Priority Based Pre-Emptive Task Scheduler With Dynamic Memory Allocation

Guhan Rajasekar(22410), Sai Krishna(23064), DESE, IISc

I INTRODUCTION

- A Scheduler is a function that gives the notion of *Concurrent processing* where multiple threads(tasks) are running at the same time.
- But Cortex-M has only one processor and only a single thread can run at any given time. The scheduler will run the threads one by one, and it makes it look as through multiple threads(tasks) are being run at the same time. In addition to implementing a priority based Pre-Emptive task scheduler, dynamic memory allocation has also been employed.
- In this report, the terms *thread* and *task* have been used interchangeably and both of them refer to while(1) loops that keep running indefinitely.

II TYPES OF SCHEDULER

- **Pre-Emptive Scheduler**
  - Here the main threads are suspended by a periodic interrupt. The scheduler will choose the next thread that needs to be run. Once we return from the interrupt, the new thread is launched. Here the OS will decide when a running thread needs to be suspended, and it will return the thread to the *Ready state*.
- **Non Pre-Emptive Scheduler**
  - This is also known as Co-Operative scheduler.
  - Here the main threads will decide by themselves when they need to stop running. Once a thread finishes execution, the thread will call a function that will decide what the next task must be. This scheduler does not need any interrupts.
- In this mini project, the focus is on implementing a *Pre-Emptive Scheduler* that schedules the tasks based on priority.

III TYPES OF THREADS

- **Main Thread**
  - Main threads are the ones that keep on running unless they are interrupted by interrupts or processor time is allocated to another thread pre-emptively by the scheduler. Unless any one of these two events happen, the main thread will keep on running. These threads have unbounded execution time.
- **Event Thread**
  - Event threads are attached to hardware and execute based on changes in hardware status. These threads are defined as *void - void* functions.
  - Time required to execute an event thread is short and bounded. In embedded systems, event threads are simply interrupt service routines.
- In this task scheduler, 7 main threads and 4 event threads have been implemented.

IV THREAD (TASK) CONTROL BLOCK

- Each main thread has a thread control block that holds specific information about that particular thread.
- Each TCB has the following information:
  - Pointer pointing to the stack of that particular thread.

- Pointer pointing to the TCB of the next thread.
- Integer variable that indicates the priority of that particular thread.

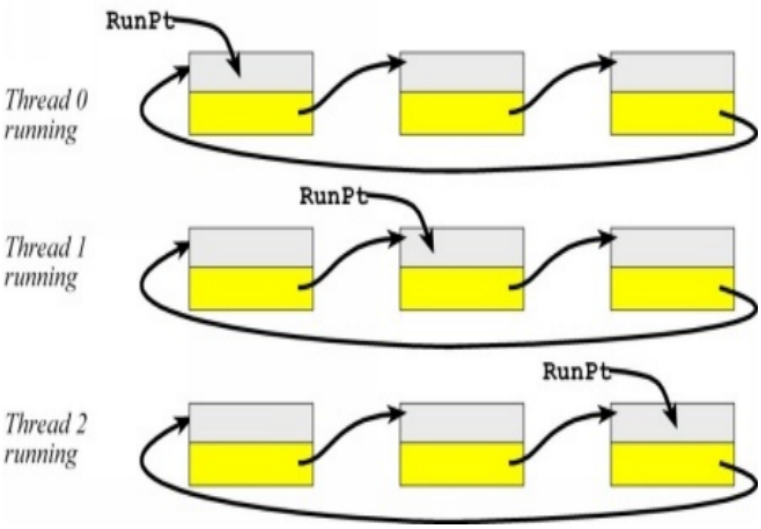

Figure 1: Linked List of TCBs. Picture Courtesy: Embedded Systems By Jonathan Valvano

