Kernel 2.6 Scheduling algorithm:

Kernel 2.4 bottleneck:

The Linux kernel 2.4 the whole global run queue was used to decide which process to schedule. Thus, making this procedure of O(n) asymptotically or in other words the time taken to find out which process to execute next was directly proportional to the number of available processes in the run queue. This approach is not scalable for large no of processes. Scalability issues were found in JVM as it spawns many tasks.

There were two scheduling algorithms used in kernel 2.6:

1. O (1) scheduling algorithm
2. Completely fair scheduler algorithm

O (1) scheduling algorithm:

The early versions of kernel 2.6, Ingo Molnár came up with a new scheduler that had O (1) time complexity asymptotically. This meant that the scheduler would execute in a certain constant amount of time regardless of the number of tasks currently available for execution. This made the kernel to even handle enormous number of tasks without increasing the overhead cost as the tasks increases. The O (1) scheduler has two major components run queue and priority list.

The processes are divided into two categories real time processes and normal processes. The normal processes are further divided into two interactive and batch processes. Real time processes are given priorities in the range 0 to 99 and normal processes are given priority in the range 100 to 139. Here lower no corresponds to higher priority.

Each of the priority list have their run queue in which the corresponding processes are placed at the end of the queue. Each task has a time slice given to it for execution. There are two run queues, active and expired, the active queue has the process that needs to be executed and have not yet been allotted CPU time. After one process has been executed it is placed in the expired queue and thus help in preventing starvation. Before placing into the expired queue, the priorities and time slice of processes are recalculated. When the active queue becomes empty the active and expired queue are swapped making the expired queue active and vice versa.

The process of scheduler has two steps:

1. Find the highest priority in the active queue that is nonempty
2. Execute the first task in that queue

The above steps are done in O (1) by using a bit map for run queue and is set if it has nonzero entries and special instruction *find-first-bit-set* is used to find out the first non-zero entry in that bitmap in O (1). As second the first element of a linked must be fetched it can be done O (1) thus making the entire process of constant time complexity. After the process has executed in its time slice it is placed in expired queue after recalculating the time slice and priority which is also of constant time complexity. Once the active queue becomes empty instead of copying the expired and active queues are just swapped making this operation also in O (1). [i]

Completely fair scheduling algorithm:

After the kernel version 2.6.22 the O (1) scheduler was not used as it had limitations. The O (1) scheduler used complex heuristics to distinguish between batch and interactive process, had dependence in between time slice and priorities and the values of priorities and time slice values were not uniform. Therefore, a new scheduler i.e., completely fair scheduler (C.F.S) was used from the kernel version 2.6.23.

CFS models an "ideal, precise multi-tasking CPU" on real hardware. Ideal multi-tasking CPU is a CPU that has 100% physical power, and which can run each task at precise equal speed (1/n where n is the no process currently in the system). The CFS does this with the help of an attribute known as virtual run time which provides us with the amount of time that has been provided to a task. If the virtual run time is small that means the requirement for this process execution is high. It also incorporates sleeper fairness where if a process is not in runnable state right now will eventually get a comparable share of CPU when it becomes available.

CFS maintains a red-black tree with key value as the virtual run time. At the start of every scheduling point the virtual run time is recalculated. A pointer to the leftmost node of this tree is also maintained for constant time access to that node. The leftmost node has the smallest virtual runtime and thus is scheduled first. The following steps are performed in C.F.S:

1. Get and remove the leftmost node of the tree. Execute it for the dynamically calculated time slice
2. If completed remove, it from the tree and the system. Otherwise, insert the node again after recalculating its virtual run time.[ii]

Process of setting and updation of priorities and time slice:

O (1) scheduler:

Static priorities:
Normal processes (not real-time) are provided with a static priority of 120 by default. This static priority of a process can be changed with nice command which takes an integer parameter in the range +19 to –20 to decrease or increase the priority. The static priority is stored in *static_prio* variable.

Dynamic priority:
To implement sleeper fairness the scheduler rewards I/O bound (interactive) processes and punishes CPU bound (batch) processes by setting up dynamic priorities. The dynamic priority is calculated using the following formula:

Dynamic Priority = Max (100, Min((static_priority – bonus + 5), 139)

Here the bonus value is calculated through heuristics and its value ranges from 0 to 10. If a process bonus value is less than 5 it resembles that the process is more CPU and thus increasing its priority and if the bonus value is greater than 5 it resembles more interactive process and thus its priority moves closer to 100 by lower it.

