Lab3: Multitask Schedulers

Paria Naghavi (1441396)
Anna Mai (215101)
Assignment: Lab 3 July 29, 2023

Table of Contents

Introduction

The focus of the labs is on developing different scheduling techniques to manage and execute multiple tasks concurrently/ multitasking on a hardware platform. We also learned about flexibility of schedulers such as RR and SRRI and DDS for small tasks. Whether using DDS's TCB or array of function pointers for SRRI, we are using data structures to maintain scheduler functions. We used variables to flag states and count the tasks to provide synchronization, order and duration to produce the instructed outcome.

During Part 1 implementation, a RR scheduler was developed. In Part 2 ISR based scheduler was introduced. The Part 3 of the lab replaced the RR based scheduler with a data-driven approach. To implement Part 4, a 7-segment display task was added. Part 5 included a countdown timer. For completing Part 6, we added a songPlayCount variable to track the timing between the speaker and display. Throughout the lab, the flexibility of each scheduler, its associated functions and tasks were observed and expanded. The lab parts demos showcased the multitasking capabilities and the timing achieved by the different scheduling techniques.

Methods and Techniques

Besides the different types of schedulers discussed in the introduction, the lab involved managing tasks using arrays of function pointers, TCBs, and global flags. The setup() was updated with configurations required for new timer or circuit components such as the 7-segment LED. The hardware is similar to previous lab with the addition of 7-segment LED which was configured and manipulated using sevseg functions. They were used to control the display's LEDs, such as setNumber() and refreshDisplay() for updating timer values. Global flags were used as tools in both ISR and DDS. For example, in Part 5, a musicPlaying flag and downtimeStart variables were used to keep track of the state of the speaker.

Experimental Results

Part 1- RR: Task 1 & 2

In this section a basic RR scheduler was implemented. It executes two tasks: Flashing an external LED on for 250ms and off for 750ms, while playing the intro to the Super Mario Bros theme song, with 4 seconds of silence between each song cycle. The scheduler switches between these tasks in a cyclic manner. This is achieved by using the loop() to iterate through the tasks and running each task in sequential order. Since there is no task state management or sleep functionality, the scheduler operates on a simple first-come, first-served basis. Please see demo here.

Part 2- SRRI: Task 1 &2

To complete the requirements for this part an interrupt-based scheduler was implemented. It manages multiple tasks using a state-based approach. Tasks can be in one of the states: READY, RUNNING, or SLEEPING. Unlike Part 1, the SRRI scheduler manages multiple tasks with different states. The tasks are represented by an array of function pointers that point to the tasks to be executed.

We used 2 timers to implement this part. TIMER3 for the ISR and TIMER4 for PWM generation for controlling the speaker's frequency output. SRRI switches between tasks by iterating through the function pointer array and setting task states accordingly. Using TIMER3, the ISR is set to trigger an interrupt every 2 milliseconds. The schedule_sync() function handles task synchronization and sleep functionality. Tasks can put themselves to sleep using the sleep_474() function, which changes the task state to SLEEPING and for a given sleep duration.

During the implementation of ISR, we took ISR to testing by inserting an external LED in pin 48. The SRRI scheduler could run with an ISR that only updated the sflag. The SRI was expanded to be tested by using an external LED attached to pin 48. . Please see demo here.

Part 3- DDS: Task 1 & 2

The DDS schedulers offer more flexibility than RR. It allows tasks to terminate themselves or start other tasks-based conditionals. The scheduler for this part uses a data-driven approach to execute tasks using TCBs stored in arrays of type TCB structs. Each TCB contains information about a specific task, including its process ID, function pointer, state, and the number of times it has been restarted. An array of TCB structures called taskScheduler was used to manage the active tasks and their states. Each element of it represents a task. Tasks are added to the array and executed based on their states. There is also deadTaskList array that was used to store the terminated tasks. When a task terminates itself, it is moved from the taskScheduler to the deadTaskList.

