Contents

1.0 Introduction 3

1.1 Types of operating system schedulers 3

1.2 Context Switching 6

1.3 Linux and Linux Kernel Scheduler 7

2.0 Linux Kernel and Scheduler 9

2.1 Linux Kernel 9

2.1.1 History of Linux Kernel 10

2.1.2 Overview to Linux kernel 11

2.2 Linux Scheduler 12

2.2.1 History of Linux Scheduler 12

2.2.2 Problems with earlier Linux schedulers 14

2.2.3 Linux 2.6 Scheduler 14

3.0 Completely Fair Scheduler 18

3.1 Scheduling Goals 18

3.1.1 Fairness and Preventing Starvation 18

3.1.2 Efficiency 19

3.1.3 Interactivity and Impact on Fairness 19

3.1.4 Load Balance 19

3.1.5 Real World Application Performance 20

3.2 Overview of CFS Scheduler 20

3.3 Red-Black Tree (RBTree) 21

3.4 CFS Internals 24

3.5 Priorities and CFS 25

3.6 CFS group scheduling 25

3.7 Scheduling classes and domains 26

3.8 Multitasking in CFS 27

4.0 Conclusion 30

5.0 Future Scope 31

6.0 List of Figures 32

7.0 References 33

8.0 Abbreviations 34

CHAPTER-1

Introduction

1.0 Introduction

Scheduling is a key concept in computer multitasking, multiprocessing operating system and real-time operating system designs. **Scheduling** refers to the way processes are assigned to run on the available CPUs, since there are typically many more processes running than there are available CPUs. This assignment is carried out by software known as a **scheduler** and **dispatcher**.

The scheduler is concerned mainly with:

- Throughput - number of processes that complete their execution per time unit.
- Latency, specifically:
  - Turnaround - total time between submission of a process and its completion.
  - Response time - amount of time it takes from when a request was submitted until the first response is produced.
- Fairness - Equal CPU time to each process (or more generally appropriate times according to each process' priority).

In practice, these goals often conflict (e.g. throughput versus latency), thus a scheduler will implement a suitable compromise.

In real-time environments, such as mobile devices for automatic control in industry (for example robotics), the scheduler also must ensure that processes can meet deadlines; this is crucial for keeping the system stable. Scheduled tasks are sent to mobile devices and managed through an administrative back end.

1.1 Types of operating system schedulers

Operating systems may feature up to 3 distinct types of a long-term scheduler (also known as an admission scheduler or high-level scheduler), a mid-term or medium-term scheduler and a short-term scheduler. The names suggest the relative frequency with which these functions are performed [1].

a. Long-term scheduler

The long-term, or admission scheduler, decides which jobs or processes are to be admitted to the ready queue; that is, when an attempt is made to execute a program, its admission to the set of currently executing processes is either authorized or delayed by the long-term scheduler. Thus, this scheduler dictates what processes are to run on a system, and the degree of concurrency to be supported at any one time - i.e.: whether a high or low amount of processes are to be executed concurrently, and how the split between IO intensive and CPU intensive processes is to be handled. In modern OS's, this is used to make sure that real time processes get enough CPU time to finish their tasks. Without proper real time scheduling, modern GUI interfaces would seem sluggish [1].

Long-term scheduling is also important in large-scale systems such as batch processing systems, computer clusters, supercomputers and render farms. In these cases, special purpose job scheduler software is typically used to assist these functions, in addition to any underlying admission scheduling support in the operating system.

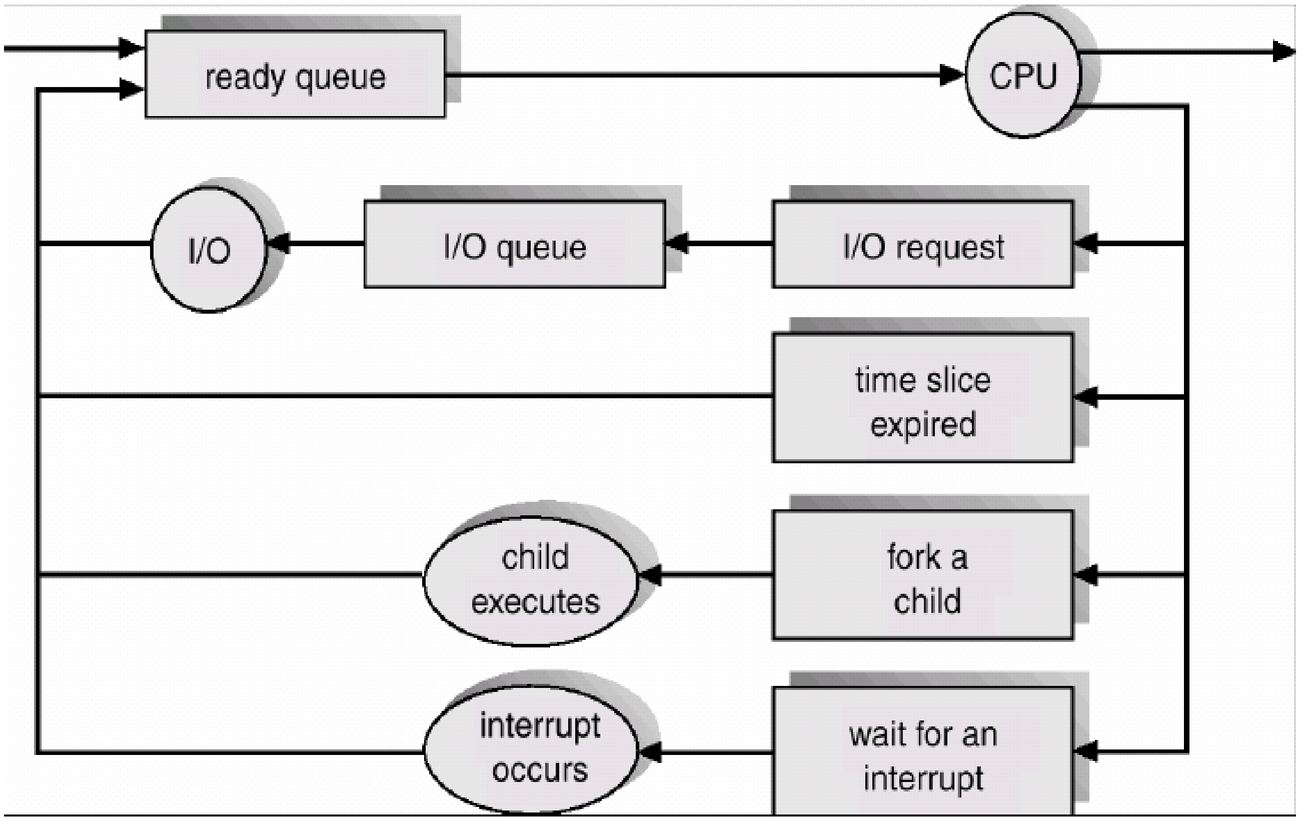

Figure 1.1: Queuing-diagram representation of process scheduling [2]

A common representation for a discussion of process scheduling is queuing diagram as shown in figure 1.1. Each rectangular box represents queue. Two types of queues are present: the ready queue and a set of device queue. The circles represent the resources that serve the queues and the arrows indicate the flow of processes in the system.

b. Mid-term scheduler

The mid-term scheduler temporarily removes processes from main memory and places them on secondary memory (such as a disk drive) or vice versa. This is commonly referred to as "swapping out" or "swapping in" (also incorrectly as "paging out" or "paging in"). The mid-term scheduler may decide to swap out a process which has not been active for some time, or a process which has a low priority, or a process which is page faulting frequently, or a process which is taking up a large amount of memory in order to free up main memory for other processes, swapping the process back in later when more memory is available, or when the process has been unblocked and is no longer waiting for a resource.

