Operating Systems

Lecture 3.5

Process Scheduling Algorithms - II

Instructor: Muhammad Arif Butt, PhD

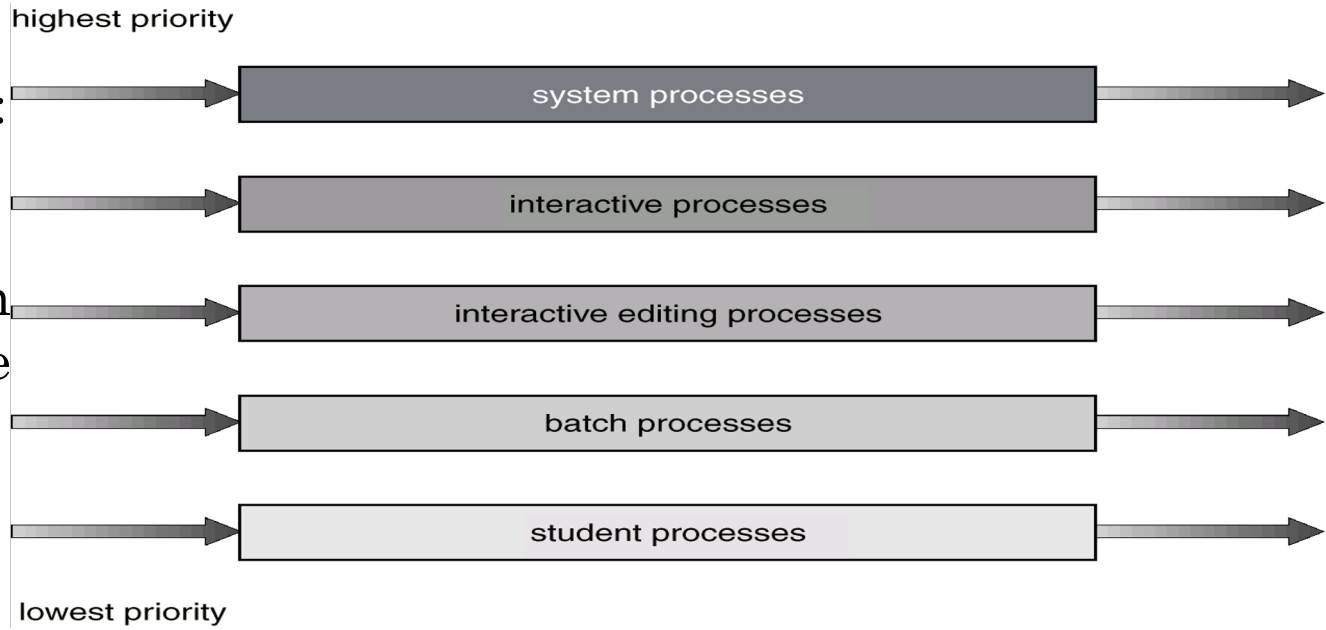
Lecture Agenda

- Multilevel Queue Scheduling
- Multilevel Feedback Queue Scheduling
- Rotating Staircase Deadline Schedular
- Unix SVR3 Scheduling Algorithm
- Linux CFS Scheduler
- Scheduling related shell commands
- Process Scheduling Info in `/proc/[PID]/`
- Scheduling related system calls

Multilevel Queue Scheduling

Multilevel Queue Scheduling

- A multilevel queue scheduling algorithm partitions the Ready Queue into several separate queues:
  - Foreground (interactive)
  - Background (batch)
- Processes are permanently assigned to a queue on entry to the system, based on some property of the process, e.g. memory size, process priority, process type.
- Processes do not move between queues.
- Each queue has its own scheduling algorithm:
  - Foreground – RR
  - Background – FCFS
- More over there must be scheduling between the queues, e.g. foreground queues may have absolute priority over back ground queues.



MLQ Scheduling - Example 14

Draw the graph (Gantt chart) for the following processes using MQ Scheduling algorithm.

If (CPU time < 4) **then** Q1 (FCFS)
else **if** (4 <= CPU time < 7) **then** Q2 (FCFS)
else Q3 (FCFS)

Process	Arrival Time	Burst Time
p1	0	8
p2	2	2
p3	3	4
p4	5	6
p5	7	9
p6	9	3
p7	10	5

MLQ Scheduling - Example 15

Draw the graph (Gantt chart) for the following processes using MQ Scheduling algorithm.

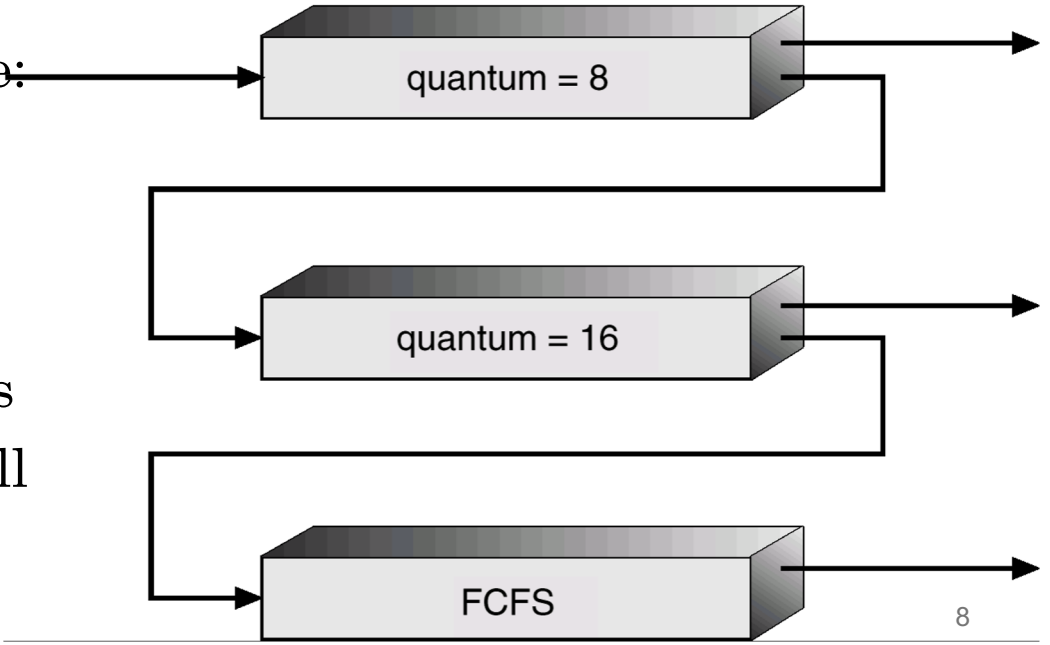
If (CPU time < 4) **then** Q1 (FCFS)
else **if** (4 <= CPU time < 7) **then** Q2 (RR – 3sec)
else Q3 (FCFS)

