 3 different threads concurrently

In the second test case, there are two sleeping threads (each of which sleeps for a different amount of time) and an idle thread. The output can be seen in *Figures 2*. When both threads are sleeping, the idle thread is running. The first sleeping thread sleeps for 13ms while the second sleeps for 23ms.

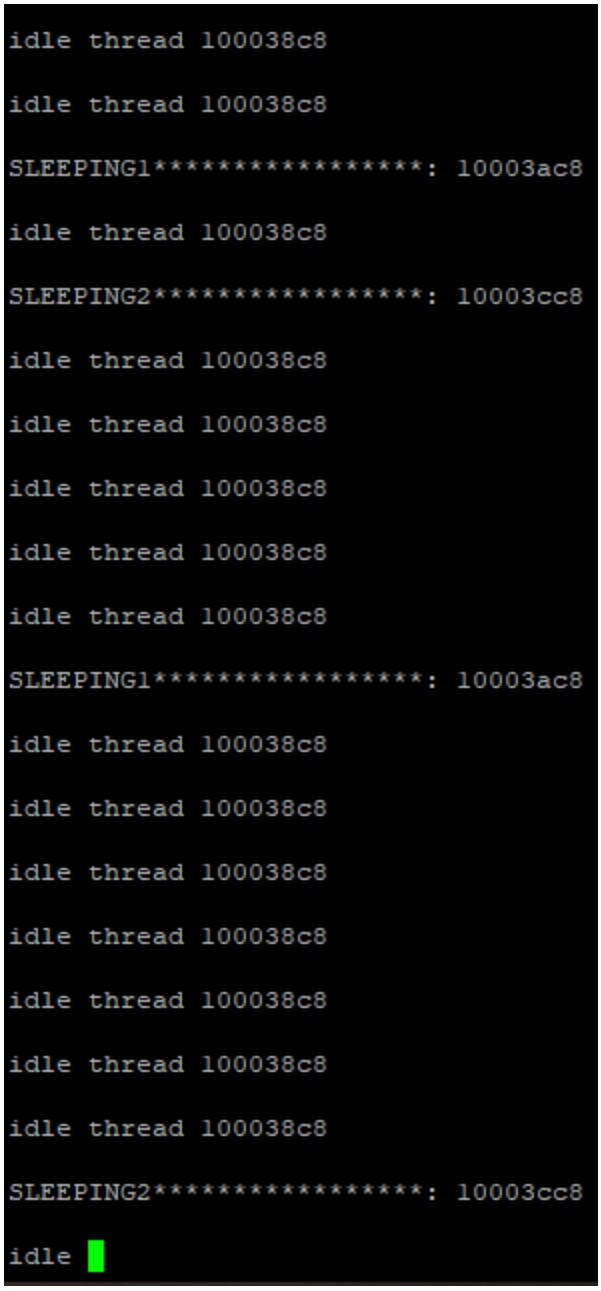

Figure 2. The start of the RTOS where each sleeping thread sleeps sequentially

It is also important to note that the following solution works only when adding a `printf()` statement in `osYield()`. This could possibly be due to some stack alignment issue which will be fixed within the next lab session as explicitly told by the lab instructor.

References

[1] “Tim Lab Project 2.” University of Waterloo. Accessed 20 Oct. 2022.

Appendix

struct threadNode	
uint32_t* threadStackP	Stores the thread stack pointer of the thread.
void (*threadFunc)(void* args)	Stores the pointer of the function to be run when the thread runs.
int status	Stores the status of the thread (such as if it is created, destroyed, etc).
struct threadNode* next	Stores the pointer to the next node in the list.

Table 3. Thread struct for Linked List Node and its characteristics

struct threadLinkedList	
threadNode* head	Stores the pointer to the head of the linked list.
int size	Stores the size of the linked list.

Table 4. Linked List characteristics