In many systems today (those that support mapping virtual address space to secondary storage other than the swap file), the mid-term scheduler may actually perform the role of the long-term scheduler, by treating binaries as "swapped out processes" upon their execution. In this way, when a segment of the binary is required it can be swapped in on demand, or "lazy loaded".

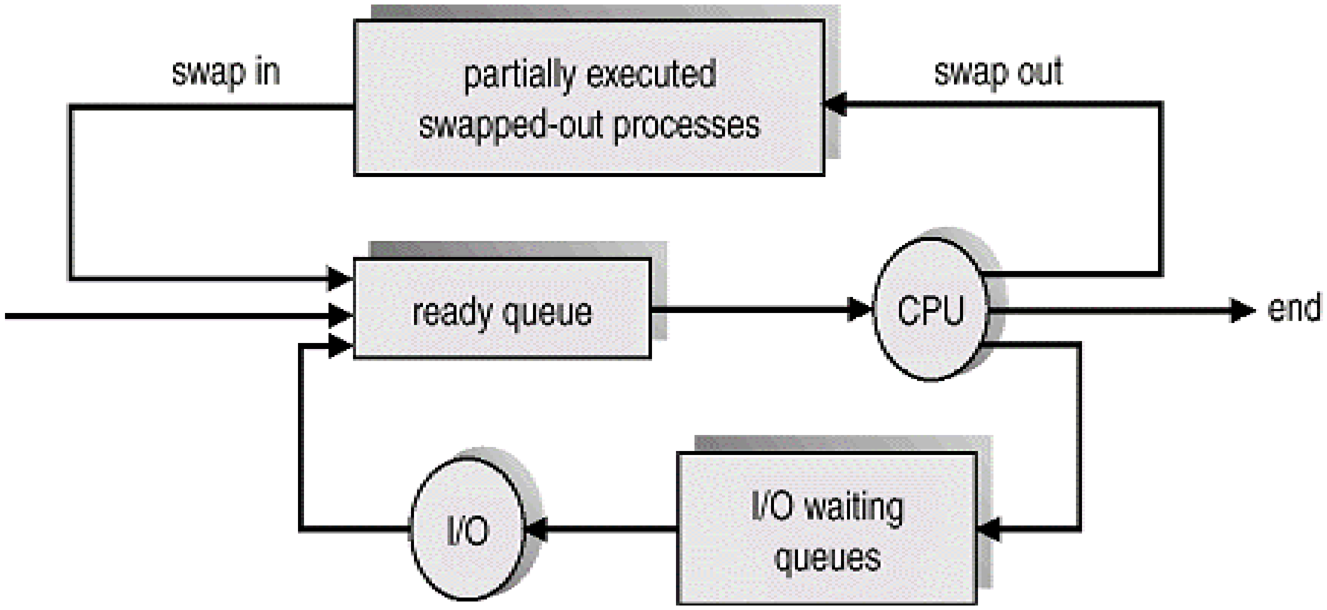

Figure 1.2: Addition of medium-term scheduling to the queuing diagram [2]

c. Short-term scheduler

The short-term scheduler (also known as the CPU scheduler) decides which of the ready, in-memory processes are to be executed (allocated a CPU) next following a clock interrupt, an IO interrupt, an operating system call or another form of signal. Thus the short-term scheduler makes scheduling decisions much more frequently than the long-term or mid-term schedulers - a scheduling decision will at a minimum have to be made after every time slice and these are very short. This scheduler can be preemptive, implying that it is capable of forcibly removing processes from a CPU when it decides to allocate that CPU to another process, or non-preemptive (also known as "voluntary" or "co-operative"), in which case the scheduler is unable to "force" processes off the CPU. In most cases short-term scheduler is written in assembler because it is critical part of operating system.

d. Dispatcher

Another component involved in the CPU-scheduling function is the dispatcher. The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves the following:

- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program

The dispatcher should be as fast as possible, since it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another running is known as the dispatch latency.

1.2 Context Switching

Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a context switch. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the save context of the new process scheduled to run. Context-switch time is pure overhead, because the system does no useful work while switching. Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). Typical speeds are a few milliseconds. Context-switch times are highly dependent on hardware support. For instance, some processors (such as the Sun UltraSPARC) provide multiple sets of registers. A context switch here simply requires changing the pointer to the current register set. Of course, if there are more active processes than there are register sets, the system resorts to copying register data to and from memory, as before. Also, the more complex the operating system, the more work must be done during a context switch. The address space of the current process must be preserved as the space of the next task is prepared for use. How the address space is preserved, and what amount of work is needed to preserve it, depend on the memory-management method of the operating system. Figure 1.3 shows the various process states performing context switching [2].

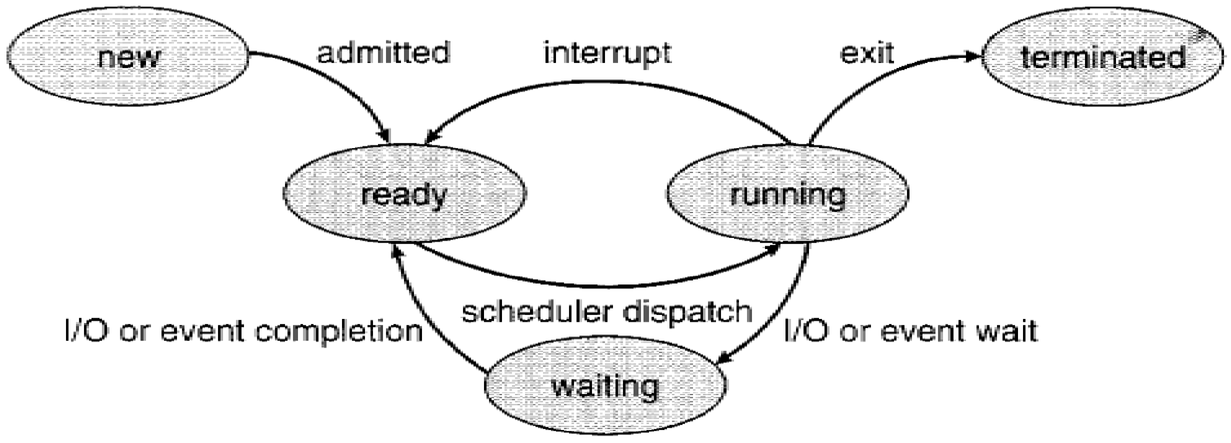

Figure 1.3: The process states [2]

1.3 Linux and Linux Kernel Scheduler

Linux refers to the family of Unix-like computer operating systems using the Linux kernel. Linux can be installed on a wide variety of computer hardware, ranging from mobile phones, tablet computers, etc to mainframes and supercomputers. Linux is a leading server operating system, and runs the 10 fastest supercomputers in the world. The development of Linux is one of the most prominent examples of free and open source software collaboration; typically all the underlying source code can be used, freely modified, and redistributed, both commercially and non-commercially, by anyone under licenses such as the GNU General Public License. Some popular mainstream Linux distributions include Debian (and its derivatives such as Ubuntu), Fedora and openSUSE. Linux distributions include the Linux kernel and supporting utilities and libraries to fulfil the distribution's intended use [3].

The Linux scheduler is an interesting study in competing pressures. On one side are the use models in which Linux is applied. Although Linux was originally developed as a desktop operating system experiment, you'll now find it on servers, tiny embedded devices, mainframes, and supercomputers. Not surprisingly, the scheduling loads for these domains differ. On the other side are the technological advances made in the platform, including architectures (multiprocessing, symmetric multithreading, non-uniform memory access [NUMA]) and virtualization. Also embedded here is the balance between interactivity (user responsiveness) and overall fairness. From this perspective, it's easy to see how difficult the scheduling problem can be within Linux. Details are in chapter-2 [3].