The heuristic for calculation of bonus value is based on the **average sleep time** of the process. An attribute sleep_avg is maintained and is updated after a process is woken up or when it gives up CPU voluntarily or involuntarily. After a process has woken up its sleep time is added to the sleep_avg variable i.e., *sleep_avg += sleep_time* and after it has given up the CPU its runtime is subtracted from the sleep_avg variable i.e., *sleep_avg -= run_time.* As the process may sleep for a long time which in turn will result in high bonus and high priority, due to which it will result in overhaul of CPU resources due to interactivity. The scheduler interactivity heuristics is used to prevent this condition to occur. The bonus is given according to the below formula:

$$bonus = current\_bonus - max\_bonus / 2$$

The current_bonus is given by:

$$current\_bonus = sleep\_avg * max\_bonus / max\_sleep\_avg$$

The current_bonus is used to map a task's average sleep time to the range 0-max_bonus, which is 0-10. **This is the role of average sleep time in the scheduling process**.

Time slice updation:

Time slice is calculated by scaling the given static priority of the process to given range of time slice set. The minimum and maximum value of these are already known to the scheduler. A higher priority task will get a larger time slice than a lower priority task. The timeslice function is defined in as an macro with the following definition:

$$timeslice = (min\_timeslice + ((max\_timeslice - min\_timeslice) * (max\_prio - 1 - static\_prio) / (max\_user\_prio - 1)))$$

Timeslice maps the static priority of the task to a range defined by (min_timeslice, max_timeslice). Static_prio stands for the static priority of the task.

CFS Scheduler:

No such complex heuristics are used in CFS as compared to O (1) which results in elegant handling of I/O bound and CPU bound processes. No priority queues are implemented in CFS scheduler but instead priorities are used in weighted tasks where each task is provided with a weight according to its static priority. A lower priority implies time moves at a faster rate for it than a task with higher priority.

Time slice updation:As a process waits for the CPU, the scheduler tracks the amount of time it would have used on the ideal processor. This wait time, represented by the per-task wait_runtime variable. CFS's task picking logic is based on this wait_runtime, it always tries to run the task with the largest wait_runtime value. In other words, CFS tries to run the task with the 'gravest need' for more CPU time. So CFS always tries to split up CPU time between runnable tasks as close to 'ideal multitasking hardware' as possible.[iii]

The system runs a task a bit, and when the task schedules (or a scheduler tick happens) the task's CPU usage is 'accounted for': the (small) time it just spent using the physical CPU is deducted from wait_runtime. [minus the 'fair share' it would have gotten anyway]. Once wait_runtime gets low enough so that another task becomes the 'leftmost task' of the time-ordered rbtree it maintains then the new leftmost task is picked and the current task is preempted. The rq->fair_clock value tracks the 'CPU time a runnable task would have fairly gotten, had it been runnable during that time'. So by using rq->fair_clock values we can accurately timestamp and measure the 'expected CPU time' a task should have gotten. All runnable tasks are sorted in the rbtree by the "rq->fair_clock - p->wait_runtime" key, and CFS picks the 'leftmost' task and sticks to it. As the system progresses forwards, newly woken tasks are put into the tree more and more to the right - slowly but surely giving a chance for every task to become the 'leftmost task' and thus get on the CPU within a deterministic amount of time. The time slice is updated according to the weight corresponding to the static priority.[iv]

References:

1. https://www.linuxjournal.com/node/10267
2. https://people.eecs.berkeley.edu/~kubitron/courses/cs194-24-S14/hand-outs/linuxKernelUnderstandingQueudet.pdf
3. http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt
4. https://trepo.tuni.fi/bitstream/handle/10024/96864/GRADU-1428493916.pdf
5. https://www.youtube.com/watch?v=scfDOof9pww
6. https://www.youtube.com/watch?v=vF3KKMI3_1s&feature=youtu.be

[i] youtube.com/watch?v=vF3KKMI3_1s&feature=youtu.be
[ii] https://people.eecs.berkeley.edu/~kubitron/courses/cs194-24-S14/hand-outs/linuxKernelUnderstandingQueudet.pdf
[iii] https://www.linuxjournal.com/node/10267
[iv] http://people.redhat.com/mingo/cfs-scheduler/sched-design-CFS.txt