Process	Arrival Time	Burst Time
p1	0	8
p2	2	2
p3	3	4
p4	5	6
p5	7	9
p6	9	3
p7	10	5

Multilevel Feedback Queue Scheduling

Multilevel Feedback Queue Scheduling

- In MLFQ, processes can move up or down between queues based on their behaviour over time. This is what distinguishes MLFQ from static multilevel queue scheduling.
- Processes that uses too much CPU time is moved to a lower priority queue thus leaving the interactive and I/O bound processes in the higher priority queue. This ensures that interactive and I/O-bound processes, which often block and yield the CPU early, stay in higher-priority queues, improving responsiveness.
- Starvation can occur when lower-priority processes are never scheduled because higher-priority queues are always full. Aging is a common solution: processes that wait too long are gradually promoted to higher queues.
- Parameters of a Multilevel-feedback-queue scheduler are:
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine when to demote a process
  - Method used to determine when to upgrade a process
  - Method used to determine which queue a process will enter when that process needs service



MLFQ Scheduling - Example 16

Queue Setup is given in the opposite table and the rules are given below:

- All processes enter Q1 first.
- If a process does not finish in its quantum, it is moved to the next lower queue.
- If a process waits more than 10 seconds in Q3 without execution, it is promoted to Q2 (aging).
- All processes arrive at time 0.

Queue	Priority	Scheduling	Time Quantum	Behaviour
Q1	High	Round Robin	4 sec	All processes starts here
Q2	Medium	Round Robin	6 sec	For CPU-intensive jobs
Q3	Low	FCFS	–	Starvation-prone jobs

Given the Process table having its arrival time and CPU burst. Draw the Gantt Chart that shows the process completion time.

Process	Arrival Time	CPU Burst
P1	0	6
P2	0	9
P3	0	3
P4	0	12
P5	0	2

Rotating Staircase Deadline Scheduler

Rotating Staircase Deadline Scheduler

RSDL is a scheduling algorithm that employs **multi-level queues** arranged like a staircase. Each priority level has its own execution quota, and each process has a personal quota at its current level

How It Works:

- **Arrival & Placement:** Processes are placed in the queue corresponding to their priority level.
- **Execution:** Scheduler cycles through processes at the current highest non-empty level using RR.
- **Per-Process Quota:** Each process at that level gets a fixed CPU time slice (its quota). When it runs out, it is demoted to the next lower priority.
- **Per-Level Quota:** Each level has an aggregate runtime limit. When this level-level quota is exhausted, all remaining processes at that level are collectively pushed down one level, even if they haven't used their individual quotas.
- **Fairness Guarantee:** Lower-priority processes eventually get CPU time despite higher-priority workloads, eliminating starvation.

Con Kolivas implemented RSDL as a Linux scheduler, however, it was not merged into mainline Linux. However, it heavily influenced the development of the Completely Fair Scheduler (CFS), which has been the default Linux scheduler since kernel version 2.6.23

UNIX SVR3
Scheduling Algorithm

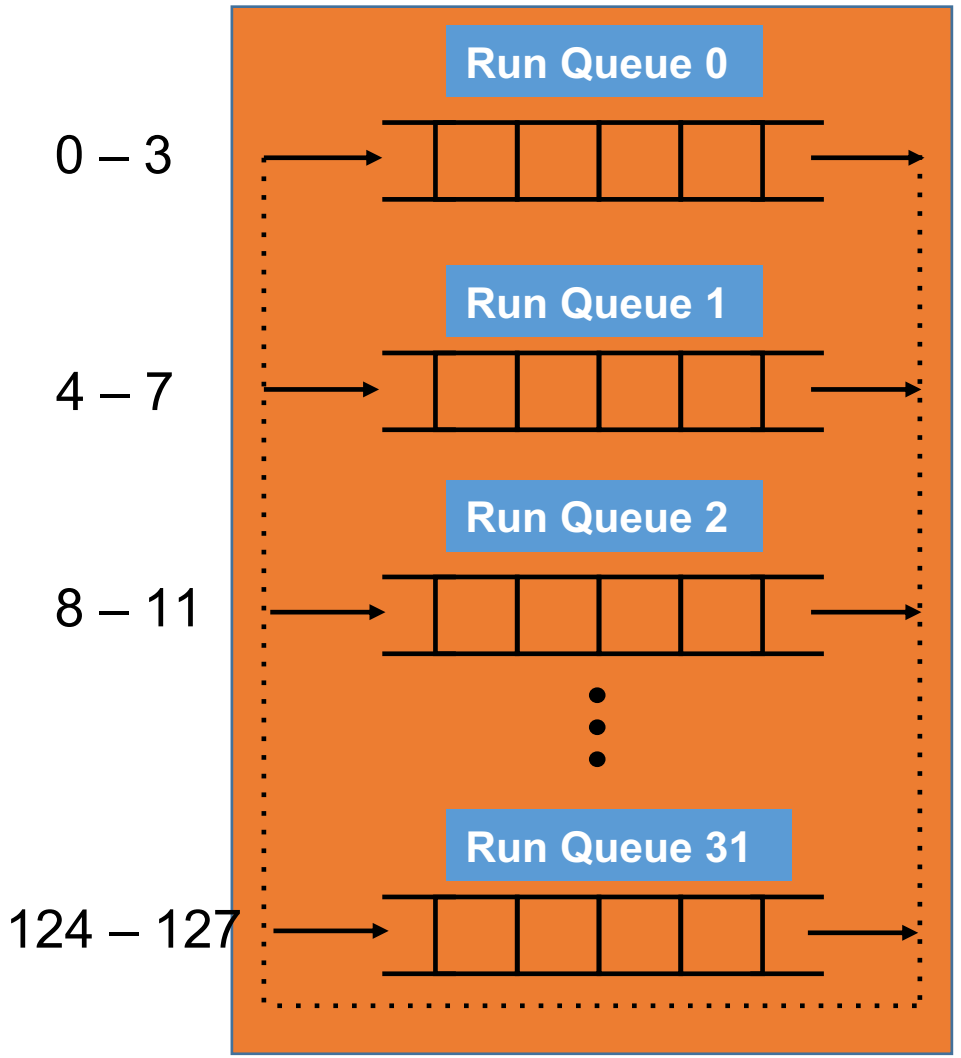
Unix Svr3 Scheduling Algorithm (Cont.)