CHAPTER-2

Linux Kernel and Scheduler

2.0 Linux Kernel and Scheduler

2.1 Linux Kernel

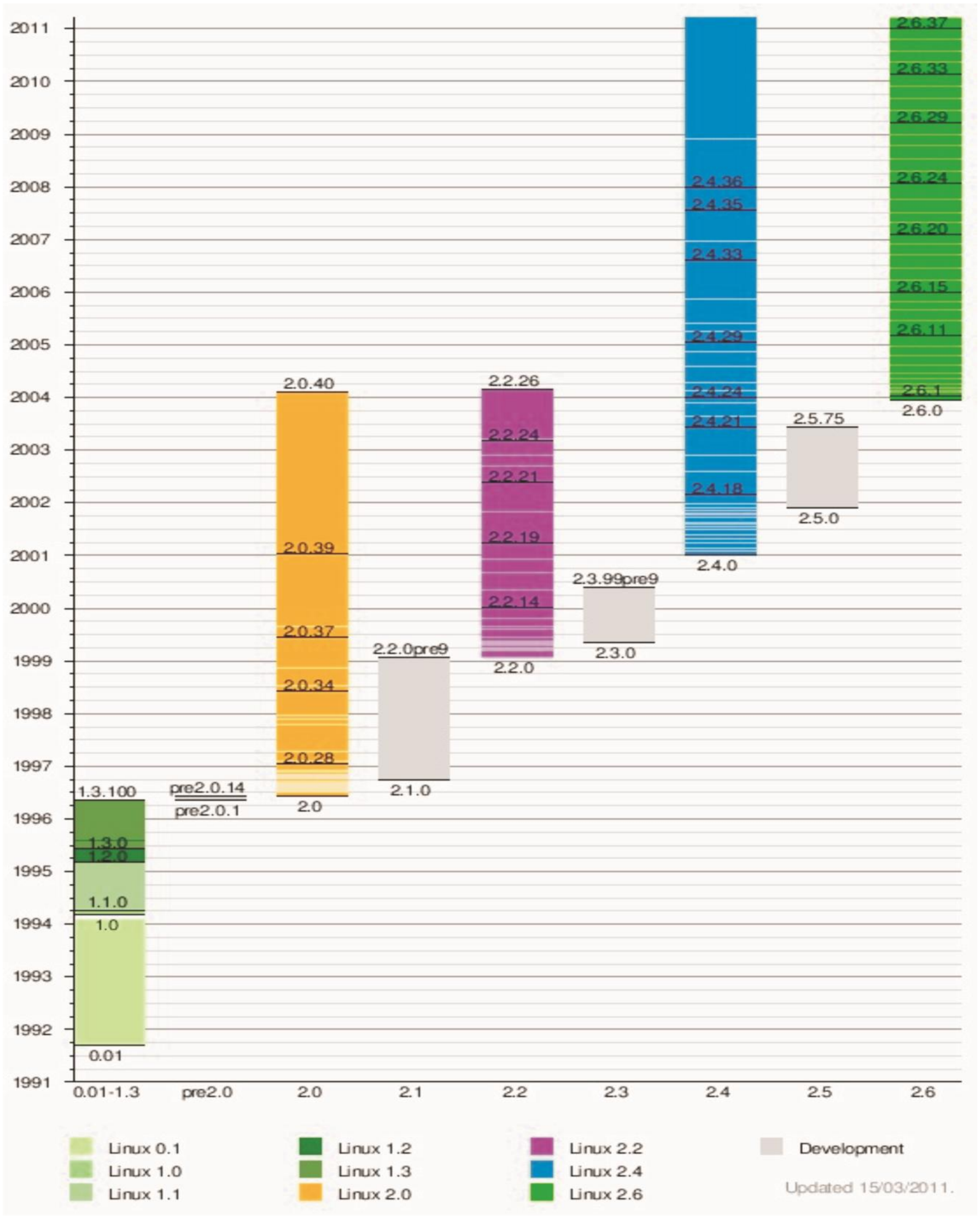

Figure 2.1: Timeline for Linux Versions [4]

The Linux kernel is an operating system kernel used by the Linux family of Unix-like operating systems. It is one of the most prominent examples of free and open source software. The Linux kernel is released under the **GNU General Public License** version 2 (plus some firmware images with various non-free licenses), and is developed by contributors worldwide.

The Linux kernel was initially conceived and created by **Finnish computer science** student **Linus Torvalds** in 1991. Linux rapidly accumulated developers and users who adapted code from other free software projects for use with the new operating system. The Linux kernel has received contributions from thousands of programmers. Many Linux distributions have been released based upon the Linux kernel [4].

The figure 2.1 shows the timeline of Linux versions that has been introduced since 1991.

2.1.1 History of Linux Kernel

While Linux is arguably the most popular open source operating system, its history is actually quite short considering the timeline of operating systems. In the early days of computing, programmers developed on the bare hardware in the hardware's language. The lack of an operating system meant that only one application (and one user) could use the large and expensive device at a time. Early operating systems were developed in the 1950s to provide a simpler development experience. Examples include the General Motors Operating System (GMOS) developed for the IBM 701 and the FORTRAN Monitor System (FMS) developed by North American Aviation for the IBM 709 [5].

In the 1960s, Massachusetts Institute of Technology (MIT) and a host of companies developed an experimental operating system called Multics (or Multiplexed Information and Computing Service) for the GE-645. One of the developers of this operating system, AT&T, dropped out of Multics and developed their own operating system in 1970 called Unics. Along with this operating system was the C language, for which C was developed and then rewritten to make operating system development portable [5].

Twenty years later, Andrew Tanenbaum created a microkernel version of UNIX®, called MINIX (for minimal UNIX), that ran on small personal computers. This open source operating system inspired Linus Torvalds' initial development of Linux in the early 1990s. Figure 2.2 shows major Linux kernel releases [5].

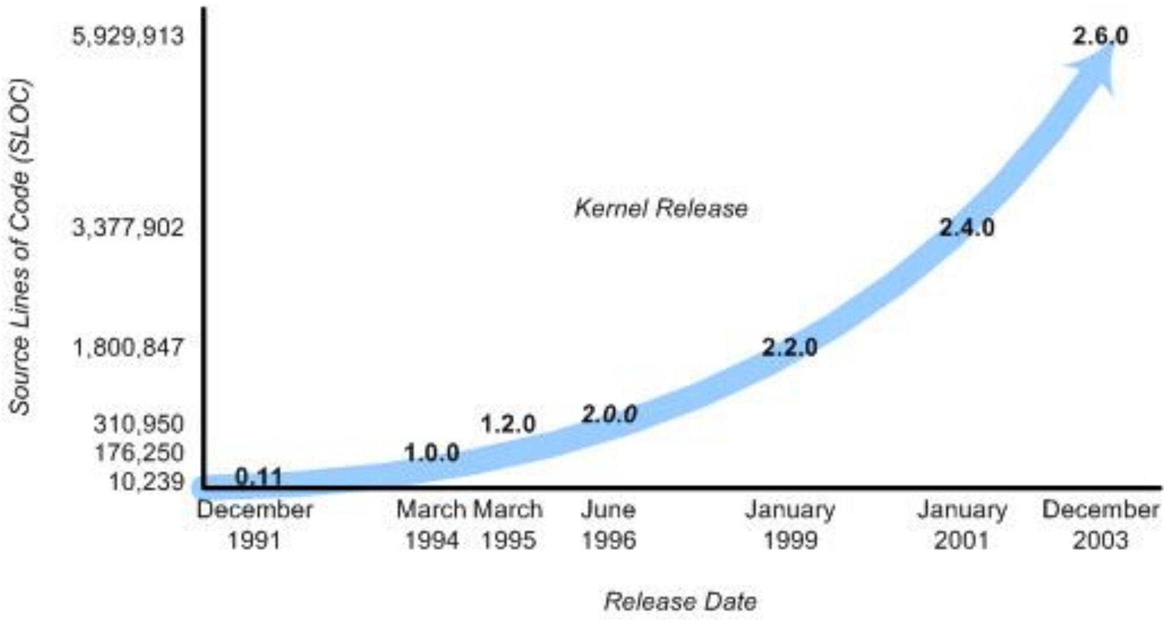

Figure 2.2: A short history of Linux kernel major releases [5]

2.1.2 Overview to Linux kernel

Now on to a high-altitude look at the GNU/Linux operating system architecture. We can think about an operating system from two levels, as shown in figure 2.3.

