GPOS Task Scheduling

-Aman Kanwar

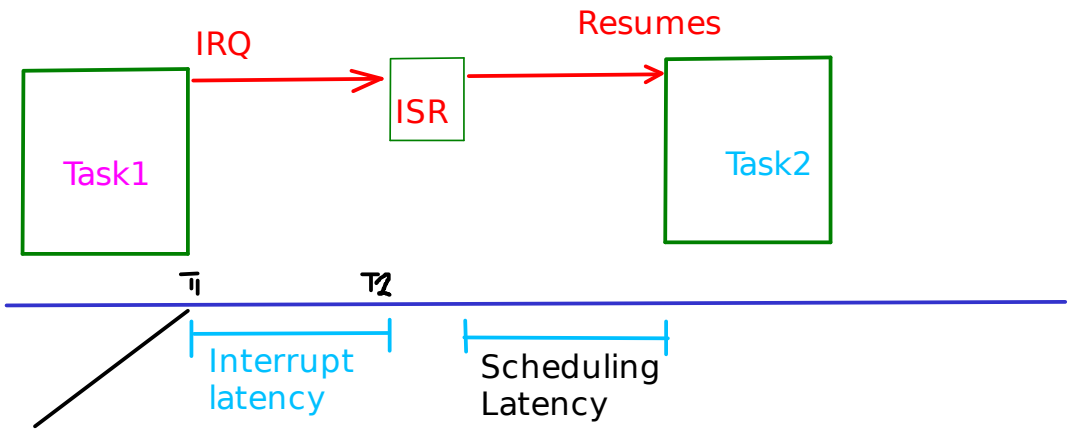
GPOS is programmed to handle schedling in such a way that it manages to achieve high thorughput.

Here the focus is mainly on the computation power and the services that needs to be provided to the user.

Mainly this is more focused on the user experience.

By definition, throughput here means the number of process that complete their execution for a given unit of time.

The scheduling implementation of GPOS is done in such a way that the high priority task will be given the CPU cycle in case there it is ready to run, regardless of any given low priority task which are available to RUN on the CPU.



Let's say that the given interrupt came at time T1 and the handler for the interrupt gets executed at the time T2 in that case the interrupt latency will be the difference in these two time stamps.

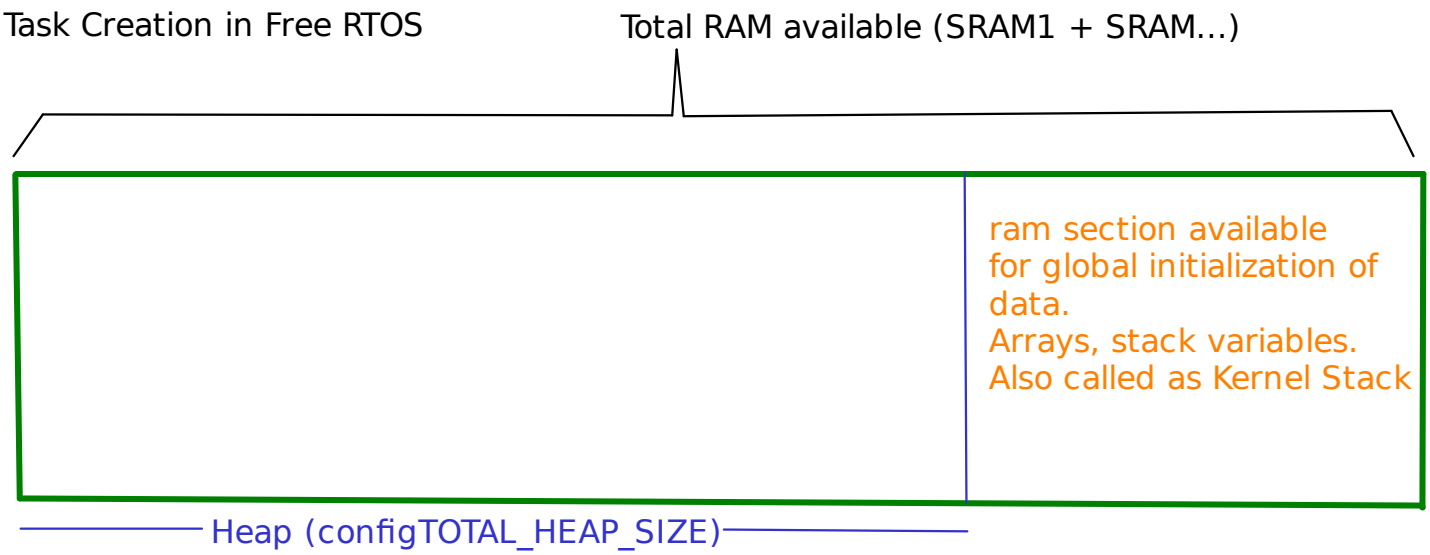
In GPOS this latency will increase as the CPU load increases.

When the ISR finshes, the preemption will be occour. In case there is no other task available to run in that case the same task will be running.

Here the latency that will come-up with this preemption is called as scheduling latency.

In case of RTOS this interrupt latency is bounded. However with GPOS the interrupt latency is not bounded. Hence, the Scheduling latency in case of GPOS is not constant.

Hence, with increased CPU load the latency will be constant in case of RTOS.
In case of GPOS the scheduling latency will incrase with increasing CPU load.



Here the size of HEAP is managed by the shown macro which is availabe in FreeRTOSconfig.h file.