The two new functions that we had to build to toggle the task state were task_self_quit()and task_start(). The first function allows a task to terminate itself and moves itself to the deadTaskList. On the other hand, the second function allows a task to start up another task by changing its status from DEAD to READY. Its parameter is a pointer to the element in the TCB array which is to be started. It moves it to taskScheduler, sets the state to READY. The task is removed from the deadTaskList, and its timesRestarted parameter is incremented by 1. The scheduler() function iterates through the taskScheduler, processing each task. It checks the task's state and executes its associated function, if it's not in the SLEEPING state. The main loop calls the scheduler function to manage the execution of tasks. Please see demo here.

Part 4-SRRI: 1, 2 & 3

This section implements SRRI that runs three tasks: flashing an LED, playing the Super Mario Bros. theme song, and updating a timer on the 7-segment display. The implementation for this is very similar to part 2. The scheduler structure and ISR stay the same. The display's configurations were added to the setup() function. Additionally, updateDisplay() function was defined to show a 100ms incremented timer. The taskScheduler TCB array 3rd element was initialized with updateDisplay() function.

The SevSeg library is utilized to drive the 7-segment display. It simplifies the process of interfacing with the seven-segment display, making it easier to configure and manipulate the state of the LEDs. Its configuration with digit and segment pins, and the SevSeg higher

abstraction functions such as sevseg.setNumber allow for easily manipulations of the display's LEDs. Please see domo here.

Part 5- DDS: Task 4

This part is a combination of task 1,2 and with count down timer and changes to updateDisplay(). While task 1 and 2 remain the same, a countdown timer is shown on the display during 4s pause between song cycles. Another requirement of this part is to have the updateDisplay()to show the frequency of the beats playing and the countdown timer while there was silence. Since the scheduler is a DDS, we can use the same TCB structures, the same as part 3. The taskScheduler array's parameters had to be initialized for the updateDisplay() function.

In order to start the countdown at the end of the song and not every time the beats[] array has zero frequency element, the songCycle() function was changed to update the value of downtimeStart. This variable stores the downtime period which music is not playing for 4s, and the 7-segment display will countdown until the next song is played. The downtimeStart was initialization with a value of -1000 to ensures that the first instance of downtime is processed properly. During the execution of the playSpeaker() function musicPlaying flag is set to true. Once it is paused, the musicPlaying flag is set to false, and the downtimeStart variable is updated with the current value of timerCounter. These two variables helped with keeping track of the start time of downtime periods between playing the theme song. Lastly, the updateDisplay() function was updated to display the frequency returned from songCycle(), while the music on and the countdown when silence. The total downtime period in deciseconds for when the scheduler waits between playing the song was shown on the 7-segment display. Please see dome here.

Part 6- DDS: Task 5

For this section songPlayCount variable swas declared to keep track of the melody cycles. It is used in the updateDisplay() function conditionals along the side of the status of Task 2. If Task 2 has not been restarted after two plays, it displays a countdown on the for 3 seconds. After the countdown, Task 2 is restarted to play the melody again. If Task 2 has been restarted and the melody has played more than twice, a smile pattern is displayed.

The scheduler manages task execution by looping through the taskScheduler array and executing tasks based on their states. The `task_start()` function allows tasks to be started or restarted by changing their status from DEAD to READY, and the `task_self_quit()` function allows tasks to terminate themselves by changing their state to DEAD and moving them to the deadTaskList. This allows to use the TCB arrays to turn tasks on and off.

Code Documentation

Part 1- RR: Task 1 & 2

The code for this part implements a round-robin scheduler to execute two tasks simultaneously. The scheduler runs continuously, executing two tasks without prioritizing any task. The two

tasks are represented by the functions flashExternalLED() and playSpeaker(). The former function (task 1) flashes an external LED periodically based on the timerCounter value, while the latter function (task 2) plays a sequence of musical notes by calculating the duration of each note using the noteIndex and timerCounter. During task 1, the LED toggles the pin based on the value of timerCounter, creating a periodic LED flash.

