chrt :

<https://linux-tips.com/t/how-to-use-chrt-command/268>

You can use chrt command to set or retrieve the real time scheduling attributes-scheduling priority of an existing pid. You can also run command with the given attributes.

To get / retrieve the real-time attributes of an existing task / PID, enter:

$ chrt -p pid

renice :

renice changes only the priority of a process :

renice 3 -p <pid> : set priority of this process to 3

chrt can change priority and scheduling algorithm of a process :

sudo chrt -f -p 2 <pid> : set sched\_fifo (-f) with priority 2.

chrt -p <pid> : show the current scheduling algorithm and its priority.

Taskset :

set or retrieve a process's CPU affinity

to retrieve : taskset -p pid

to set : sudo taskset -pc (choose from a number depending on how many cpus you have) pid.

Exemple : sudo taskset -pc 5 pid (means , set cpu 5 to this process id).

**taskset** is used to set or retrieve the CPU affinity of a running process given its PID or to launch a new COMMAND with a given CPU affinity. CPU affinity is a scheduler property that "bonds" a process to a given set of CPUs on the system. The Linux scheduler will honor the given CPU affinity and the process will not run on any other CPUs. Note that the Linux scheduler also supports natural CPU affinity: the scheduler attempts to keep processes on the same CPU as long as practical for performance reasons. Therefore, forcing a specific CPU affinity is useful only in certain applications.

The CPU affinity is represented as a bitmask, with the lowest order bit corresponding to the first logical CPU and the highest order bit corresponding to the last logical CPU. Not all CPUs may exist on a given system but a mask may specify more CPUs than are present. A retrieved mask will reflect only the bits that correspond to CPUs physically on the system. If an invalid mask is given (i.e., one that corresponds to no valid CPUs on the current system) an error is returned. The masks are typically given in hexadecimal. For example,

**0x00000001**

is processor #0

**0x00000003**

is processors #0 and #1

**0xFFFFFFFF**

is all processors (#0 through #31)

When **taskset** returns, it is guaranteed that the given program has been scheduled to a legal CPU.

**SCHED\_FIFO :**

**SCHED\_FIFO** can be used only with static priorities higher than 0,

which means that when a **SCHED\_FIFO** thread becomes runnable, it will

always immediately preempt any currently running **SCHED\_OTHER**,

**SCHED\_BATCH**, or **SCHED\_IDLE** thread. **SCHED\_FIFO** is a simple scheduling

algorithm without time slicing. For threads scheduled under the

**SCHED\_FIFO** policy, the following rules apply:

1) A running **SCHED\_FIFO** thread that has been preempted by another

thread of higher priority will stay at the head of the list for

its priority and will resume execution as soon as all threads of

higher priority are blocked again.

2) When a blocked **SCHED\_FIFO** thread becomes runnable, it will be

inserted at the end of the list for its priority.

3) If a call to [sched\_setscheduler(2)](https://man7.org/linux/man-pages/man2/sched_setscheduler.2.html), [sched\_setparam(2)](https://man7.org/linux/man-pages/man2/sched_setparam.2.html),

[sched\_setattr(2)](https://man7.org/linux/man-pages/man2/sched_setattr.2.html), [pthread\_setschedparam(3)](https://man7.org/linux/man-pages/man3/pthread_setschedparam.3.html), or

[pthread\_setschedprio(3)](https://man7.org/linux/man-pages/man3/pthread_setschedprio.3.html) changes the priority of the running or

runnable **SCHED\_FIFO** thread identified by *pid* the effect on the

thread's position in the list depends on the direction of the

change to threads priority:

· If the thread's priority is raised, it is placed at the end of

the list for its new priority. As a consequence, it may

preempt a currently running thread with the same priority.

· If the thread's priority is unchanged, its position in the run

list is unchanged.

· If the thread's priority is lowered, it is placed at the front

of the list for its new priority.