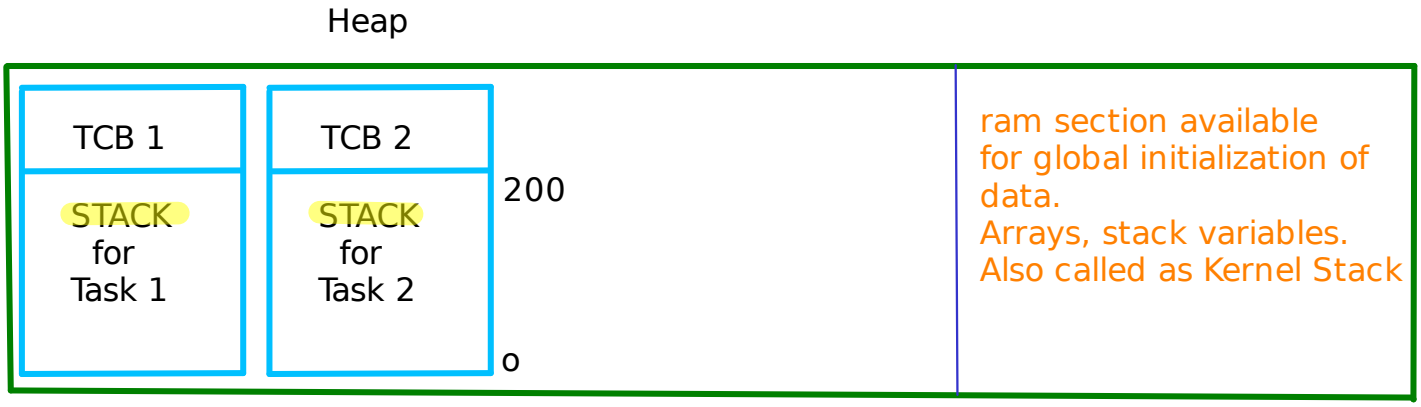
Check the value of this macro in the mentioned file.

Try to increase the size and build the project again. At a given point it may overflow.

When xTaskCreate is called.

```
BaseType_t xTaskCreate(    TaskFunction_t pvTaskCode,
                           const char * const pcName,
                           configSTACK_DEPTH_TYPE usStackDepth,
                           void *pvParameters,
                           UBaseType_t uxPriority,
                           TaskHandle_t *pxCreatedTask
                       );
```

The TCB for the created task will be allocated space in the HEAP section and a stack space for the task will also be reserved in the heap section.



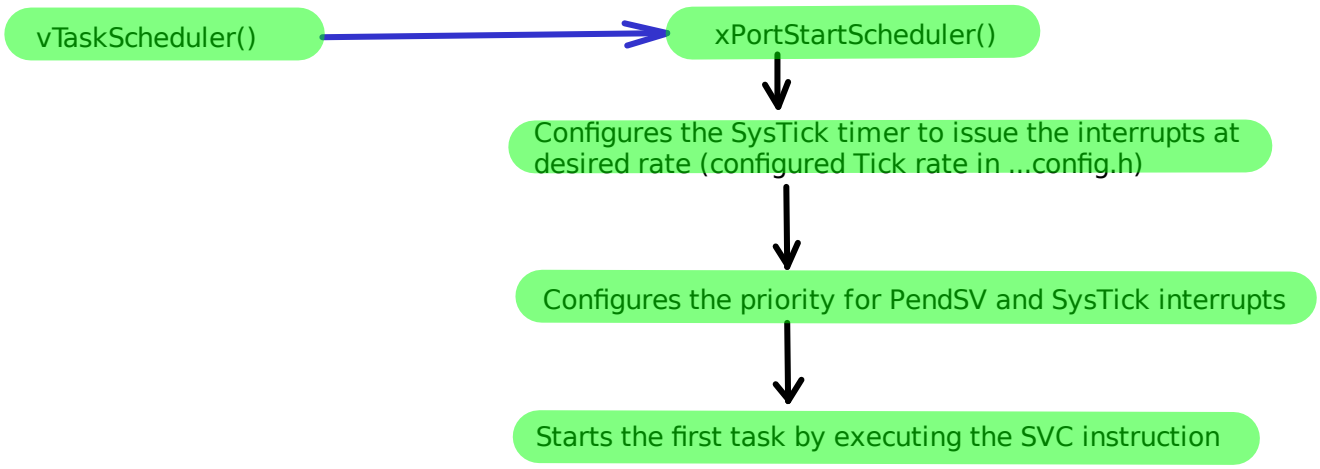

```
BaseType_t xTaskCreate(    TaskFunction_t pvTaskCode,
                           const char * const pcName,
                           configSTACK_DEPTH_TYPE usStackDepth,
                           void *pvParameters,
                           UBaseType_t uxPriority,
                           TaskHandle_t *pxCreatedTask
                       );
```

Here the size of created stack will be determined based on the size that we have provided in the API call.

vTaskStartScheduler() -Aman Kanwar

After this function is called, it will not return and the task will start getting scheduled.
Idle task is created by the scheduler which will do the clean-up later on this is very important.

This is a generic Implementation across for all architectures.



If we look at the declaration of the vTaskScheduler(), it is calling xPortStartScheduler(), where it does the initialization priority of SysTick timer using "vPortSetupTimerInterrupt()" and it sets the priority using CMSIS exported interfaces. and priority of the SysTick and PendSV is set as lowest.

After doing all this, it calls the very first task using prvPortStartFirstTask();

If we look the declaration of prvPortStartFirstTask(), it is an inline assembly code that is calling the SVC instruction in order to launch the very first task. IMPORTANT.

After this the SVC handler will be getting executed.

With SVC instruction, SVC handler named vPortSVCHandler(void) will get executed which is defined in port.c this is also an inline assembly code.

> Here in SVC it will be restoring the context from the Task TCB. This will take care of executing the very first task after that we have one more handler called.

PendSV handler, which is used for Context Switching. This handler is very big and contains inline assembly code.

Then we have a SysTick handler which runs for every 1 ms via SysTick interrupt and this actually triggers the PendSV handler.