Meanwhile, the songCycles() goes through the melody[], playing notes for specific durations determined by beats and timerCounter. Task 2 has the songCycle() function that acts as a musical note sequencer. It returns the frequency of the next note to be played based on the time elapsed using timerCounter. The first time the function is called, it initializes the currentTime to the current value of timerCounter, and subsequent calls update the currentTime after playing each note. In conjunction with the playSpeaker() function, the frequency returned from songCycle() plays the notes using bit set and clear on output compare register for Timer 4. In order to achieve the 4s pause between each time the theme song is played, the last element of beats[] array was set to 80. This element corresponds to R in melody[], which results in frequency of zero, hence silence. The value of 80 corresponds to 4s, because 50 was used to obtain the noteDuration.

The RR scheduler is implemented in the void loop() by sequentially calling different tasks in a round-robin fashion, allowing each task to execute for a short time slice before switching to the next task. This approach ensures that all tasks get a fair share of CPU time and appear to run simultaneously, creating the illusion of multitasking.

Part 2- SRRI: Task 1 &2

The scheduler utilizes ISR that runs every 2ms, which is controlled by Timer 3. To test ISR, a LED connected to pin 48 is toggled every 100 ISR runs, serving as an interrupt indicator to observe the ISR performance.

In the main loop, the tasks are defined as function pointers and stored in an array called taskScheduler. The loop iterates through this array, executing each task in a RR fashion. The schedule_sync() function plays a synchronizes task scheduling and state transitions. It waits for the ISR to complete its execution, indicated by sFlag changing from PENDING to DONE. Then, it handles the necessary functions for state transitions, such as putting tasks to sleep or reactivating them for specific durations. The setup() function was updated to configure Timer 3 to generate an interrupt every 2ms. Timer 3's TCCR3A is set to 0 for normal mode operation, while TCCR3B is configured for CTC mode with a prescaler of 64. The OCR3A register value of 500 stores the value to be compared with the Timer3 count, and once the value of Timer3 and OCR3A match, an interrupt is generated. The amount of time between interrupts can be calculated by multiplying the prescaler value and OCR3A value and dividing it by the clock frequency that runs Timer3. Since the clock on the Arduino Mega runs at 16 MHz, (64*500)/16000000 = 0.002s, giving us the 2ms per interrupt we need.

Tasks 1 and 2 are executed using function pointers managed by the SRRI scheduler. The taskScheduler array holds these function pointers. The scheduler loops through this array,

executing each task whose state is not SLEEPING. Task 1, flashExternalLED(), toggles an the external LED on 48 based on the value of timerCounter, while Task 2, playSpeaker(), plays a sequence of musical notes based on the melody[] and beats[] arrays using Timer 4 for precise timing control. The ISR increments the isrCounter variable and toggles the interrupt indicator LED when the toggleThreshold is reached. It then sets sFlag to DONE, indicating that a task has been completed and the scheduler can proceed with the next task. The schedule_sync() function updates the state of the current task to READY after sFlag changes to DONE, ensuring proper task synchronization.

Part3: DDS: Tasks 1 & 2

The scheduler is implemented using an array of TCBs named taskScheduler to manage the tasks' states, function pointers, and other information. The code repeatedly runs the scheduler in the loop() function, managing the execution of the two tasks based on their states and functions. The setup() function was updated by initializing task 1 and 2 in taskScheduler. The two functions that change the state and update the TCB arrays of tasks are task_self_quit and task_start functions. When the first function is called it sets the current task's state to DEAD in the taskScheduler, indicating that the task has terminated. Then moves the terminated task from the taskScheduler to the deadTaskList array. Next it shifts down the remaining tasks in the taskScheduler to remove the terminated task from the array.