- The Traditional UNIX scheduler employs thirty-two multi-level feed back queues implementing round robin algorithm with a fixed time quantum of 100 ms.
- There are total of 128 different priority values, 0 to 49 for kernel processes and rest for user level programs. Four priority values are mapped on each queue.
- A process enters in an appropriate queue based on its priority value, which is computed by a formula and is recomputed every second (not inherited).
- When it comes to scheduling the process in the smallest priority number queue is selected. After every second, the priorities of all the processes are recalculated and they are promoted or demoted in the queues accordingly.
- The priority of a process is calculated as the sum of three terms as shown in the formula:

$$\mathbf{usrpri_j(i) = Base_j + CPU_j(i) + nice_j}$$

- $\mathbf{Base_j}$ means base value for process j, which differentiate between user and kernel priorities. For user processes its value is 50-127, while for kernel processes its value is 0-49
- $\mathbf{CPU_j(i)}$, means the CPU utilization of process j through interval i. It is calculated by multiplying the previous cpu utilization with a decay rate. In SVR3 the decay rate is ½
- The nice value (a per process attribute) is a user controllable adjustment factor. It is called nice because a process increases its nice value and in turn reduces its priority and show nice behavior to other processes by giving them the opportunity to run. A user can change the nice value of a process by nice(1) and renice(1) commands. The nice values range from -20 to 19 with a default value of 0

Unix SVR3 Scheduling Algorithm



128 Priority values

- 0–49: Kernel
- 50–127: User level programs

$$usrpri_j\ (i) = Base_j + cpu_j\ (i) + nice_j$$

Where $Base_j = 50$

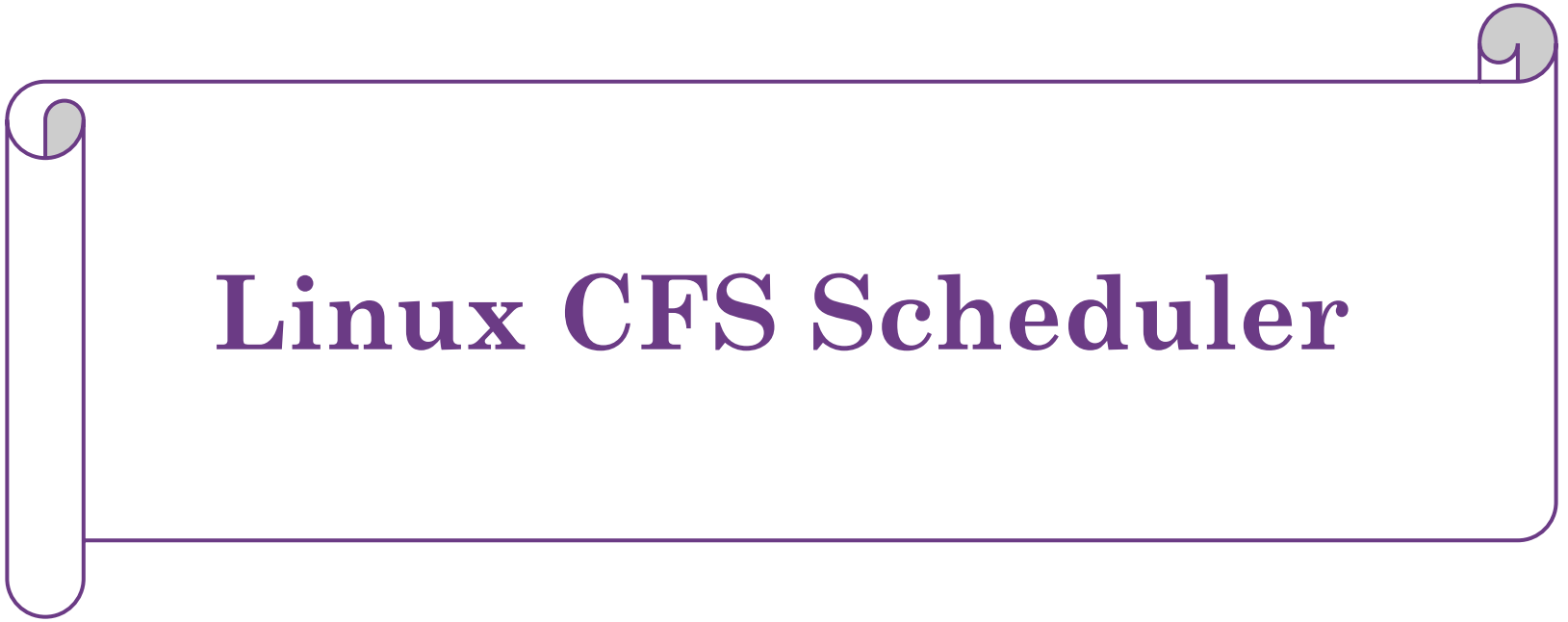
$cpu_j\ (i) = DR * cpu_j\ (i-1)$

$nice_j$ = -20 to +19

Limitations of Unix Svr3 Scheduling Algorithm

- With large number of processes, overhead of re-computing process priorities every second is very high.

- Since the kernel itself is non-preemptive, high priority processes may have to wait for low priority processes executing in kernel mode


Linux CFS Scheduler

Introduction to CFS

- The **Completely Fair Scheduler (CFS)** is the default process scheduler in modern Linux systems, introduced in kernel version 2.6.23. It was developed to address the limitations of earlier scheduling models, especially the **O(1)** scheduler, which, while efficient, lacked true fairness in how CPU time was distributed. It is designed by Ingo Molnar and is based on Rotating Staircase Deadline (RSDL) scheduler by Con Kolivas.
- Traditional schedulers used fixed time slices and discrete priority queues. This often led to:
  - Starvation of lower-priority or I/O-bound processes.
  - Poor responsiveness for interactive applications
  - Unfair CPU distribution, especially under load
  - Complexity in tuning scheduling policies manually
- CFS was designed to eliminate these issues by modeling CPU scheduling as a fair sharing problem.