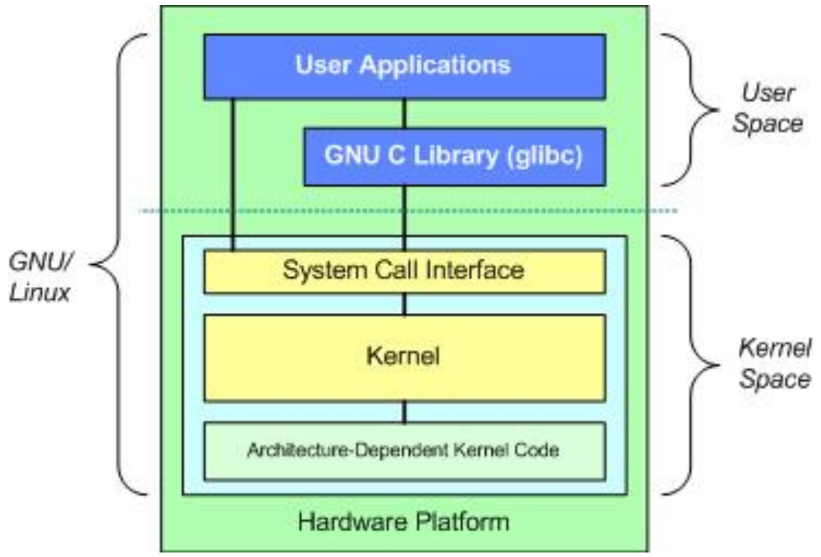

Figure 2.3: The fundamental architecture of Linux operating system [5]

At the top is the user, or application, space. This is where the user applications are executed. Below the user space is the kernel space. Here, the Linux kernel exists.
There is also the GNU C Library (glibc). This provides the system call interface that connects to the kernel and provides the mechanism to transition between the user-space application and the kernel. This is important because the kernel and user application

occupy different protected address spaces. And while each user-space process occupies its own virtual address space, the kernel occupies a single address space.

The Linux kernel can be further divided into three gross levels. At the top is the system call interface, which implements the basic functions such as **read** and **write**. Below the system call interface is the kernel code, which can be more accurately defined as the architecture-independent kernel code. This code is common to all of the processor architectures supported by Linux. Below this is the architecture-dependent code, which forms what is more commonly called a BSP (Board Support Package). This code serves as the processor and platform-specific code for the given architecture [5].

2.2 Linux Scheduler

The Linux scheduler is an interesting study in competing pressures. On one side are the use models in which Linux is applied. Although Linux was originally developed as a desktop operating system experiment, you'll now find it on servers, tiny embedded devices, mainframes, and supercomputers. Not surprisingly, the scheduling loads for these domains differ. On the other side are the technological advances made in the platform, including architectures (multiprocessing, symmetric multithreading, non-uniform memory access [NUMA]) and virtualization. Also embedded here is the balance between interactivity (user responsiveness) and overall fairness. From this perspective, it's easy to see how difficult the scheduling problem can be within Linux.

2.2.1 History of Linux Scheduler

Early Linux schedulers used minimal designs, obviously not focused on massive architectures with many processors or even hyperthreading. The original Linux process scheduler was a simple design based on a goodness() function that recalculated the priority of every task at every context switch, to find the next task to switch to. This served almost unchanged through the 2.4 series, but didn't scale to large numbers of processes, nor to SMP [6]. In 1995 the 1.2 Linux scheduler used a circular queue for runnable task management that operated with a round-robin scheduling policy. This scheduler was efficient for adding and removing processes (with a lock to protect the structure). In short, the scheduler wasn't complex but was simple and fast [7].

Linux version 2.2 (year 1999) introduced the idea of scheduling classes, permitting scheduling policies for real-time tasks, non-preemptible tasks, and non-real-time tasks. The 2.2 scheduler also included support for symmetric multiprocessing (SMP) [7].

By 2001 there were calls for change (such as the OLS paper Enhancing Linux Scheduler Scalability), and the issue came to a head in December 2001. In January 2002, Ingo Molnar introduced the "O(1)" process scheduler for the 2.5 kernel series, a design based on separate "active" and "expired" arrays, one per processor. As the name implied, this found the next task to switch to in constant time no matter how many processes the system was running [6].

Other developers (such as Con Colivas) started working on it, and began a period of extensive scheduler development. The early history of Linux O(1) scheduler development was covered by the website Kernel Traffic.

During 2002 this work included preemption, User Mode Linux support, new drops, runtime tuning, NUMA support, cpu affinity, scheduler hints, 64-bit support, backports to the 2.4 kernel, SCHED_IDLE, discussion of gang scheduling, more NUMA, even more NUMA). By the end of 2002, the O(1) scheduler was becoming the standard even in the 2.4 series.

2003 saw support added for hyperthreading as a NUMA variant, interactivity bugfix, starvation and affinity bugfixes, more NUMA improvements, interactivity improvements, even more NUMA improvements, a proposal for Variable Scheduling Timeouts (the first rumblings of what would later come to be called "dynamic ticks") [6].

In 2004 there was work on load balancing and priority handling, and still more work on hyperthreading.

In 2004 developers proposed several extensive changes to the O(1) scheduler. Linux Weekly News wrote about Nick Piggin's domain-based scheduler and Con Colivas' staircase scheduler. The follow-up article Scheduler tweaks get serious covers both. Nick's scheduling domains were merged into the 2.6 series [6].

The 2.6 scheduler, called the O(1) scheduler, was designed to solve many of the problems with the 2.4 scheduler—namely, the scheduler was not required to iterate the entire task list to identify the next task to schedule (resulting in its name, O(1), which meant that it was much more efficient and much more scalable). The O(1) scheduler kept track of runnable tasks in a run queue (actually, two run queues for each priority level—one for active and one for expired tasks), which meant that to identify the task to execute next, the scheduler simply needed to dequeue the next task off the specific active per-priority run queue. The O(1) scheduler was much more scalable and incorporated interactivity metrics with numerous heuristics to determine whether tasks were I/O-bound or processor-bound. But the O(1) scheduler became unwieldy in the kernel. The large mass of code needed to calculate heuristics was fundamentally difficult to manage and, for the purist, lacked algorithmic substance [7].

2.2.2 Problems with earlier Linux schedulers

O-notation gives you an idea how much time an algorithm will use. The time for an O(n) algorithm depends on the input (linear with n), whereas $O(n^2)$ is quadratic to the input. O(1) is independent of the input and operates in constant time.

Before the 2.6 kernel, the scheduler had a significant limitation when many tasks were active. This was due to the scheduler being implemented using an algorithm with O(n) complexity. In this type of scheduler, the time it takes to schedule a task is a function of the number of tasks in the system. In other words, the more tasks (n) are active, the longer it takes to schedule a task. At very high loads, the processor can be consumed with scheduling and devote little time to the tasks themselves. Thus, the algorithm lacked scalability.

