ME241 – Introduction to Computer Structures & Real-Time Systems

Lab Project 4: Pre-Emptive EDF Scheduling

Prepared by:

Nathan Tang (20890522)

Eddy Zhou (20873917)

Submitted on:

16 November 2022

Part 1: Restructuring Thread Implementation

To allow gizmOS to support pre-emptive EDF and cooperative context switching, modifications were made to the previous thread struct node used in the dynamic linked list. The original members of the struct and the same members of the dynamic linked list can be found in the *APPENDIX* section of this report. The additional members added to the following thread struct node include `period` (which stores the period of the thread based on the frequency specified by the user), `deadline` (which stores the deadline of the thread based on the frequency specified by the user) and `deadlineTimer` (which stores the time left until the deadline of a task has reached). The following definitions can be found in `osDefs.h`. With three new members in the struct, the creation of threads also changes and is discussed in Part 3 of this report.

It is also important to note that although `sleepTimer` is not a new member of the struct, it has been repurposed to also serve periodic threads. Referenced in the lab manual, the approach taken was that a "periodic thread can be thought of as a thread that sleeps for a set period, reawakens, then sleeps again for the same amount of time". This also means that the `WAITING` and `SLEEP` (although definitely not the same in real-life RTOS's or commercial RTOS's) would be treated as the same state for the purposes of gizmOS. When a periodic thread must "wait" it is instead put to sleep in a cyclical manner (as mentioned in the sections below.

Part 2: Modifying newThread()

With three new members and more user specified characteristics of the thread such as the `period` and `deadline`, it is important to note how user facing functions are changed and how to use them properly. `newThread()` (declared in `_threadsCore.c`) is used to create new threads. Its functionality does not change much from its previous implementation in Lab Project 3, except that it also receives two additional user passed arguments (deadline, `uint32_t deadline`, and `frequency, uint32_t freq`). The frequency received is converted to milliseconds/cycle by using `(1/(double)freq)*1000)` and then assigned to the thread's period. If a frequency is specified as `NULL`, the user is creating a non-periodic thread. Similarly, the deadline in milliseconds is received and stored in the thread's deadline. If a deadline is specified as `NULL`, the user is creating a thread without a deadline. The deadline is then specified to be `INF_DEADLINE` (specified in `osDefs.h`) if the user passes in `NULL` (reasoning for this is explained later when the idle thread is introduced). Regardless of what deadline is passed, it is immediately assigned to the deadlineTimer during thread initialization. The function continues to assign the new node as the head of the linked list as done in previous lab projects and then stores registers.

Part 3: The New Idle Thread

A couple of changes have been made to the idle thread. The idle thread function is now declared in `_kernelCore.c` and is implemented using `newThread()` (as previously and currently done with user created threads as well) within `kernelInit()`. By declaring, creating, and storing the idle thread in the linked list when the kernel is being initialized, it is not visible to the user and cannot be modified by the user. Most importantly, the `freq` and `deadline` arguments passed into `newThread()` when creating

this idle thread are both NULL. This specifies that the idle thread is not periodic and also sets its deadline to INF_DEADLINE (declared in osDefs.h to be an incredibly large number). Since the idle thread's deadline is INF_DEADLINE, it will never have the earliest deadline unless it is the only ACTIVE thread ready to be scheduled. Execution flow of the new and modified SysTick_Handler() (explained later in Part 4) is also used to avoid decrementing the deadline of the idle thread so it always remains low priority since its deadline is unreasonably large.

Part 4: SysTick_Handler

Much of the SysTick_Handler code also remains the same. It starts by decrementing the run timer of the currently running thread node and then loops through the entire linked list to identify sleeping thread nodes. Then, these sleeping nodes are decremented and those who have finished sleeping are returned to the ACTIVE state with the deadlineTimer set to enforce a deadline for the thread to finish. If the thread is not sleeping and it is not the tail of the list (the tail of the list is always the idle thread since it is the first to be declared in kernelInit()), it must be running or active, in which case the thread's deadline timer is decremented. If the deadline was reached (deadlineTimer is 0) then the thread is put to sleep by setting its status to SLEEP and its sleepTimer to its period. Note that is a thread is not periodic (period is equal to NULL or 0), it will essentially instantaneously become active on the next SysTick_Handler call. If the thread is periodic, it simply sleeps (synonymous to "waits") again for the specified period time.