CFS (Normal scheduling):

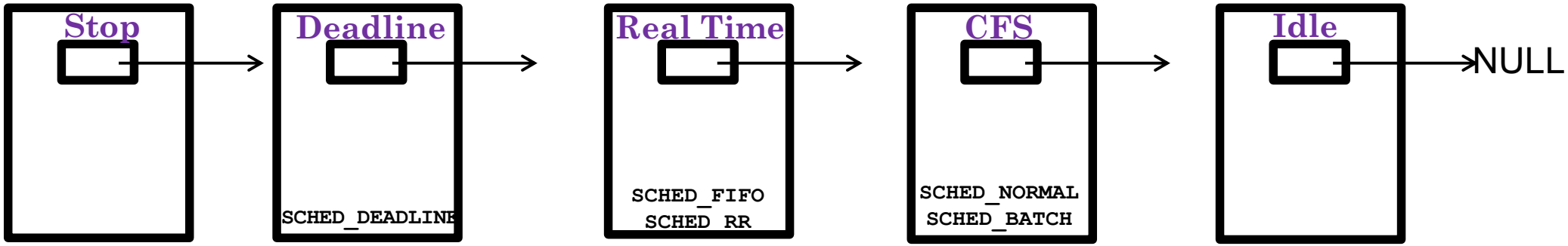
- Web browsers, text editors, compilers
- Most applications and background tasks
- When you want the system to stay responsive

Real-time scheduling:

- Audio/video processing with strict deadlines
- Industrial control systems
- Scientific data acquisition

Scheduling Classes of CFS

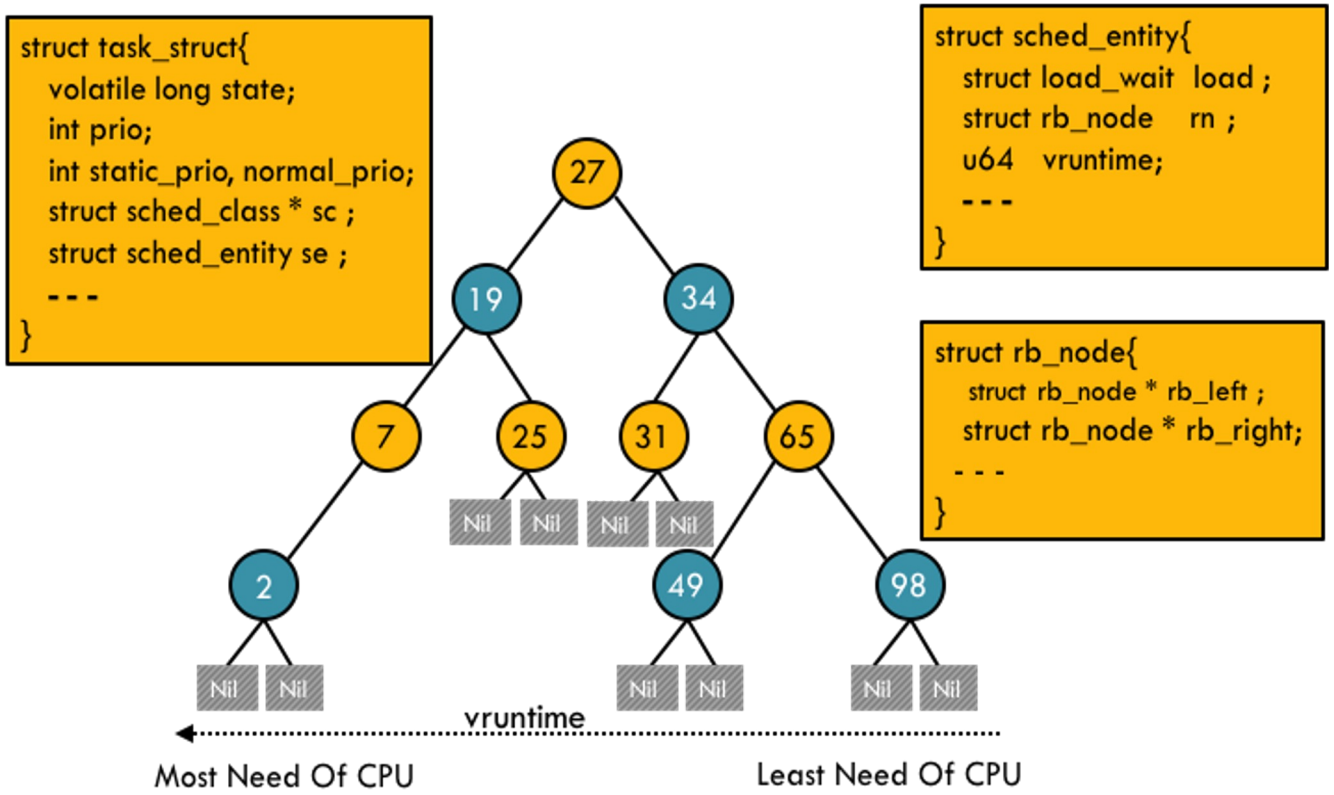
There are five scheduling classes: *Stop*, *Deadline*, *Realtime*, *CFS*, and *Idle*. The scheduler iterates over each class in priority order, starting with the highest priority class. If a class has a runnable process it is run, if not then the turn of a process from the lower priority class comes. In the very end, comes the turn of idle scheduling class, which runs last, never fails: it always returns the idle task.



- The **Stop class** is the highest priority class. A process of this class is preempted by nothing.
- The **Deadline class** is introduced in Linux kernel version 3.14. The scheduling policy used by this class is `SCHED_DEADLINE` and is used for periodic real time tasks.
- The **Realtime class** is used by POSIX real time tasks like IRQ threads with priority values in the range of 0 to 99. The scheduling policies used by this class are `SCHED_FIFO`, and `SCHED_RR`.
- The **CFS class** is the class under which most of the user processes including the famous bash command interpreter runs. It uses priority values in the range of 100 to 139. The scheduling policies used by CFS class are `SCHED_NORMAL` (which is the default for all user processes), and `SCHED_BATCH` (for CPU intensive tasks).
- Finally, the **Idle class** is the lowest priority scheduling class having no scheduling policy. The kernel idle thread runs when nothing else is runnable on a CPU and it may take the CPU to low power state as well.

