Android OS - Processes Scheduling

Android Process Scheduling

Process scheduling in Android, similar to other **multitasking operating systems**, is the **method** by which the **system allocates CPU time** to various running applications (processes).

KEY aspects:

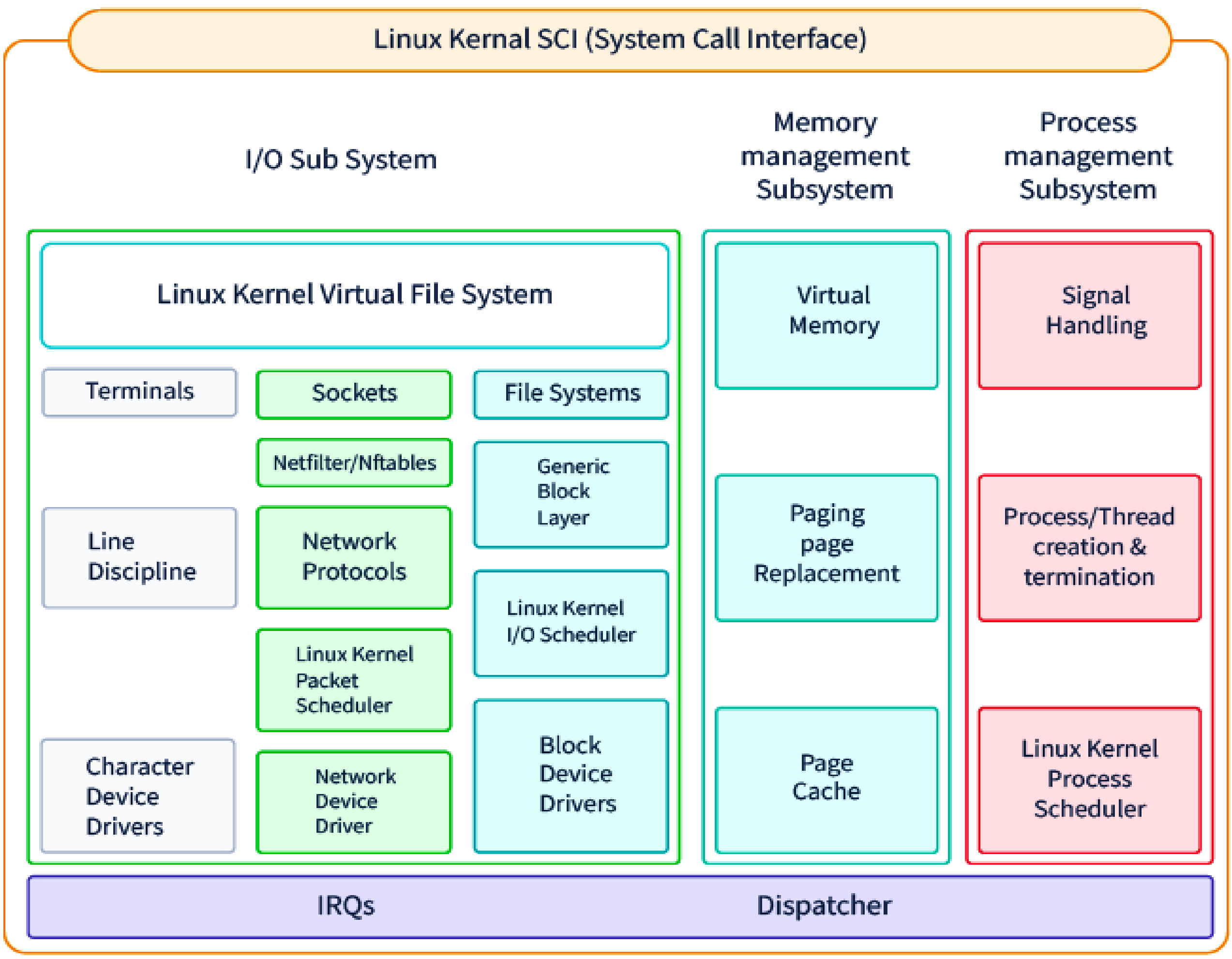
- **Preemptive Scheduling:**
  - Android utilizes **a preemptive scheduling algorithm**,
  - **inheriting** this from the **Linux kernel** at its core.
  - **interrupt** a running process with **a lower priority** if a **higher priority process** enters the "ready" state.
- **Priority Based:** Android uses a **priority-based scheme** to determine which process gets **CPU time**. There are generally two categories:
  - **Foreground processes**
  - **Background processes**
- **Linux Kernel Scheduling:** Android leverages the **scheduling mechanisms** of the Linux kernel, specifically the **Completely Fair Scheduler (CFS).**

CPU Scheduling: **Dispatcher**.