According to POSIX.1-2008, changes to a thread's priority (or

policy) using any mechanism other than [pthread\_setschedprio(3)](https://man7.org/linux/man-pages/man3/pthread_setschedprio.3.html)

should result in the thread being placed at the end of the list

for its priority.

4) A thread calling [sched\_yield(2)](https://man7.org/linux/man-pages/man2/sched_yield.2.html) will be put at the end of the

list.

No other events will move a thread scheduled under the **SCHED\_FIFO**

policy in the wait list of runnable threads with equal static

priority.

A **SCHED\_FIFO** thread runs until either it is blocked by an I/O

request, it is preempted by a higher priority thread, or it calls

[sched\_yield(2)](https://man7.org/linux/man-pages/man2/sched_yield.2.html).

**SCHED\_RR :**



IMPORTANT TO KNOW :

SCHED\_FIFO and SCHED\_RR are so called "real-time" policies. They implement the fixed-priority real-time scheduling specified by the POSIX standard. Tasks with these policies preempt every other task, which can thus easily go into starvation (if they don't release the CPU).

The difference between SCHED\_FIFO and SCHED\_RR is that among tasks with the same priority, SCHED\_RR performs a round-robin with a certain timeslice; SCHED\_FIFO, instead, needs the task to explicitly yield the processor.

**SCHED\_OTHER :**

SCHED\_OTHER is the common round-robin time-sharing scheduling policy that schedules a task for a certain timeslice depending on the other tasks running in the system.

**SCHED\_OTHER** can be used at only static priority 0 (i.e., threads

under real-time policies always have priority over **SCHED\_OTHER** pro‐

cesses). **SCHED\_OTHER** is the standard Linux time-sharing scheduler

that is intended for all threads that do not require the special

real-time mechanisms.

The thread to run is chosen from the static priority 0 list based on

a *dynamic* priority that is determined only inside this list. The

dynamic priority is based on the nice value (see below) and is

increased for each time quantum the thread is ready to run, but

denied to run by the scheduler. This ensures fair progress among all

**SCHED\_OTHER** threads.

In the Linux kernel source code, the **SCHED\_OTHER** policy is actually

named **SCHED\_NORMAL**.

**SCHED\_IDLE :**

**SCHED\_IDLE: Scheduling very low priority jobs**

(Since Linux 2.6.23.) **SCHED\_IDLE** can be used only at static priority

0; the process nice value has no influence for this policy.

This policy is intended for running jobs at extremely low priority

(lower even than a +19 nice value with the **SCHED\_OTHER** or **SCHED\_BATCH**

policies).

**SCHED\_DEADLINE:**

**Sched\_deadline is an algorithm based of Earliest Deadline First (EDF) scheduling.**

**We have a unit to execute, deadline and period.**

**A process P needs to execute x unit in a period of t and not exceed the deadline.**

each task scheduled under such policy is associated with a *budget* Q (aka *runtime*), and a *period* P, corresponding to a declaration to the kernel that Q time units are required by that task every P time units, on any processor. This makes SCHED\_DEADLINE particularly suitable for [real-time](https://en.wikipedia.org/wiki/Real-time_computing) applications, like multimedia or industrial control, where P corresponds to the minimum time elapsing between subsequent activations of the task, and Q corresponds to the worst-case execution time needed by each activation of the task.

My understanding :

CPU Scheduling in Linux operates with Scheduling algorithms.

There are two type :

Real time scheduling and time sharing scheduling.

For example : Sched\_fifo and sched\_rr are realtime and preemptive while other algorithms are time sharing and non preemptive.

Those algorithms are using calculations to keep the CPU busy and give him the priority and how to complete different tasks. Each algorithm has its own way to work but in a general manner,

The real time algorithms will give a task to execute to the CPU and wont let the task go until it finishes while time sharing (depending on the algorithm) will pass to the CPU a task but can pause the process to execute another process and re integrate the older task to finish the execution and this is the principle behind time sharing.