Working of CFS

- CFS scheduler maintains a "time-ordered red black tree" to manage the list of runnable processes.
- Every process `task_struct` has a member `struct sched_entity`, which further contains `struct rb_node` to represent every node of the tree.
- The `sched_entity` structure also contains a 64 bit field `vruntime`, which indicates the amount of time a process has run and serves as the index for the red black tree.
- Tasks with the gravest need for the CPU (lowest `vruntime`) are stored towards the left side of the tree.
- The tasks with the least need for the CPU (highest `vruntime`) are stored towards the right side of the tree.

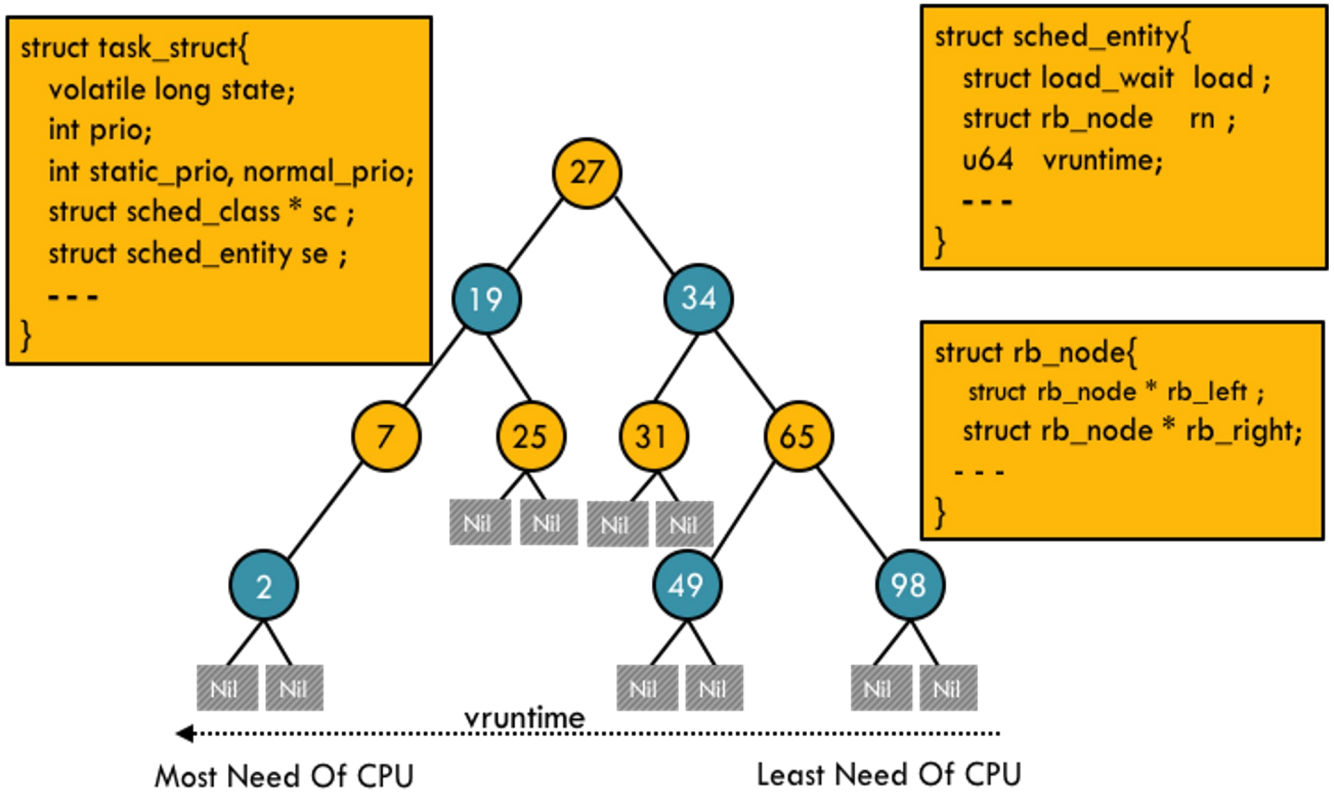


- Suppose we have a red-black tree populated with every runnable process in the system, with their `vruntime` shown in the figure. When it is time to select next process to run from the CFS class, the scheduler picks the left most node to maintain fairness.

Working of CFS (cont…)

Context Switch and Time Slice:

- In CFS, there is no concept of time slice. Once a process is selected to run, during execution its `vruntime` is incremented. A context switch or rescheduling occurs when:
  - The `vruntime` of another task is smaller than the currently running task, or
  - When a process that has a higher priority than the currently running process is awakened.