The pre-2.6 scheduler also used a single runqueue for all processors in a symmetric multiprocessing system (SMP). This meant a task could be scheduled on any processor -- which can be good for load balancing but bad for memory caches. For example, suppose a task executed on CPU-1, and its data was in that processor's cache. If the task got rescheduled to CPU-2, its data would need to be invalidated in CPU-1 and brought into CPU-2.

The prior scheduler also used a single runqueue lock; so, in an SMP system, the act of choosing a task to execute locked out any other processors from manipulating the runqueues. The result was idle processors awaiting release of the runqueue lock and decreased efficiency.

Finally, preemption wasn't possible in the earlier scheduler; this meant that a lower priority task could execute while a higher priority task waited for it to complete [7].

2.2.3 Linux 2.6 Scheduler

The 2.6 scheduler was designed and implemented by Ingo Molnar. Ingo has been involved in Linux kernel development since 1995. His motivation in working on the new scheduler was to create a completely O(1) scheduler for wakeup, context-switch, and timer interrupt overhead. One of the issues that triggered the need for a new scheduler was the use of Java™ virtual machines (JVMs). The Java programming model uses many threads of execution, which results in lots of overhead for scheduling in an O(n) scheduler. An O(1) scheduler doesn't suffer under high loads, so JVMs execute efficiently.

The 2.6 scheduler resolves the primary three issues found in the earlier scheduler (O(n) and SMP scalability issues), as well as other problems. Now we'll explore the basic design of the 2.6 scheduler [7].

a. Major scheduling structures

Let's start with a review of the 2.6 scheduler structures. Each CPU has a runqueue made up of 140 priority lists that are serviced in FIFO order. Tasks that are scheduled to execute are added to the end of their respective runqueue's priority list. Each task has a time slice that determines how much time it's permitted to execute. The first 100 priority lists of the runqueue are reserved for real-time tasks, and the last 40 are used for user tasks (see Figure 2.4) [8].

In addition to the CPU's runqueue, which is called the active runqueue, there's also an expired runqueue. When a task on the active runqueue uses all of its time slice, it's moved to the expired runqueue. During the move, its time slice is recalculated (and so is its priority; more on this later). If no tasks exist on the active runqueue for a given priority, the pointers for the active and expired runqueues are swapped, thus making the expired priority list the active one.

The job of the scheduler is simple: choose the task on the highest priority list to execute. To make this process more efficient, a bitmap is used to define when tasks are on a given priority list. Therefore, on most architectures, a find-first-bit-setinstruction is used to find the highest priority bit set in one of five 32-bit words (for the 140 priorities). The time it takes to find a task to execute depends not on the number of active tasks but instead on the number of priorities. This makes the 2.6 scheduler an O(1) process because the time to schedule is both fixed and deterministic regardless of the number of active tasks [8].

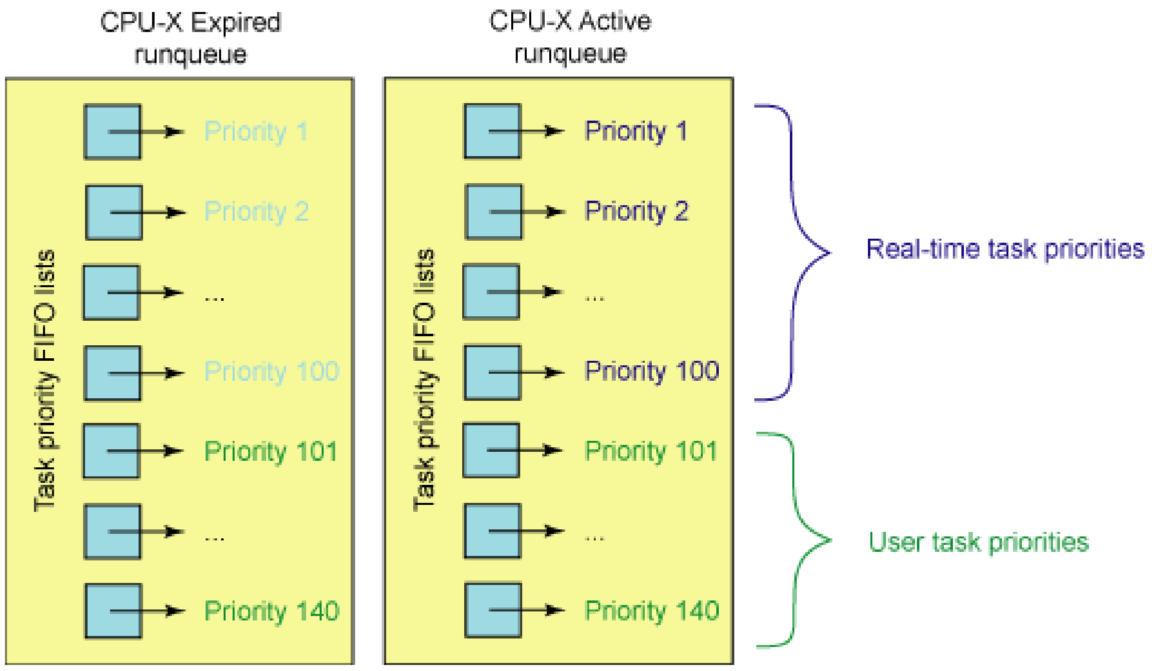

Figure 2.4: The Linux 2.6 scheduler runqueue structure [5]

b. Better support for SMP systems

So, what is SMP? It's an architecture in which multiple CPUs are available to execute individual tasks simultaneously, and it differs from traditional asymmetrical processing in which a single CPU executes all tasks. The SMP architecture can be beneficial for multithreaded applications. Even though the prior scheduler worked in SMP systems, its big-lock architecture meant that while a CPU was choosing a task to dispatch, the runqueue was locked by the CPU, and others had to wait. The 2.6 scheduler doesn't use a single lock for scheduling; instead, it has a lock on each runqueue. This allows all CPUs to schedule tasks without contention from other CPUs. In addition, with a runqueue per processor, a task generally shares affinity with a CPU and can better utilize the CPU's hot cache [8].

c. Task pre-emption

Another advantage of the Linux 2.6 scheduler is that it allows preemption. This means a lower-priority task won't execute while a higher-priority task is ready to run. The scheduler preempts the lower-priority process, places the process back on its priority list, and then reschedules [8].

CHAPTER-3

Completely Fair Scheduler

3.0 Completely Fair Scheduler

3.1 Scheduling Goals

Upon increase in usage of operating systems in computers and portable devices the responsibility for allocating CPU time among tasks is very necessary. CPU Scheduler is responsible for allocating CPU time among tasks. As the most widely used open source operating system, Linux has been widely used in embedded system, desktop and server environment; therefore, there is a need of achieving certain goals as follows:

3.1.1 Fairness and Preventing Starvation

When there is contention for resources, it is important for resources to be allocated fairly. Among scheduling algorithms significant discrepancies may exist in service provided to different flows over the short term. For example, two scheduling algorithms may have the same delay guarantees but can have very different fairness behaviours [9].

There is no commonly accepted method for estimating the fairness of a scheduling algorithm. In general, we would like the system to always serve flows proportional to their reserved rate and distribute the unused bandwidth left behind by idle flows proportionally among active ones. In addition, flows should not be penalized for excess bandwidth they received while other flows were idle [9].

Generalized Processor Sharing (GPS) is an idealized scheduling model which achieves perfect fairness. Lag is commonly used to measure fairness. Let $S_{i,A}(t1,t2)$ denotes the CPU time that task *i* receive in *[t1, t2]* under algorithm A.

