Change nice value

- renice & nice commands are used to modify nice value of existing process or new process.
- To increase the nice value, root permission is not needed; but to decrease the decrease the nice value, root permission is required.
- nice command internally call nice() syscall.

```
nice -n 12 ./sched.out
sudo nice -n -3 ./sched.out
sudo renice -n 18 -p <pid>
```

- The non-realtime priorities (i.e. nice values) are always considered as lower priorities than real time priorities.
- Inside kernel, nice values (-20 to +19) are mapped to 100 to 139, which is after realtime priorities 0 to 99.
- The CPU time is alloted to any process depending on its nice value.
    - More the nice value, lesser the CPU time alloted.
    - Lower the nice value, more the CPU time alloted.

RTOS Norms (POSIX)

- Multi-tasking
    - Tasks must be "light-weight".
- Priority scheduling
    - Each task must have priority.
- Pre-emptive
    - High priority task should preempt low priority task (immediately).
    - High priority task should never wait for/due to low priority task.
- Inter-task communication
    - Deterministic latency
- Deterministic memory allocation
- Deterministic interrupt latency
- Deterministic scheduling latency
- Deterministic scheduler duration
- Must handle problem of "priority inversion".

Latencies

- Interrupt latency
    - Time from arrival of interrupt to begin execution of ISR.
        - ARM 7: 24-30 CPU cycles (can vary due to implementation of IRQ_Handler).
        - ARM CM3: 12 CPU cycles (no software IRQ handler)
    - It may delay due to interrupt masking.
- Scheduler latency
    - Time from completion of ISR to begin execution of scheduler.

    - Depends on house-keeping work done by system after ISR call.
- Memory latency
    - Time required for allocating memory block.
    - Memory allocation must be done in fixed time.
- ITC latency
    - Inter-task communication must be completed in fixed time.
    - There can be priority for the messages.

Kernel Response Time

- Time to handle the interrupt, schedule the next process and begin its execution.
    - Interrupt arrived --> Interrupt handler --> ISR call --> ISR execution --> house-keeping -->
    - Scheduler called --> Decide the next process to execute --> Dispatch/restore context of new process.
- Kernel Response Time = Interrupt Latency + ISR Duration + Scheduler Latency + Scheduler Duration
    - Interrupt Latency = Interrupt arrived --> Interrupt handler --> ISR call
    - ISR Duration = ISR execution
    - Scheduler Latency = house-keeping --> Scheduler called
    - Scheduler Duration = Decide the next process to execute

Pre-emptible kernel and Non-preemptible kernel

- Pre-emptible kernel
    - User space and kernel space preemption of process is supported.
    - Better responsiveness
    - Real time systems
- Non Pre-emptible kernel
    - User space preemption of process is supported.
    - Better throughput
    - Server systems and heavy computations

RTOS types

- RTOS can be classified as follows:
    - Embedded RTOS
    - Real-time Linux
    - Linux based RTOS -- Dual kernel approach

Real-time Linux

- Linux kernel can be configured with RT settings E.g. HZ=1000, Kernel=Pre-emptible Kernel, etc.
- Linux kernel source code can be modified heavily to provide realtime behaviour. E.g. scheduler = O(1), interrupt handling mechanism updated for minimal latency.
- "RTPatch" is an open-source project (rtpatch.org) by Ingo Monlar that converts Linux into RTOS.

Dual kernel approach

- Linux kernel is treated as lowest priority task by the "real time executive/kernel". Hence when no real time tasks are running, Linux kernel is scheduled. Then Linux scheduler share the available CPU time

with Linux processes.

- Hardware interrupts in this approach are controlled/managed by the real time executive. They will be dispatched to the Linux kernel only if not handled by the RT tasks. The RT kernel act as "virtual PIC" for the Linux kernel. When Linux kernel try to disable/mask the interrupt, RT kernel will not send interrupts to the Linux kernel (Note that hardware interrupts cannot be manipulated by Linux kernel directly).
- The RT executive mainly contains RT task management, RT scheduler and RT IPC mechanisms. The rest of the functionality i.e. memory management, file io, networking, etc will be handled by the Linux kernel. Due to this size of RT kernel is too small & it also referred as "nano-kernel".
- E.g. RTAI (Real-time Application Interface), Xenomai, etc.

Embedded RTOS

- Also known as Traditional RTOS or Conventional RTOS.
- RTOS is embedded into embedded controller device along with real-time application/tasks.
    - RTOS kernel + RT Tasks ---> .hex or .bin ---> uC Flash
- ARM application development
    - Source code (C/Asm) --> Cross Compile --> Executable (ELF) --> Hex/Bin file --> ARM uC Flash
    - Application have full control of Hardware peripherals, Timers, Interrupts, etc.
    - Difficult to manage (modularize, debug, ...) in complex applications (multiple peripherals).
- ARM + Embdedded RTOS application development
    - Source code + RTOS code (C/Asm) --> Cross Compile --> Executable (ELF) --> Hex/Bin file --> ARM uC Flash
    - RTOS reserve some system peripherals e.g. Timer, Interrupt controller, Software Interrupt, etc.
    - Application code can be divided into RT tasks (e.g. LED task, ADC task, UART task, etc). These task may communicate with each other.
    - E.g. FreeRTOS, uC-OS, etc