V MAIN THREADS AND INTERRUPTS PRESENT IN THE PROGRAM

- The scheduler has seven main threads(tasks) and are as follows:
  - **task0**: Red led is ON continuously and count value is displayed in the righ most ssd.
  - **task1**: Blue led is ON continuously and count value is displayed in the second ssd from the right.
  - **task2**: Green led is ON continuously and count value is displayed on the third ssd from the right.
  - **task3**: White LED is ON continuously and count value is displayed on the fourth ssd from the right (leftmost ssd).
  - **task4**: Values to generate a sine wave are sent from a look-up table to LTC 1661 DAC present in the EDU - ARM board. When this task is being performed, LED colour is Magenta.
  - **task5**: This task requests for chunk of dynamically allocated memory and performs addition operations in the allocated chunk of memory. Once done with the calculations, the allocated memory is de-allocated and we start afresh. When this task is performed, LED is yellow in color.
  - **task6**: Similar to task5, this task also requests for a chunk of dynamically allocated memory, performs arithmetic operations in the dynamically allocated memory and the memory is de-allocated once the operations are done. After de-allocation process, the entire process is repeated all over again. LED is Cyan colored when this task is performed.
- The 4 keys of the last row of the Edu-Arm board act as triggers for four interrupts. The four triggers are as follows:
  - **Interrupt1**: This interrupt is triggered by the press of the left most key in the last row of the EduArm board. This interrupt reduces the priority of each main thread by 1 (increment the numerical value of priority by 1).
  - **Interrupt2**: This interrupt is triggered by the press of the second key from the left in the last row of the EduArm4 board. This interrupt will print the text ***Inhale.... Exhale.....and Repeat***. This text can be viewed in the UART terminal.
  - **Interrupt3**: This interrupt is triggered by the press of the third key from the left in the last row of the EduArm4 board. This interrupt will print the text ***This mini project is a Pre-Emptive Task Scheduler done as part of Embedded Systems Course*** and can be viewed in the UART terminal.
  - **Interrupt4**: This interrupt is triggered by the rightmost key in the last row of the EduArm board. When this key is pressed, information regarding usage of the dynamically allcoated memory is printed in UART terminal.
  - Ideally, the event based interrupts should not have any delays in them. But in this case, since the interrupts involve taking actions based on key press, some delay has been employed to overcome the de-bouncing issues that are inherently present with key presses.

VI WORKING OF THE SCHEDULER

- The main threads task0, task1, task2, task3, task4, task5 and task6 keep running in the background. First, we start with task0. Each task (thread) has a priority associated with it. Lower the value of the *priority* field in the TCB of the task, higher is the priority of that task.
- A *count* value proportional to the priority is allotted to each task. Higher the priority of task, higher is the count value allotted to that task.
- The Systick Timer is configured to generate an interrupt every one second. When the Systick Timer fires, the **Systick_Handler()** function is called. This function acts as the scheduler. Every time the scheduler is called, the count value is decremented. Once the count value reaches 0, we switch from one task to another.
- Since high priority tasks have a higher count value, it takes more time for the count value of the high priority tasks to reach 0. Hence the high priority tasks get more processor time when compared with the low priority tasks.
- When we switch from one task to another, the following operations are performed:
  - Push the contents of the registers R4-R11 in the dummy stack that is allocated for each task.
  - Make the SP register point to the sp of the dummy stack of the next task to be performed.
  - Load the contents from the dummy stack of the next task into the registers R4-R11.
  - The pointer *runpt* always points to the TCB that is currently running. During the task switching process, the scheduler also makes sure that the runpt is updated properly.