Definition-1. *For any interval [t1, t2], the lag of task i at time t ∈[t1, t2] is*

$$lagi\ t = S_{i,GPS}\,(ti\,,t) - S_{i,A}\,(ti\,,t).$$

The smaller the lag is, the fairer the algorithm is. An algorithm is considered strongly fair if its lag is bounded by a small constant. On the contrary, fairness is poor and nonscalable if the lag bound is an O(N) function, where N is the number of tasks, because the algorithm increasingly deviates from GPS as the number of threads in the system increases [11].

It is important for tasks to be treated with a certain degree of fairness, including the stipulation that no thread ever starves. Starvation happens when a thread is not allowed to run for an unacceptably long period of time due to the prioritization of other threads over it. Starvation must not be allowed to happen, though certain threads should be allowed to have a considerably higher priority level than others based on user-defined values and/or heuristic indicators. Somehow, threads that are approaching the starvation threshold (which is generally defined by a scheduler's implementers) must get a significant priority boost or one-time immediate pre-emption before they starve. Fairness does not mean that every thread should have the same degree of access to CPU time with the same priority, but it means that no thread should ever starve or be able to trick the scheduler into giving it a higher priority or more CPU time than it ought to have [10].

3.1.2 Efficiency

An important goal for the Linux scheduler is efficiency. This means that it must try to allow as much real work as possible to be done while staying within the restraints of other requirements. For example – since context switching is expensive, allowing tasks to run for longer periods of time increases efficiency. Also, since the scheduler's code is run quite often, its own speed is an important factor in scheduling efficiency. The code making scheduling decisions should run as quickly and efficiently as possible. Efficiency suffers for the sake of other goals such as interactivity, because interactivity essentially means having more frequent context switches. However, once all other requirements have been met, overall efficiency is the most important goal for the scheduler [10].

3.1.3 Interactivity and Impact on Fairness

Another important feature is good response time for interactive tasks, which always requires small latency. Most schedulers recognize an interactive task based on the assumption that an interactive task tends to sleep frequently. However, fairness should be maintained at the same time. In the case where a scheduler chooses to benefit interactive tasks, when there are multiple interactive tasks in the system, fairness should be kept among the interactive tasks to ensure that no interactive task gets starved by its peers. Besides, non-interactive tasks should not be starved too much by the interactive ones. A program should not steal the CPU time in an unfair way using the reward of interactivity too [11].

3.1.4 Load Balance

Multi-processor architectures, such as Symmetrical Multi-Processing (SMP), Chip Multi-Processor (CMP) and Chip Multi-threading (CMT), are commonly used in embedded areas today to accelerate computation in a parallel way. So load balance becomes an important performance goal, while the core issue is still how to balance tasks among CPUs. A well designed scheduler is required to maintain fairness among CPUs on these multi-processor architectures. Because load of each CPU is unpredictable at any given time, there would be large unfairness among CPUs without load balance. Definition 1 also gives the lag in the multi-processor situation and schedulers are aimed to limit it as small as possible [11].

3.1.5 Real World Application Performance

Now more and more large real world applications can run on high-end embedded environment which is like desktop and server environment today. A good scheduler should maximize throughputs and ensure fairness in this situation and make full use of system resources. In a high-end embedded environment, interactivity is also a significant feature. The scheduler needs to ensure that the interactive tasks have a relative low latency when running together with other background tasks [11].

3.2 Overview of CFS Scheduler

The main idea behind the CFS is to maintain balance (fairness) in providing processor time to tasks. This means processes should be given a fair amount of the processor. When the time for tasks is out of balance (meaning that one or more tasks are not given a fair amount of time relative to others), then those out-of-balance tasks should be given time to execute [12].

To determine the balance, the CFS maintains the amount of time provided to a given task in what's called the virtual runtime. The smaller a task's virtual runtime—meaning the smaller amount of time a task has been permitted access to the processor—the higher its need for the processor. The CFS also includes the concept of sleeper fairness to ensure that tasks that are not currently runnable (for example, waiting for I/O) receive a comparable share of the processor when they eventually need it [12].

But rather than maintain the tasks in a run queue, as has been done in prior Linux schedulers, the CFS maintains a time-ordered red-black tree (see Figure 3.1). A red-black tree is a tree with a couple of interesting and useful properties. First, it's self-balancing, which means that no path in the tree will ever be more than twice as long as any other. Second, operations on the tree occur in O(log n) time (where n is the number of nodes in the tree). This means that you can insert or delete a task quickly and efficiently [12].

With tasks (represented by sched_entity objects) stored in the time-ordered red-black tree, tasks with the gravest need for the processor (lowest virtual runtime) are stored toward the left side of the tree, and tasks with the least need of the processor (highest virtual runtimes) are stored toward the right side of the tree. The scheduler then, to be fair, picks the left-most node of the red-black tree to schedule next to maintain fairness. The task accounts for its time with the CPU by adding its execution time to the virtual runtime and is then inserted back into the tree if runnable. In this way, tasks on the left side of the tree are given time to execute, and the contents of the tree migrate from the right to the left to maintain fairness. Therefore, each runnable task chases the other to maintain a balance of execution across the set of runnable tasks [12].

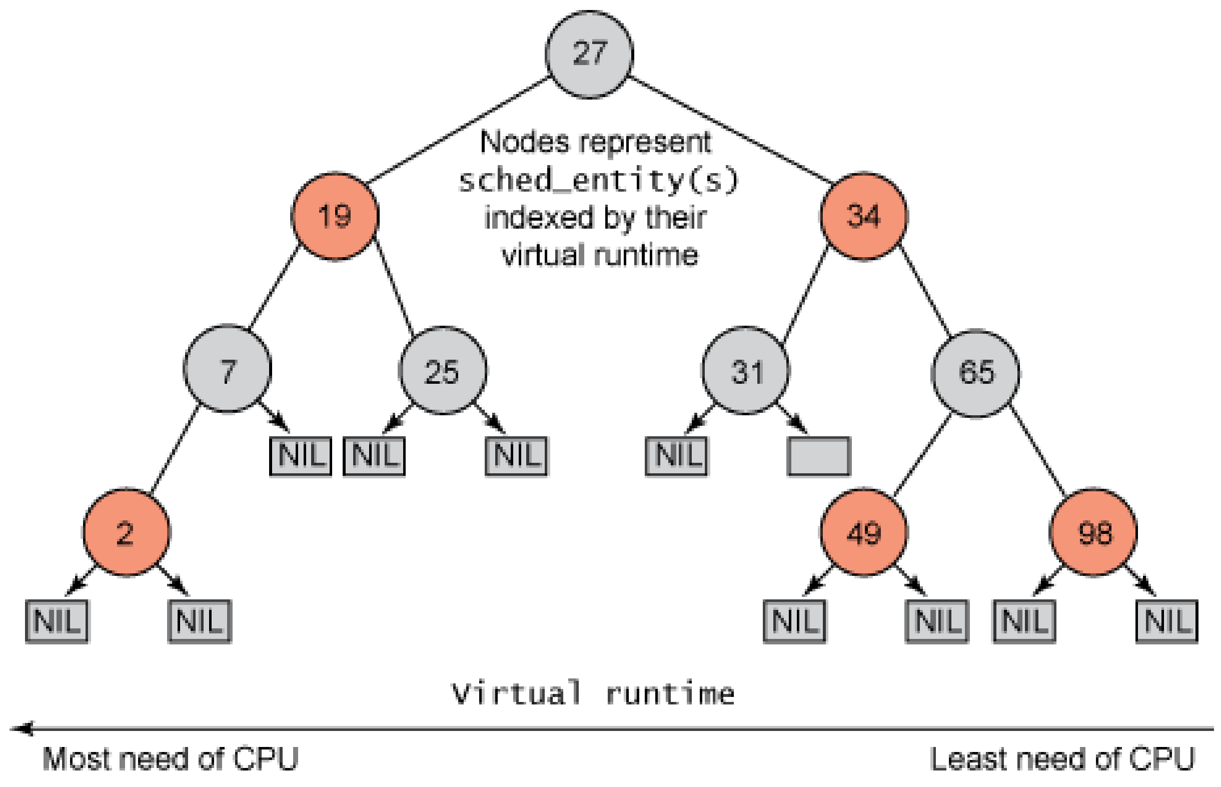

Figure 3.1: Example of a red-black tree [12]