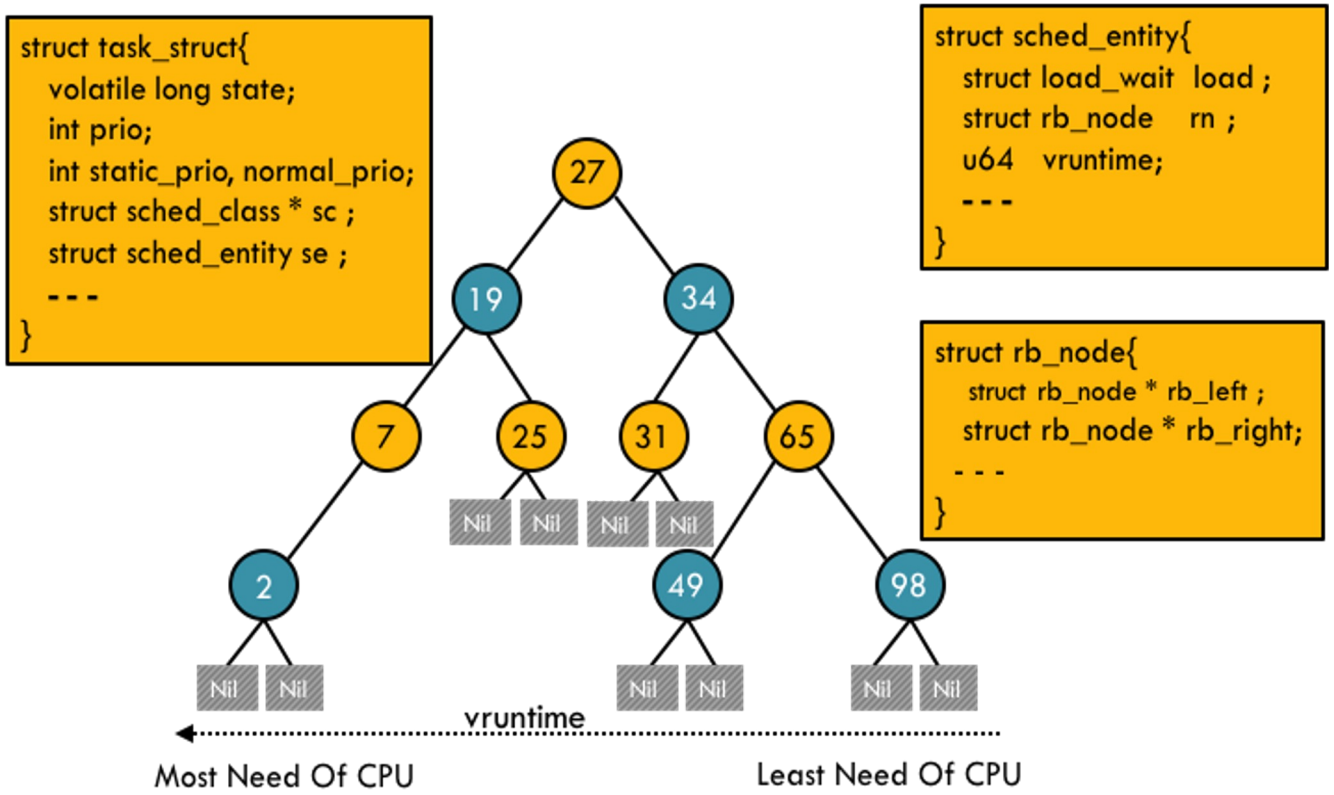


Working of CFS (cont…)

Vruntime of a new Process:

- CFS keeps track of the minimum `vruntime` value among all the processes in the tree. Whenever a new task is created, this value is given to it so as to give it a chance to schedule quickly and to achieve good response time

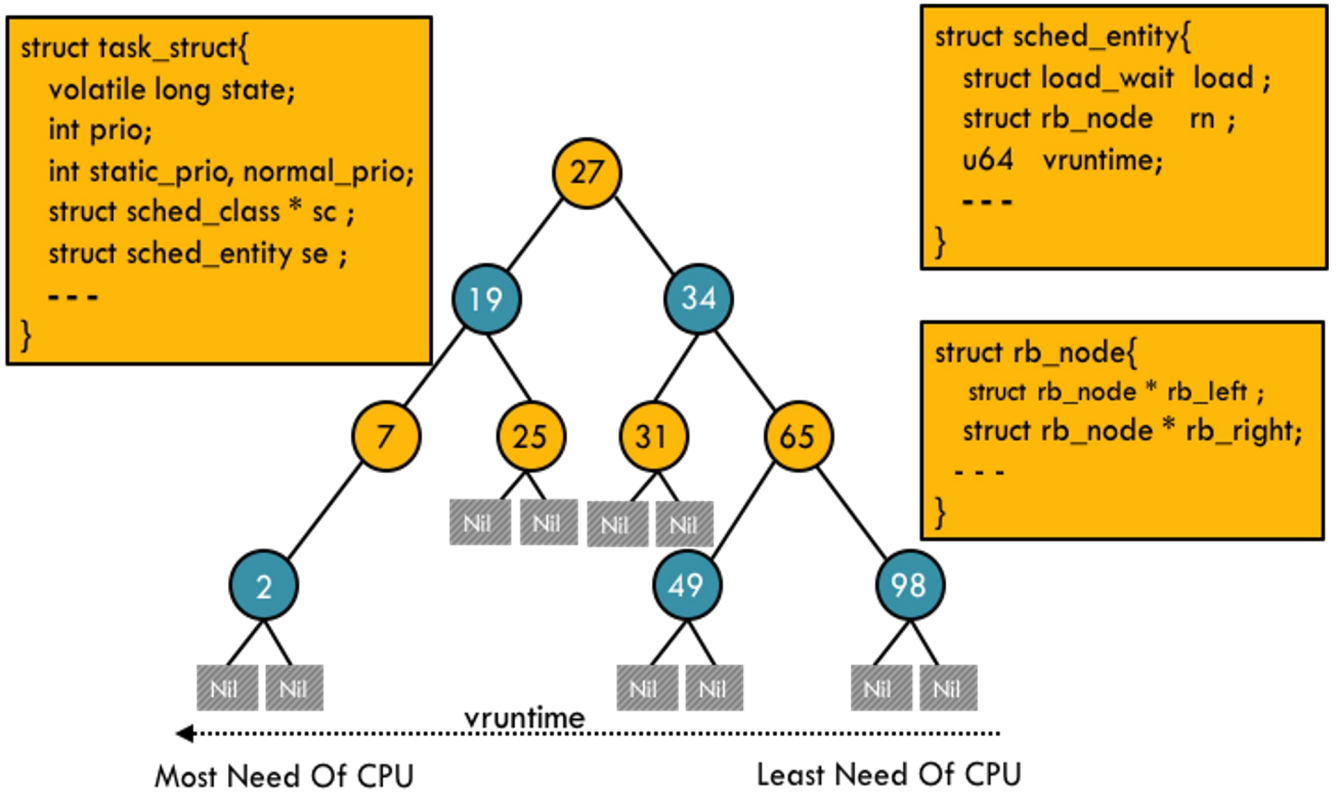
- Question: A user's process which is forking again and again will get more CPU time as compared to another user's process which is not forking. Answer: A process that forks child processes share its `vruntimes` among all processes of the group, while the single task maintains its own independent `vruntime`.



Working of CFS (cont…)

How Priorities are managed?