The scheduler() function is where the DDS is implemented. It iterates through the taskScheduler array and processes each task based on its state. The scheduler runs continuously in the loop() function. First, it checks the state of the current task, taskScheduler[currentTask].state. If the task's state is not SLEEPING, it means the task is either READY or RUNNING, and it proceeds to execute the task's associated function using a function pointer. Once the task starts executing, it is changed to RUNNING. Next, the currentTask index is incremented to move to the next task in the taskScheduler. The process continues until it reaches the end of the taskScheduler. In order to dynamically execute different tasks based on their function pointers, we use function_ptr(). It is a utility function, a wrapper for executing any given task. It takes a function pointer as a parameter and invokes the function pointed to by that pointer. It is called in the scheduler() function, when a task is ready to run.

Part 4-SRRI:1,2 & 3

The code for this part is very similar to part 2. The 7-segment LED configuration was added to setup function using sevseg() library. The taskScheduler's 3rd element was updated with updateDisplay() function. This function was defined using sevseg() functions to show a timer on the display. First, it calculates the elapsed time in deciseconds, using the timerCounter. Then, it sets the number to be displayed using sevseg.setNumber(deciSecondsElapsed, 0);, followed with an display update, using sevseg.refreshDisplay().

Part 5- DDS: Task 4

In this part, the scheduler is a data-driven and uses TCB structures similar to Part 3.

The countdown timer implementation uses downtimeStart variable and musicPlaying flag. The songCycle() function was updated to handle downtime and music playing. It now in addition to returning the frequency of the next note, it updates the downtimeStart variable and musicPlaying flag when when it reaches the last elements in beats[]. Using these two variables, the updateDisplay() function was modified to display the frequency returned from songCycle() when musicPlaying is set. When there's silence, it calculates the elapsed time in deciseconds since the downtime started and displays the countdown timer on the 7-segment display.

Part 6- DDS: Task 5

For this part, the songCycle() was changed to increments the songPlayCount every time the beats[] array reaches the last note. The updateDisplay() function has different behaviors based on the songPlayCount and the restart status of Task 2. If Task 2 has not been restarted after two plays, a countdown is displayed on the display for 3 seconds. After the countdown, Task 2 is restarted for one final time. If Task 2 has been restarted and the melody has played more than twice, a "smile" pattern is displayed on the 7-segment display for 2 seconds.

The songCycle() function checks if the melody has completed one full cycle. In other words, if noteIndex equals the length of the melody array. If so, it resets noteIndex to 0 to start playing the melody from the beginning, and it increments songPlayCount. If songPlayCount is greater than or equal to 2 (meaning the melody has played twice), the function exits.

The updateDisplay() function updates the content displayed based on the songPlayCount. If songPlayCount is equal to 2, it means Task 2 has played the melody twice, and now it's time to display the countdown on the display. The countdown starts at 3 seconds and decrements. Once the elapsed time is greater than 3 seconds, the function restarts Task 2 by calling task_start() and passing the deadTaskList entry for Task 2. This means Task 2 will play the melody two more cycles before it completely stops. When the songPlayCount is greater than 2, it means Task 2 is done, and the function will display a smile on the display for 2 seconds. After the smile duration is passed, the function displays nothing on the display and exits the task. The function task_self_quit() was called to terminate the tasks and set the states to DEAD.

Overall Performance Summary

Our demo videos hyperlinked throughout the report.

Teamwork Breakdown

This lab was completed in collaboration of Paria Naghavi and Anna Mai.

Discussion and Conclusions

During the implementation of the lab the scheduler structure maintained almost unchanged with only updating their arrays when adding new tasks. There was also flexibility in their structure as the conditionals grew in each task for desired output.
The sleep_474() function did not generate the desired output in part 2 and 4. We were able to produce the 4s pause between each iteration of the music, but without using sleep_474()

function. The ISR for part 2 and 4 was tested using an external LED in pin 48. This allowed us to visually observe the performance of the ISR. We kept it in the code in case you would like to see it.

Reference

1. ChatGPT
2. C programming language documentations: https://en.cppreference.com/w/c/preprocessor/conditional
3. Lab provided resources.
4. Sev_seg library documentations