As done in the previous lab project, a runTimer is still enforced on the currently running thread and forces the thread to pre-empt if it exceeds this time. This puts the thread back to an ACTIVE ready-to-be-scheduled state and its runTimer is reset for the next time it runs again.

The scheduler() is then called and returns the next thread node to be run. The return was added to avoid context switching excessively and only context switch when the scheduler has identified a thread that is of greater importance because its deadline is closer. If this thread node is not the same as the currently running thread node, then push the 8 PSP registers and set the currently running thread node to be the next scheduled one. Then perform a context switch to the new node.

Part 5: The New EDF scheduler()

Changes were made to the scheduler() to implement EDF (earliest deadline first). Priority is given to the thread with the closest deadline (smallest deadlineTimer) and is scheduled to be run first. When scheduler() is called, it loops through the entire list of thread nodes, identifies the ones that are not sleeping and checks its deadline. If its deadlineTimer is less than the currently stored minimum value, its deadlineTimer and index (number of iterations to linked list thread node identified) is stored. The minimum is identified and a pointer to the node to be scheduled is returned.

Part 6: Test Cases

The first test case can be found in Figure 1. The following test case implements three concurrent threads each with a frequency of 12, 100, and 256 Hertz respectively.

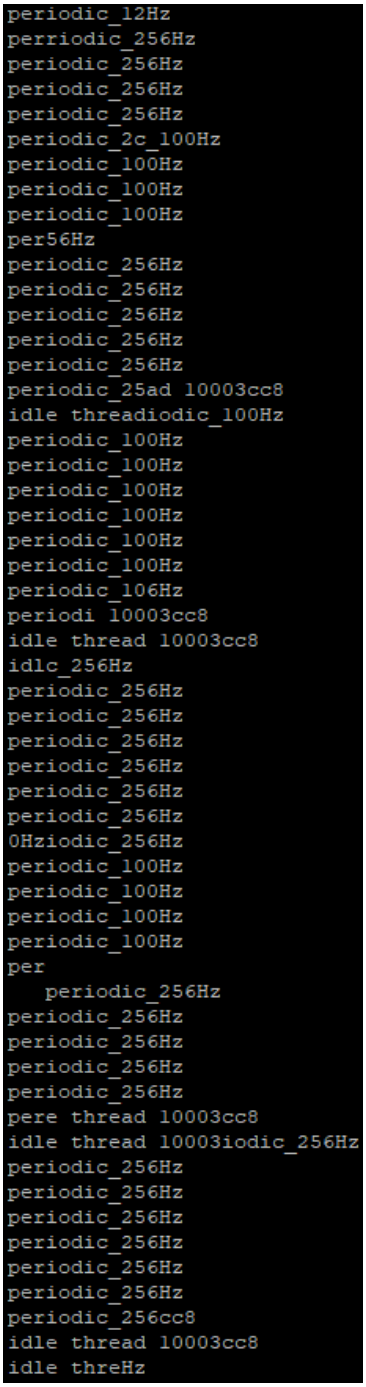

Figure 1. Three Concurrent Periodic Threads Running with 12, 100, and 256 Hertz respectively

As predicted, the thread with a higher frequency (256 Hz) executes more often than the rest of the concurrent threads since its period is shorter and therefore, is put into an active state more often. The

second most frequent thread to run is 100Hz followed by 12Hz seen at the very top of the figure. These also align with the fact that a smaller frequency causes the thread to run less frequently. There is also proof that the idle thread also works when all three threads are not ready to be scheduled.

The second test case can be found in Figure 2. One thread yields, one sleeps, and one is periodic with a frequency of 200Hz.