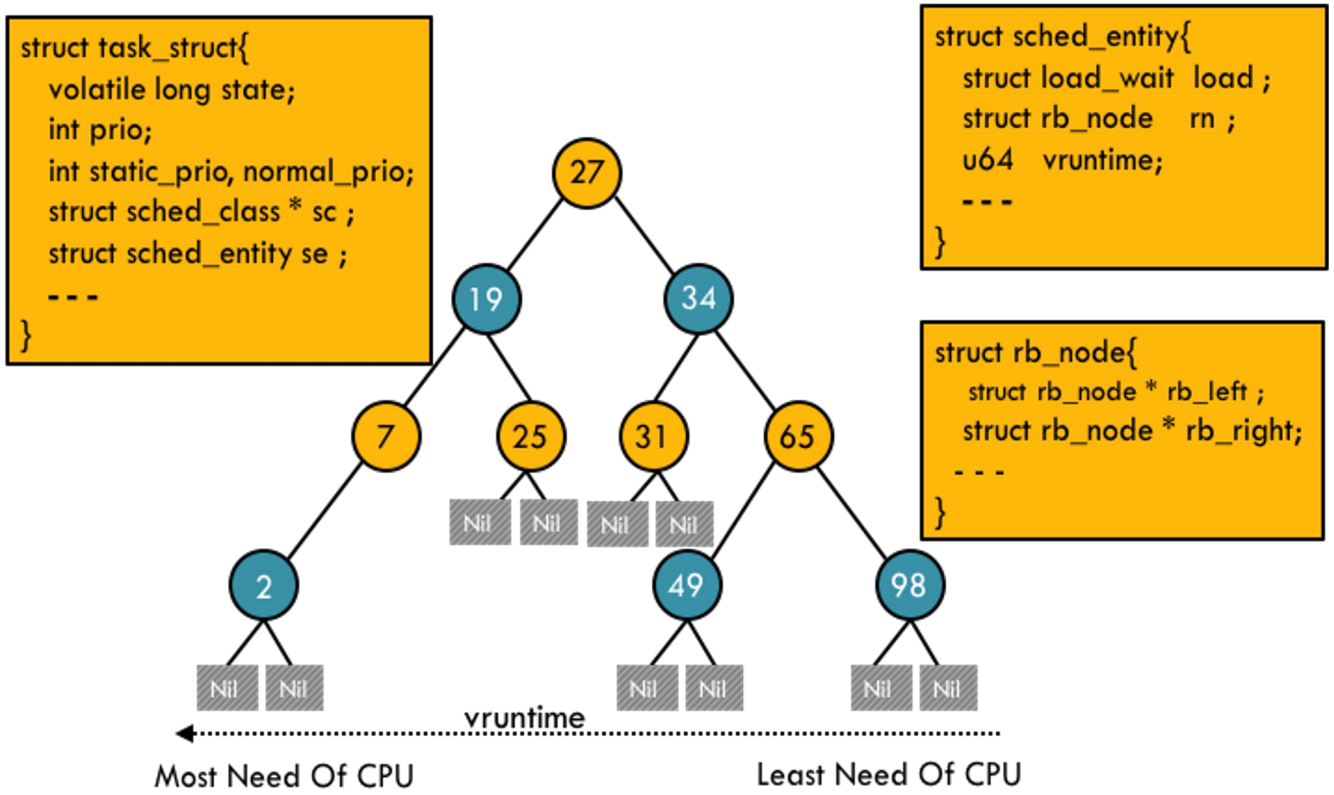
- CFS does not use priorities directly, but uses them as a decay factor of `vruntime`.
- Lower priority tasks have higher factors of decay, i.e., `vruntime` grows faster.
- Higher priority tasks have smaller factors of decay, i.e., `vruntime` grows slower



Working of CFS (cont…)

How CFS Handles CPU Bound and I/O Bound Processes?

- Consider the example of a text editor, an I/O bound process and a video encoder, which is a CPU bound process.
- The video encoder executes most of the times and therefore will have a high `vruntime` value
- On the contrary, the text editor sleeps for most of the time and therefore will have a very low `vruntime` value
- Hence, whenever the text editor wakes up it preempts the video encoder, handle the user keys and sleeps again



Scheduling Related Shell Commands

The `nice` & `renice` Commands

- The nice value (a per process attribute) is a user controllable adjustment factor. It is called nice because a process increases its nice value and in turn reduces its priority and show nice behavior to other processes by giving them the opportunity to run.
- Lower nice value: More CPU time (higher priority).
- Higher nice value: Less CPU time (lower priority).
- The **`nice`** command is used to set the priority at process start, while `renice` command is used to change the priority of a running process. The nice values range from -20 to 19 with a default value of 0. Regular users can only increase niceness of own processes, while root can set negative nice values as well.
- **Examples:**
  - Run a process with a nice value of -5: **`$ sudo nice -n -5 top`**
  - Make PID 1234 lower priority: **`$ renice -n 5 -p 1234`**
  - Increase priority of PID 5678: **`$ sudo renice -n -10 -p 5678`**
  - Change priority for all in group 1010: **`$ renice -n 15 -g 1010`**

Normal Processes (use CFS Scheduler):
PRI = 20 (base user priority) + nice value + 60
nice = 0 → PRI = 20 + 0 + 60 = 80 (default)
nice = -10 → PRI = 70 (higher priority)
nice = +10 → PRI = 90 (lower priority)

Real-time Processes (use SCHED_FIFO or SCHED_RR):
Real-time processes must be manually assigned a static PRI value from 0 to 99. You cannot change their PRI using `nice/renice`. However, can use `chrt` command

The `chrt` Command

- The **`chrt`** command is used to display or set a process's real-time scheduling policy and priority.
- Unlike **`nice`**, which adjusts relative niceness under the **CFS**, **`chrt`** sets real-time scheduling attributes, managed under the Real-Time Scheduling Classes in Linux. [**`$ chrt --help`**]
- Real-time scheduling gives selected processes predictable CPU access ahead of normal tasks.
- It can set policies like SCHED_FIFO or SCHED_RR with priorities ranging from 1 (lowest) to 99 (highest) for real-time tasks.
- Changing a process to a real-time policy can make it highly responsive but may starve other processes if not managed carefully.
- Regular users can view scheduling policies and priorities, but only root can change them.
- **Examples:**
  - Display scheduling policy and priority of PID 1234: **`$ chrt -p 1234`**
  - Launch a process with FIFO policy and priority 50: **`$ sudo chrt -f 50 <mycommand>`**
  - Change running process to RR policy with priority 20: **`$ sudo chrt -r -p 20 <PID>`**
- **Note:** Only root can assign real-time policies and even doing that misusing real-time scheduling can lock up the system if a task never yields the CPU. So use with caution