VII RESULTS

7.1 Processor Time Variation Based On Priority Level

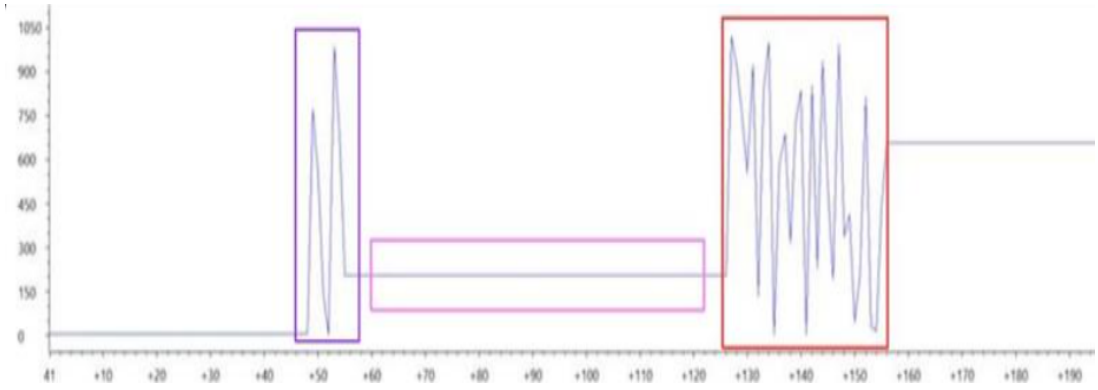

Figure 2: Execution of task4 that sends sine wave values to LTC 1661 DAC

- The above picture shows sine wave values being sent from the TIVA microcontroller to LTC 1661 DAC.
- Since the values are not viewed at the output of LTC1661, the interpolation is not accurate and hence we do not get a sine wave.
- But here, the main focus is on scheduling of the tasks and to view the results of the scheduler.
- **Purple Coloured Portion**: This part denotes task4 being executed with a low priority. We say that this is low priority as we can see only few spikes in the waveform.
- **Pink Coloured Portion**: This part indicates task4 being pre-empted by other tasks due to scheduler action. The flat line indicates that task4 is not being performed and CPU has been allocated to other processes using the scheduler.
- **Red Coloured Portion**: The red coloured part denotes task4 being implemented with a high priority (priority of task4 was changed by pressing the right most key of the last row in the EduArm board). Here since the number of spikes are more, we see that this task gets more CPU time when compared with how much it received when it had a low priority.

7.2 Dynamic Memory Allocation

- Every chunk of dynamically allocated memory has a*BLOCK_HEADER* associated with it. The contents of this *BLOCK_HEADER* structure are as follows:
  - Integer variable denoting the ID of current block of memory.
  - Void Pointer indicating the parent block of the current block of memory.
  - Void Pointer indicating the child block of the current block of memory.
  - Integer variable denoting size of the block. This denotes the amount of memory(in bytes) that is allocated to the threads.

7.2.1 Dynamic Memory Status After Intialization Step

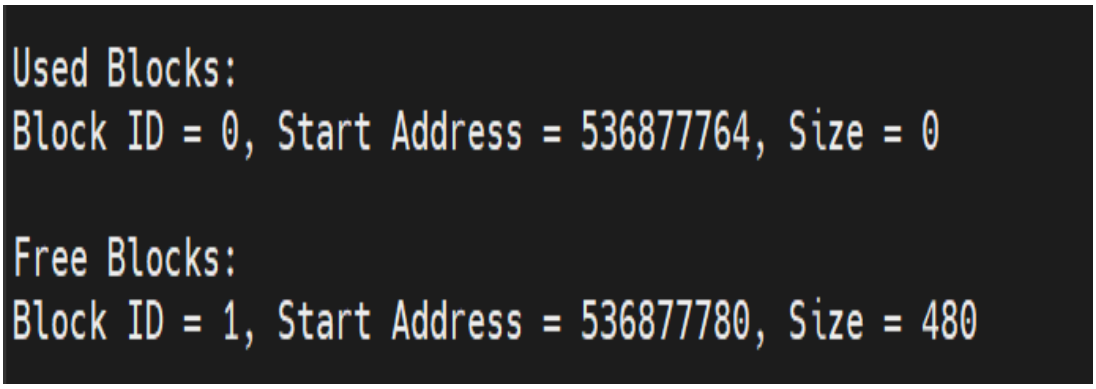

Figure 3: Dynamic Memory Usage After Initialization Step

- The above picture shows status of dynamic memory after the initialization step. In this example, **512** bytes of memory was allotted for dynamic memory usage in the initialization step. We always have two lists: *Used List* and *Free List*.
- **Used List**: This list indicates the portion of memory that has been allotted to the threads.
- **Free List**: This list indicates the portion of memory that is free and can be allotted to the threads.
- Here the start address of the heap memory section is *536877764* in decimal, which corresponds to *20001AC4* in hex. This address was obtained from __heap_start__ pointer present in the linker file.
- The used block starts at 536877764. In the used block section, there is a BLOCK_HEADER structure. The BLOCK_HEADER structure has a size of 16 bytes. Hence the next chunk of memory starts at 536877764 + 16 = 536877780 and this address is the starting address of the free list.
- In the free list, the size field indicates the amount of memory that is available for the threads to make use of. This size is computed as follows:

$$Size = (Total\ Requested\ Size) - (2 * (size\ of\ (BLOCK\_HEADER\ structure))) \tag{1}$$

$$Size = 512 - (2 * 16) = 512 - 32 = 480 \tag{2}$$

- This 480 is shown in the above image and this tells us that starting from address 536877764, 32 bytes are allotted to store BLOCK_HEADER structures at the beginning of Used and Free list. And the remaining 480 bytes of memory can be allotted to the threads as per the requests.