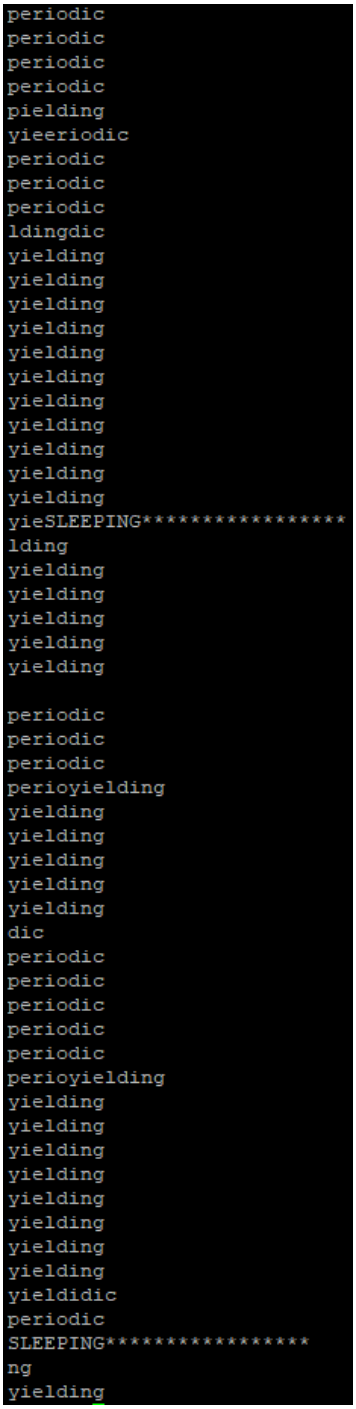

Figure 2. Three Concurrent Threads, one which yields, one which sleeps, and one periodic with 200Hz

All three threads can be seen running in the figure, with sleep occurring the least frequent (sleeping for longer periods of time that is longer than the period of the 200Hz periodic thread). The yielding thread occurs most frequently since it is active almost all the time, followed by the periodic thread which is only sometimes active.

The last test case can be seen in Figure 3. With two sleeping threads that do not sleep for times that are not multiples of each other.

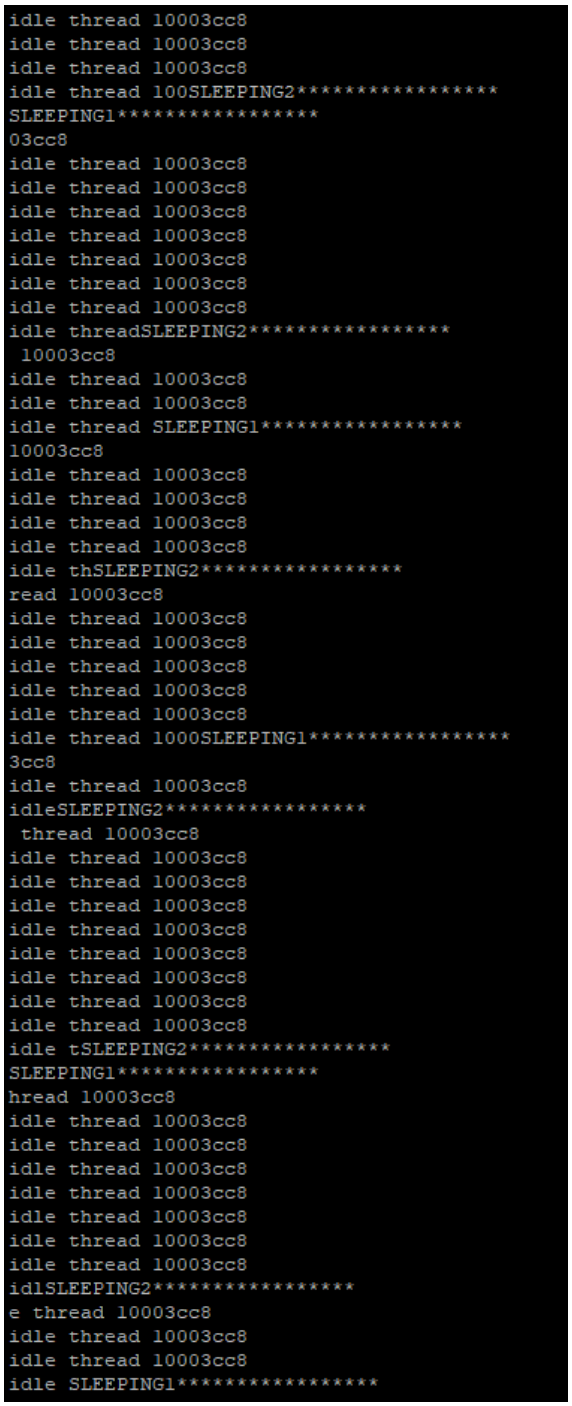

Figure 3. The idle thread runs when both sleeping threads are sleeping

As shown in the figure, both sleeping threads sleep at their own respective times and the idle thread is run whenever both threads are asleep.

Appendix

Table 1. Thread struct for Linked List Node and its characteristics

`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.
`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 2. Linked List characteristics

`struct threadLinkedList`	
`threadNode* head`	Stores the pointer to the head of the linked list.
`threadNode* tail`	Stores a pointer to the last thread (node) in the linked list
`int size`	Stores the size of the linked list.