3.3 Red-Black Tree (RBTree)

A red-black tree is a type of self-balancing binary search tree—a data structure typically used to implement associative arrays. It is complex, but it has good worst-case running time for its operations and is efficient in practice. It can search, insert and delete in O(log n) time, where n is the number of elements in the tree. In red-black trees, the leaf nodes are not relevant and do not contain data. These leaves need not be explicit in computer memory—a null child pointer can encode the fact that this child is a leaf—but it simplifies some algorithms for operating on red-black trees if the leaves really are explicit nodes. To save memory, sometimes a single sentinel node performs the role of all leaf nodes; all references from internal nodes to leaf nodes then point to the sentinel node [13].

CFS uses the fair clock and wait runtime to keep all the runnable tasks sorted by the rq->fair_clock - p->wait_runtime key in the rbtree (see the Red-Black Tree sidebar). So, the leftmost task in the tree is the one with the "gravest CPU need", and CFS picks the leftmost task and sticks to it. As the system progresses forward, newly awakened tasks are put into the tree farther and farther to the right—slowly but surely giving every task a chance to become the leftmost task and, thus, get on the CPU within a deterministic amount of time [13].

Because of this simple design, CFS no longer uses active and expired arrays and dispensed with sophisticated heuristics to mark tasks as interactive versus non-interactive.

Ty tune CFS implements priorities by using weighted tasks—each task is assigned a weight based on its static priority. So, while running, the task with lower weight (lower-priority) will see time elapse at a faster rate than that of a higher-priority task. This means its wait_runtime will exhaust more quickly than that of a higher-priority task, so lower-priority tasks will get less CPU time compared to higher-priority tasks.

CFS has been modified a bit further in 2.6.24. Although the basic concept of fairness remains, a few implementation details have changed. Now, instead of chasing the global fair clock (rq->fair_clock), tasks chase each other. A clock per task, vruntime, is introduced, and an approximated average is used to initialize this clock for new tasks. Each task tracks its runtime and is queued in the RBTree using this parameter. So, the task that has run least (the one that has the gravest CPU need) is the leftmost node of the RBTree and will be picked up by the scheduler. See Resources for more details about this implementation [13].

Ty tune In kernel 2.6.24, another major addition to CFS is group scheduling. Plain CFS tries to be fair to all the tasks running in the system. For example, let's say there is a total of 25 runnable processes in the system. CFS tries to be fair by allocating 4% of the CPU to all of them. However, let's say that out of these 25 processes, 20 belong to user A while 5 belong to user B. User B is at an inherent disadvantage, as A is getting more CPU power than B. Group scheduling tries to eliminate this problem. It first tries to be fair to a group and then to individual tasks within that group. So CFS, with group scheduling enabled, will allocate 50% of the CPU to each user A and B. The allocated 50% share of A will be divided fairly among A's 20 tasks, while the other 50% of the CPU time will be distributed fairly among B's 5 tasks.

Scheduling Classes/Modular Scheduler

With kernel 2.6.23, the Linux scheduler also has been made modular. Each scheduling policy (SCHED_FIFO, SCHED_RR, SCHED_OTHER and so on) can be implemented independently of the base scheduler code. This modularization is similar to object-oriented class hierarchies (Figure 3.2).

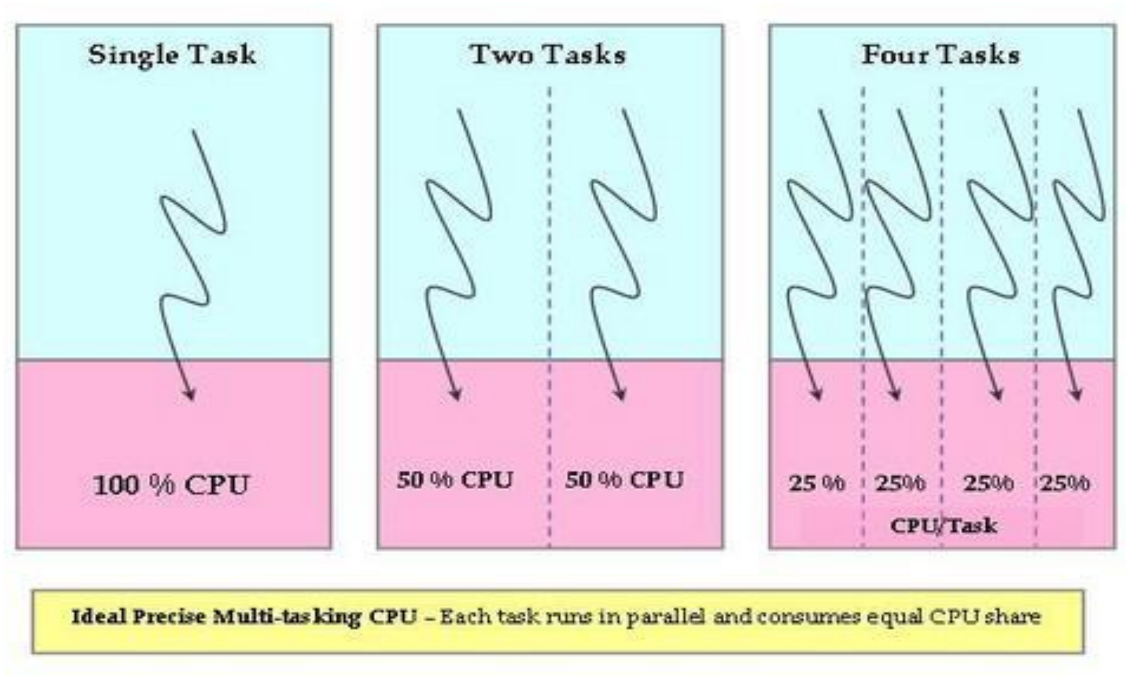
Single Task
Two Tasks
Four Tasks
100 % CPU
50 % CPU
50 % CPU
25 %
25%
25%
25%
CPU/Task
Ideal Precise Multi-tasking CPU – Each task runs in parallel and consumes equal CPU share

Figure 3.2: Modular Scheduler [13]

The core scheduler does not need to be aware of the implementation details of the individual scheduling policies. In kernel 2.6.23, sched.c (the "scheduler" from older kernels) is divided into the following files to make the scheduler modular:

- kernel/sched.c: contains the code of a generic scheduler, thereby exposing functions like sched(). The specific scheduling policy is implemented in a different file.
- kernel/sched_fair.c: this is the main file that implements the CFS scheduler and provides the SCHED_NORMAL, SCHED_BATCH and SCHED_IDLE scheduling policies.
- kernel/sched_rt.c: provides the SCHED_RR and SCHED_FIFO policies used by real-time (RT) threads.

Each of these scheduling policies (fair and RT) registers its function pointers with the core scheduler. The core scheduler calls the appropriate scheduler (fair or RT), based on the scheduling policy of the particular process. As with the O(1) scheduler, real-time processes will have higher priority than normal processes. CFS mainly addresses non-real-time processes, and the RT scheduler remains more or less the same as before (except for a few changes as to how non-active/expired arrays are maintained).