- The **Dispatcher** is the **element** that **contains the CPU scheduling function**.
- **Dispatcher module** is used in **CPU scheduling**, which provides **control** to the **CPU** in the **selection** of processes using the **short-term scheduler**.
- **Dispatcher involves**: **Context switching; Switching to user mode**.

CPU Scheduling: is a **process** of determining which process will **own CPU for execution** while another process is **on hold**.

- The main task of CPU scheduling is to make sure that whenever the **CPU** remains **idle**, the **OS** at least **select one** of the processes available in the **ready queue** for execution.



Presented Scheduling Algorithms

For batch systems

- First-Come First-Served (**FCFS**)
- Shortest Job First (**SJF**)
- Shortest Remaining Time (**SRT**)

For interactive systems

- Round-Robin scheduling (**RR**)
- Priority-based scheduling (**PBS**)
- Group-based scheduling (**GBS**)
- Fair-share scheduling (**FSS**)
- Lottery scheduling (**LS**)

Scheduling classes

- **Real-time processes**: **SCHED_FIFO**, **SCHED_RR**
- **Interactive and batch processes**: **SCHED_OTHER, SCHED_BATCH**
- **Low-priority processes**: **SCHED_IDLE**

One active queue for each of the 140 priorities and for each processor.

Android CPU scheduling

- **Roughly**, the scheduler is based on the **Linux** one
  - → **Fair scheduling approach**
- **BUT**: **fairness** according to **Groups of processes**
  - **Foreground/Active, visible, service, background, empty**
- To **reclaim resources**, Android may **kill** processes according to their **running priority.**

Completely Fair Scheduler

Completely fair Scheduler (CFS):

- It is based on **Rotating Staircase Deadline Scheduler (RSDL).**
- It is **default scheduling** process since **version 2.6.23**.
- Elegant handling of **I/O** and **CPU bound process.**

CFS implements three scheduling policies:

1. **SCHED_NORMAL** (traditionally called SCHED_OTHER): The scheduling policy that is used for **regular tasks**.

2. **SCHED_BATCH**: Does not preempt nearly as often as **regular tasks** would, thereby allowing **tasks to run longer and make better use of caches** but at the cost of interactivity. This is well suited for **batch jobs**.

3. **SCHED_IDLE**: It is also derived from SCHED_OTHER, but it has **nice values weaker** than **19**.

SCHED_FIFO/_RR are implemented in **sched/rt.c** and are as specified by **POSIX**.

Completely Fair Scheduling (CFS) — CPU Scheduler Algorithm

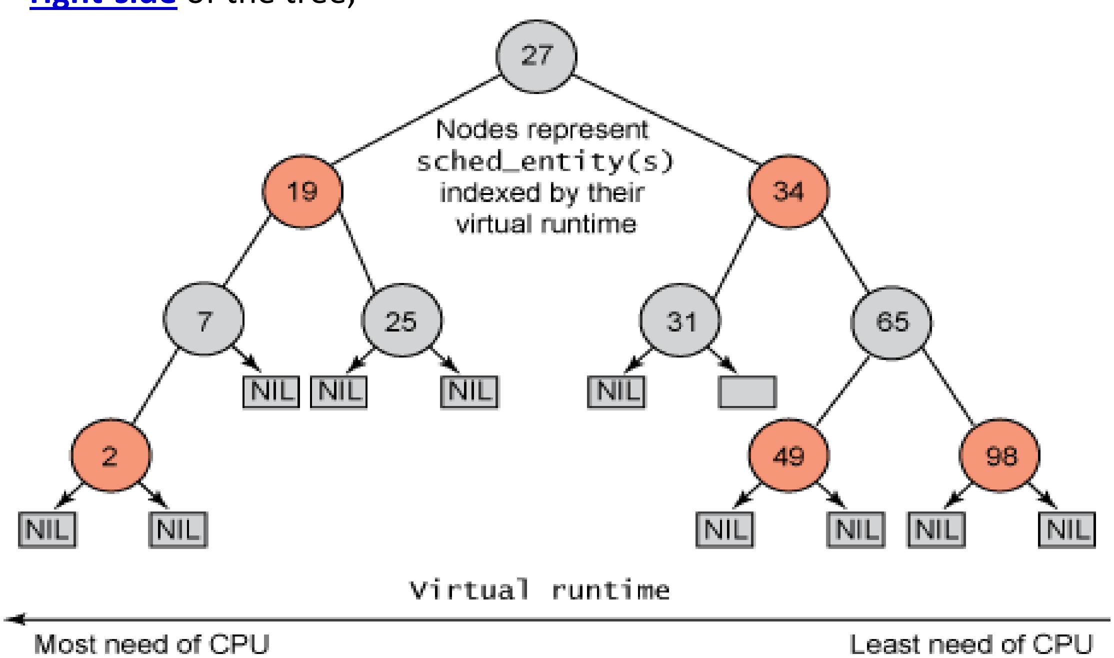
- Both **interactive** and **non-interactive** processes can fit into this easily
- Each process receives **equally fair CPU time** for execution. **Idea**: If **N** processes are in the system, each process should have for **(100N) %** of the **CPU time**.