The `taskset` Command

- **CPU affinity** is a per-process attribute that defines the set of CPU cores on which a process is allowed to run. In a multi-processor / multi-core system, when a process is rescheduled, it does not necessarily run on the same CPU on which it ran previously. If a process moves from one CPU to the other, the cache of the first CPU must be invalidated and the cache of the second CPU must be populated with the process data. Restricting a process to specific CPUs can improve cache performance, reduce context switching, and control CPU load distribution.
- CFS tries to ensure **soft CPU affinity**, i.e., tries to run the task on the same CPU on which it ran previously. We can ensure **hard CPU affinity** using a per process attribute `cpus_allowed`, which is a 32 bit mask having one bit per CPU or core in the system.
- The **`taskset`** command is used to view or set a process's CPU affinity mask, either for a new process or for an already running one. Regular users can change the affinity of their own processes, while root can set affinity for any process.
- **Examples:**
  - Display CPU affinity of a process:: **`$ taskset -p <PID>`**
  - Launch a command restricted to CPU 0 & 2: **`$ taskset -c 0,2 <mycommand>`**
  - Launch a server to CPU 0 to 3: **`$ taskset -c 0-3 ./server`**

The schedtool Command

- The **schedtool** is a powerful CLI utility to view and modify Linux process scheduling policies and priorities.
- Use `sudo apt-get install schedtool` to install it on you machine.

```
$ schedtool
get/set scheduling policies - v1.3.0, GPL'd, NO WARRANTY
USAGE: schedtool PIDS                          - query PIDS
       schedtool [OPTIONS] PIDS                - set PIDS
       schedtool [OPTIONS] -e COMMAND          - exec COMMAND
set scheduling policies:
    -N                    for SCHED_NORMAL
    -F -p PRIO            for SCHED_FIFO       only as root
    -R -p PRIO            for SCHED_RR         only as root
    -B                    for SCHED_BATCH
    -I -p PRIO            for SCHED_ISO
    -D                    for SCHED_IDLEPRIO
    -M POLICY             for manual mode; raw number for POLICY
    -p STATIC_PRIORITY    usually 1-99; only for FIFO or RR
                          higher numbers means higher priority
    -n NICE_LEVEL         set niceness to NICE_LEVEL
    -a AFFINITY_MASK      set CPU-affinity to bitmask or list

    -e COMMAND [ARGS]     start COMMAND with specified policy/priority
    -r                    display priority min/max for each policy
    -v                    be verbose
```

The schedtool Command (cont...)

- **Query Process Information:**
  - Display all scheduling info for a specific process: `$ schedtool -v 1234`
  - Query multiple processes: `$ schedtool -v 1234 2345 5678`
- **Set Nice Value of a Process:**
  - Set nice value to 10: `$ schedtool -n 10 -p 1234`
  - Set nice value to -5: `$ sudo schedtool -n -5 -p 1234`
  - Launch a command with nice value: `$ schedtool -n 15 -e make -j4`
- **Set CPU Affinity:**
  - Pin to CPU 0 only: `$ schedtool -a 0 -p 1234`
  - Pin to CPU 0, 2 and 3: `$ schedtool -a 0,2,3 -p 1234`
- **Set Scheduling Policy:**
  - Set to batch scheduling (SCHED_BATCH): `$ schedtool -B -p 1234`
  - Set to normal scheduling (SCHED_OTHER): `$ schedtool -N -p 1234`
- **Set Static Priority (Real-time only) :**
  - Set FIFO with priority 50: `$ sudo schedtool -F -p 50 -P 1234`
  - Set to RR with priority 80:: `$ sudo schedtool -R -p 80 1234`

Process Scheduling Info in `/proc/[PID]/`

Process Scheduling info in `/proc/[PID]/`

File	Content Description
`stat`	Contains scheduling policy, nice value, priority, CPU times, and state in space-separated fields
`status`	Human-readable format showing voluntary/involuntary context switches and CPU times
`sched`	Detailed CFS scheduler statistics including runtime, wait time, and scheduling policy
`task/[TID]/stat`	Per-thread scheduling information (same format as main stat file)
`task/[TID]/sched`	Per-thread detailed scheduler statistics
`cpuset`	Shows which CPU cores/nodes the process is allowed to run on
`comm`	Process command name (useful for identifying the process)
`cmdline`	Full command line with arguments

Scheduling Related System Calls

System Calls Related to Scheduling

System Call	Description
`nice()`	Adjusts the calling process's nice value by a specified increment
`getpriority()`	Retrieves the nice value (priority) of a specified process, process group, or user
`setpriority()`	Sets the nice value (priority) of a specified process, process group, or user
`sched_get_priority_min()`	Returns the minimum priority for a given scheduling policy
`sched_get_priority_max()`	Returns the maximum priority for a given scheduling policy
`sched_getscheduler()`	Retrieves the current scheduling policy of a given process or thread
`sched_setscheduler()`	Sets the scheduling policy and real-time parameters for a specified process
`sched_getparam()`	Gets the real-time scheduling parameters (e.g., priority) of a specified process
`sched_setparam()`	Sets the real-time scheduling parameters (e.g., priority) for a specified process
`sched_yield()`	Voluntarily gives up the CPU, allowing other threads of equal priority to run
`sched_rr_get_interval()`	Fetches the time quantum used under the SCHED_RR (round-robin) policy for a process
`sched_getcpu()`	Returns the number of the CPU on which the calling thread is currently executing
`sched_getaffinity()`	Retrieves the CPU affinity mask, indicating on which CPUs a process is allowed to run on
`sched_setaffinity()`	Sets the CPU affinity mask to specify which CPUs a process is permitted to execute on

To Do

- Watch OS video on Process Scheduling:
https://www.youtube.com/watch?v=3ap2kU4bA9E&list=PL7B2bn3G_wfBuJ_WtHADcXC44piWLRzr8&index=12

- Watch SP video on Process Scheduling:
https://www.youtube.com/watch?v=Y86pa2nrT_k&list=PL7B2bn3G_wfC-mRpG7cxJMnGWdPAQTViW&index=21

Coming to office hours does NOT mean that you are academically weak!