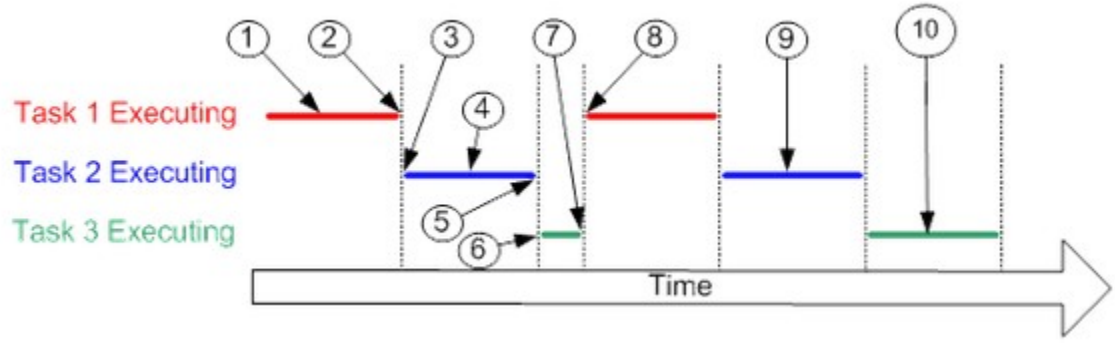
RTOS Scheduler



1. Task 1 is executing
2. The time slice for task 1 ends and the scheduler routine is called.
3. The scheduler selects task 2.
4. Task 2 executes for some time and takes a lock on a device.
5. Task 3 is now scheduled.
6. It runs and since it needs the same lock, it is put in wait queue
7. Again scheduler is called, and it selects task 1
8. Task 1 executes for its time slice
9. Now task 2 executes and once it is done, it releases the lock
10. Now task 3 runs

The problem with real time scheduling is that, if a task is not scheduled at the correct time, then it will fail. Thus, the keyword here is timely scheduling of processes.
One method for achieving this is to assign a **priority** to each task. The kernel then uses this priority to determine a task's place within the sequence of execution of other tasks. In a prioritized system the **highest priority task that is ready is given control of the processor**. Tasks of lower priority must wait and, even when running, may have their execution preempted by a task of higher priority. When a task completes its operation or blocks (waits for something to happen before it can continue) it releases the CPU. The scheduler in the RTOS then looks for the next highest priority code entity that is ready to run and assigns it control of the CPU.

Most RTOSes use a **preemptive priority based** scheduler (**highest priority ready task runs until it terminates, yields, or is preempted by a higher priority task or interrupt**). Some also schedule round-robin for tasks at the same priority level (task runs until it terminates, yields or consumes its time-slice and other tasks of the same priority are ready to run).
The difference with schedulers in the desktop machines is that processes are scheduled based on fairness. But in RTOS fairness is not important.

One common problem encounters with conventional preemptive based RTOS schedulers is that of **priority inversion.** In this situation a low (medium) priority task obtains access to a shared resource even if a high priority task is pending. If a critical task requires that resource, it cannot run until the low priority task releases the mutex. One way to eliminate this is the priority inheritance protocol. It prevents a low-priority task from being elevated to high priority task.

We also must see, a **low priority task simply does not hog the resource** and keep running indefinitely in a state in which it cannot be easily preempted.
In an RTOS, a third party code cannot be allowed to run.

Key factors in a real-time OS are **minimal interrupt latency and minimal thread switching latency**; a real-time OS is valued more for how quickly or how predictably it can respond than for the amount of work it can perform in a given period of time.

In priority inversion a high priority task waits because a low priority task has a semaphore, but the lower priority task is not given CPU time to finish its work.