How CFS algorithm is incorporated in CPU scheduler?

- Each process **PCB (process control block)** has an entry for '**virtual runtime (vruntime)**'.
- At every scheduling point, if the process has run for **t ms**, then its **vruntime** is **incremented** by **t**. So **vruntime** monotonically **increases. i.e** **vruntime += t**
- Whenever timer **interrupt** or **context-switch** happens, it always chooses the next task with the **lowest vruntime** (**min_vruntime**).
- **min_vruntime** is a **pointer** which points to the **lowest vruntime**.
- **Time slice** will be **dynamically recomputed**; Process executes the task ; context-switch again occurs and **cycle continues**.

How the internal mechanism of CFS picking the next task to run works?

- **CFS uses Red-Black tree data structure** (self-balancing binary search tree) ; **inserting/deleting** tasks from the tree is **O(logN)**
- Each **node** represents a **runnable task**
- **Nodes** are **ordered** according to **vruntime**;
- **Nodes** in **left-side** have **lower vruntime** compared to the nodes on the **right-side** of the tree;



- **At the point of context-switch:**
  - It picks the **left most node** of the tree in **O(1)** as its cached in **min_vruntime**
  - If the **previous process is still runnable**, it is **re-inserted** into the tree with re-computed **vruntime** in **O(logN)**;
  - Tasks **move from left to right side** of the tree after its execution completes and hence "**Starvation**" has been **avoided**.

Starvation — > When **high priority processes** keep executing and **low priority processes** get **blocked** for indefinite time

CFS does NOT use any **priority-based queues** and **priority** is used to just weigh the **vruntime** (i,e **nice values**).

How I/O and CPU bound processes are handled by CFS?

- **I/O bound** processes should get **higher priority** and get a **longer time** to execute compared to **CPU bound** processes.

Because

- **I/O processes** have **low CPU burst time** , so they will have **lower vruntime**. They mainly would appear on the **left-side** of the **RBtree** and so ending up with **higher priorities**!

What happens when a new process is created?

- It gets added to **RB-tree**
- Starts with **initial value** of **min_vruntime** and that ensures that it gets to execute quickly as possible[**Remember**, **lower vruntime tasks end up in left-side**!]

EX-(CFS algorithm): Let's take **four process** and their **burst time** as shown below waiting in the ready queue for the execution:

Process	Burst Time (in ms)
A	**10 ms**
B	**6 ms**
C	**14 ms**
D	**6 ms**

CFS	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
A	1	2	3	4	5	6	**8**	**10**	
B	1	2	3	4	5	**6**			
C	1	2	3	4	5	6	**8**	**10**	**14**
D	1	2	3	4	5	**6**			

Execution = (Time quantum/N).

- So **4/4=1** each process gets **1ms** to execute in **first quantum**.
- **After** the completion of **six quantum** process **B** and **D** are completely executed.
- **Remaining** are **A** and **C**, which are already executed for **6ms** and their **remaining time** is **A**=4ms and **C**=8ms).
- In the **seventh quantum** (**Q7**) of time **A** and **C** will execute (**4/2=2ms** as there are only **two process** remaining).