7.2.2 Dynamic Memory Usage After Memory Allotment

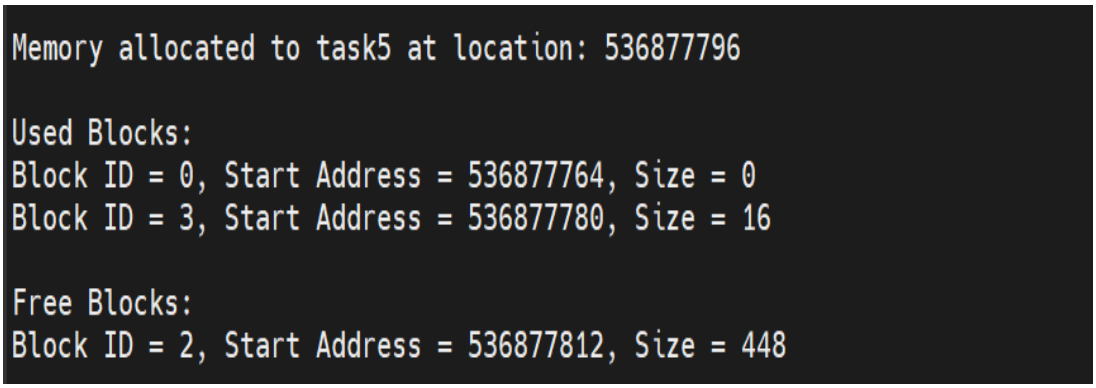

Figure 4: Dynamic Memory Usage After Memory Allotment Step

- In this scheduler, both task5 and task6 work on dynamically allocated memory. Both the tasks request for **16 bytes** of memory. Once allocated, they initialize the first value of that memory section with the integer value 0. And this value will be incremented. Once the value reaches 10, the allocated memory will be de-allocated and the whole process is repeated again.
- The above image shows us the status of Dynamic Memory usage after memory allotment for the first time. As mentioned in the above image, a portion of the dynamic memory has been allocated to task5 at location *536877796*.
- Now we see that there are two nodes in the Used List and one node in the Free list. This is because when task5 requested for 16 bytes of memory, a chunk of memory is taken from the free list and added to the used list . And head node of the free list is updated to the next available memory location that can be allotted to the threads.
- The block of memory starting at address *536877764*(in decimal) was previously present in the Free Block (refer Figure 3). Now once task5 requests for memory, that block is allocated to task5 and hence that block is moved to the used list as shown in the above image(refer Figure 4).
- Here, memory allocated to task5 is at location *535877796*. This denotes the address of the location that is actually given to the threads to perform computations. As mentioned earlier, every chunk of memory that is allocated carries a BLOCK_HEADER structure along with it. This structure can be thought of as a label to the allocated memory and this label contains information regarding ID of the memory, size of the memory, address of parent block of the memory and the address of the child block of the memory.
- Here the label of the memory allocated to the task5 starts at address *536877780* and the portion of memory that the task can actually use for computations is given by the following equation:

$$Size = Start\ Address\ of\ Label + Size\ of\ Label \tag{3}$$

$$Size = 536877780 + 16 = 536877796 \tag{4}$$

- This 536877796 is mentioned in the above image(refer Figure 4) as the starting address of the memory chunk allotted to task5 to perform actual computations.
- Similarly, the head node of the Free List is moved to the address denoting the next location that can be allotted to the threads. In this case it is, *536877812*.
- As we can see, size of the Free list was 480 bytes after initialization step(refer Figure 3). Once a portion of memory is allocated, size of the Free list is reduced to 448 bytes of memory.
- As the program keeps running, more sections of memory will be allocated and de-allocated to task5 and task6 and the sizes of used list and free list will keep growing and reducing dynamically .

VIII POSSIBLE IMPROVEMENTS IN THE SCHEDULER

- In this scheduler, all the tasks are assumed to be in of the following two states:
  - Running state
  - Ready state
- At any point of time, only one task is in running state and all the other tasks are in ready state. Currently *Waiting State* has not been implemented. This is always present in RTOS and this could be one of the possible improvements.
- This can be introduced in one of the following ways:
  - **Wait State Activation Based On Delay**: In this method, when a task is running we introduce some delay. Post the delay, the task voluntarily goes to waiting state. This is to simulate the condition of the task not getting a particular resource for it to execute.
  - **Wait State Through Task Synchronization**: In this method, we can have some code that is shared between multiple tasks. And once a particular task blocks a shared resource, the other task with which the resource is shared will go into Waiting state. For this, we need to use concepts of Mutex and Semaphore.

IX REFERENCE

- Embedded Systems: Real Time Operating Systems For Arm Cortex-M Microcontrollers by *Jonathan W. Valvano*

X ACKNOWLEDGEMENTS

- Thanks to the instructor of the course Prof.Haresh Dagale for his guidance over the course of the mini-project.
- Thanks to Tessin Jose, the author of dynamic memory allocation code that has been integrated with the scheduler code to allocate memory to tasks dynamically.