Embedded System Design with MCU and FPGA

LAB06 0858706 黃品嫚

Goal

Let us know more about Multitasking and FreeRTOS.

Problems

1. What are the two types of multitasking?

There are two types of multitasking, cooperative multitasking, which also known as non-preemptive multitasking, and preemptive multitasking. **Non-preemptive multitasking** means that once resources are allocated to a process, the process holds it until it completes its burst time or switches to the waiting state. **Preemptive multitasking** differs from non-preemptive multitasking in that the operating system can take control of the processor without the task's cooperation and the resources are allocated to a process for a limited time. [1]

And there are some other differences between this two kind multitasking: [2]

Comparison	Preemptive Multitasking	Non-preemptive Multitasking
CPU	The CPU is allocated to the processes for the limited time.	The CPU is allocated to the process till it terminates or switches to waiting state.
Interrupt	Process can be interrupted in between.	Process can not be interrupted till it terminates or switches to waiting state.
Starvation	If a high priority process frequently arrives in the ready queue, low priority process may starve.	If a process with long burst time is running CPU, then another process with less CPU burst time may starve.
Overhead	Preemptive scheduling has overheads of scheduling the processes.	Non-preemptive scheduling does not have overheads.
Flexibility	Preemptive scheduling is flexible.	Non-preemptive scheduling is rigid.

2. When scheduling via time stamps, how to get the number of milliseconds passed since the Arduino board began running the current program? How long will it be overflow?

There are two functions that we could call, the first one is `millis()` and another one is `micros()`. [3][4]

Syntax:

```
time = millis()
time = micros()
```

The `millis()` will return the number of milliseconds passed since the Arduino board began running the current program, and the `micros()` will return the number of microseconds.

Both return data types of `millis()` and `micros()` are unsigned long, which ranges from 0 to 4,294,967,295. [5] So we could know that the return number of **`millis()`** will overflow, after

approximately 50 days. Because 50 days is 4,320,000 seconds, `millis()` return number will be 4,320,000,000 which is larger than 4,294,967,295. And the return number of **`micros()`** will overflow, after **approximately 70 minutes**. In `micros()`, there are 1,000 microseconds in a millisecond and 1,000,000 microseconds in a second.

3. Explain the following c code:

   int *ptr = new int[10];

   Let's separate to two parts, `int *ptr;` and `ptr = new int[10];`, and discuss.
   The first part, `int *ptr;`, means dynamic allocating memory and `ptr` will point to the memory address. The second part, `ptr = new int[10];`, means the size of memory is an `int` array.
   Combine these two part, we could know it means allocating a block (an array) of elements of type `int`. The system dynamically allocates space for 10 elements of type int and returns a pointer to the first element of the sequence, which is assigned to `ptr` (a pointer). [6]

4. What cause the fragmented heap?

   The heap is the area of the RAM where the dynamic memory allocation happens. Every time we call `malloc()`, we will reserve a block of memory in the heap. Similarly, we call `new` or `String`, we will also reserve a block of memory in the heap. Because `new` calls `malloc()`. When we call `free()` to release a block, we create a hole of unused memory. After some time, the heap will become a cheese with many holes.
   Here is an example of when will the fragmentation happen. At first, suppose we just released a block of memory and therefore created a hole in the heap. There are three situations we will face. First, we allocate another block of the same size. The new block takes the place left by the old one. No hole remains. Second, we allocate a smaller block. The new block fits in the hole but doesn't fill it. A small hole remains. Last, we allocate a larger block. The new block cannot fit in the hole, so it's allocated further in the heap. A big hole remains. [7]

5. Would you use FreeRTOS in your multitask projects? Explain your reason.

   I would like to use FreeRTOS in my multitask projects. Although we could use other methods, like carefully designed procedure, scheduling via system timer, and using ISR, these methods still exist limitations for more complex applications. Such as using ISR, the ISR functions may become complex and require long execution times. Using system timer (timestamps), when a system requires several different cycle times, it is hard to implement. So, that's why I will prefer to use FreeRTOS in my multitask projects.
   There are many advantages to using FreeRTOS. In the library, we could easily implement the multitask, and it allows us to focus on application development rather than resource management. Each task is allocated a defined stack space, enabling predictable memory usage. We also could manage the sharing of data, memory, and hardware resources among multiple tasks. [11]

6. Compare the difference between delay() and vTaskDelay()

Using delay() will take up CPU computing resources, that is, the task is running, but did not do anything. The task occupies resources just to wait for the event, so it is better to use vTaskDelay(), which is the function of the FreeRTOS library. Using the vTaskDelay(), the task will enter the blocked state and the CPU resources will be released. So, many tasks could run simultaneously. [8]

7. In TaskAnalogRead(), is the variable sensorValue allocated in the heap area or allocated in the stack area?

Let's compare stack and heap first: [9][10]

Stack

I. It stores temporary variables created by each function. (LIFO)
II. The allocating memory is continuous.
III. The allocation happens in the function call stack.
IV. The allocation and deallocation are automatically done.
V. It is not flexible, the memory size allotted cannot be changed.
VI. Space is managed efficiently by CPU, and memory will not become fragmented.

Heap

I. It stores dynamically allocated variables because these variables will not be created when the code is compiling.
II. The allocating memory is not continuous.
III. The memory is allocated during the execution of instructions written by programmers.
IV. The allocation and deallocation need to be done by the programmer manually.
V. It is flexible, and the allotted memory can be altered.
VI. No guaranteed efficient using of space, the memory may become fragmented over time as blocks of memory are allocated, and then freed.

The variable sensorValue will be allocated in the **stack** area because it is not a dynamic allocation variable, it will be allocated when the code is compiled.

Reference

[1] https://sqljunkieshare.com/2012/01/06/preemptive-vs-non-preemptive-and-multitasking-vs-multithreading/
[2] https://techdifferences.com/difference-between-preemptive-and-non-preemptive-scheduling-in-os.html
[3] https://www.arduino.cc/reference/en/language/functions/time/micros/
[4] https://www.arduino.cc/reference/en/language/functions/time/millis/
[5] https://www.arduino.cc/reference/en/language/variables/data-types/unsignedlong/#targetText=Unsigned%20long%20variables%20are%20extended,2%5E32%20%2D%201).
[6] http://www.cplusplus.com/doc/tutorial/dynamic/
[7] https://www.youtube.com/watch?v=_G4HJDQjeP8
[8] http://wiki.csie.ncku.edu.tw/embedded/freertos
[9] https://www.geeksforgeeks.org/stack-vs-heap-memory-allocation/
[10] https://www.gribblelab.org/CBootCamp/7_Memory_Stack_vs_Heap.html

[11] http://www.keil.com/rl-arm/rtx_rtosadv.asp