With this new modular scheduler in place, people who want to write new schedulers for a particular policy can do so by simply registering these new policy functions with the core scheduler.

3.4 CFS Internals

All tasks within Linux are represented by a task structure called task_struct. This structure (along with others associated with it) fully describes the task and includes the task's current state, its stack, process flags, priority (both static and dynamic), and much more. You can find this and many of the related structures in ./linux/include/linux/sched.h. But because not all tasks are runnable, you won't find any CFS-related fields in task_struct. Instead, a new structure called sched_entity was created to track scheduling information (see Figure 3.3).

The relationships of the various structures are shown in Figure 2. The root of the tree is referenced via the rb_root element from the cfs_rq structure (in ./kernel/sched.c). Leaves in a red-black tree contain no information, but internal nodes represent one or more tasks that are runnable. Each node in the red-black tree is represented by an rb_node, which contains nothing more than the child references and the color of the parent. The rb_node is contained within the sched_entity structure, which includes the rb_node reference, load weight, and a variety of statistics data. Most importantly, the sched_entity contains thevruntime (64-bit field), which indicates the amount of time the task has run and serves as the index for the red-black tree. Finally, the task_struct sits at the top, which fully describes the task and includes the sched_entity structure [12].

The scheduling function is quite simple when it comes to the CFS portion. In ./kernel/sched.c, you'll find the generic schedule()function, which preempts the currently running task (unless it preempts itself with yield()). Note that CFS has no real notion of time slices for preemption, because the preemption time is variable. The currently running task (now preempted) is returned to the red-black tree through a call to put_prev_task (via the scheduling class). When the schedule function comes to identifying the next task to schedule, it calls the pick_next_task function. This function is also generic (within ./kernel/sched.c), but it calls the CFS scheduler through the scheduler class. The pick_next_task function in CFS can be found in ./kernel/sched_fair.c (called pick_next_task_fair()). This function simply picks the left-most task from the red-black tree and returns the associated sched_entity. With this reference, a simple call to task_of() identifies the task_struct reference returned. The generic scheduler finally provides the processor to this task [12].

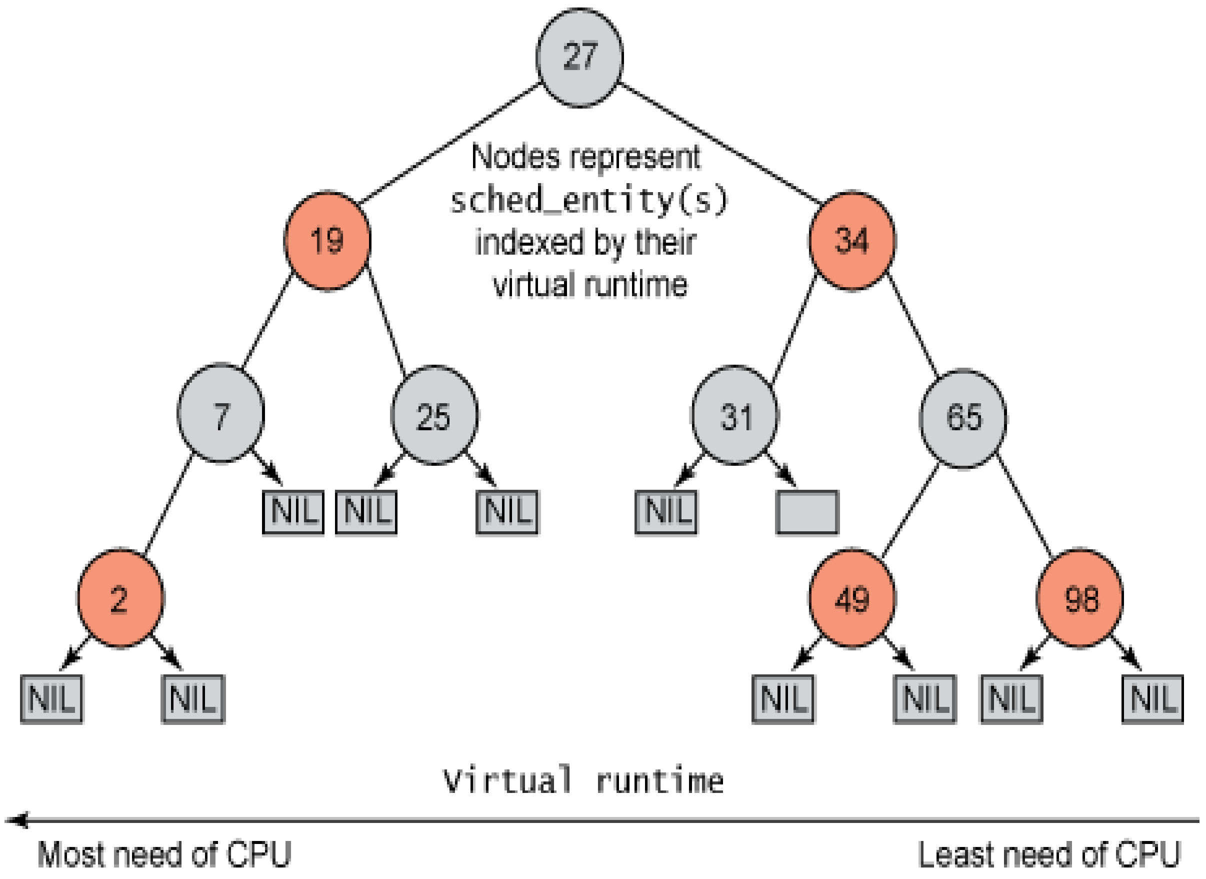

Figure 3.3: Structure hierarchy for tasks and the red-black tree [12]

3.5 Priorities and CFS

CFS doesn't use priorities directly but instead uses them as a decay factor for the time a task is permitted to execute. Lower-priority tasks have higher factors of decay, where higher-priority tasks have lower factors of delay. This means that the time a task is permitted to execute dissipates more quickly for a lower-priority task than for a higher-priority task. That's an elegant solution to avoid maintaining run queues per priority [12].

3.6 CFS group scheduling

Another interesting aspect of CFS is the concept of group scheduling (introduced with the 2.6.24 kernel). Group scheduling is another way to bring fairness to scheduling, particularly in the face of tasks that spawn many other tasks. Consider a server that spawns many tasks to parallelize incoming connections (a typical architecture for HTTP servers). Instead of all tasks being treated fairly uniformly, CFS introduces groups to account for this behavior. The server process that spawns tasks share their virtual runtimes across the group (in a hierarchy), while the single task maintains its own independent virtual runtime. In this way, the single task receives roughly the same scheduling time as the group. You'll find a /proc interface to manage the process hierarchies, giving you full control over how groups are formed. Using this configuration, you can assign fairness across users, across processes, or a variation of each [12].

3.7 Scheduling classes and domains

Also introduced with CFS is the idea of scheduling classes (recall from Figure 3.2). Each task belongs to a scheduling class, which determines how a task will be scheduled. A scheduling class defines a common set of functions (via sched_class) that define the behavior of the scheduler. For example, each scheduler provides a way to add a task to be scheduled, pull the next task to be run, yield to the scheduler, and so on. Each scheduler class is linked with one another in a singly linked list, allowing the classes to be iterated (for example, for the purposes of enablement of disablement on a given processor). The general structure is shown in Figure 3.4. Note here that enqueue and dequeue task functions simply add or remove a task from the particular scheduling structures. The function pick_next_task chooses the next task to execute (depending upon the particular policy of the scheduling class) [12].

But recall that the scheduling classes are part of the task structure itself (see Figure 3.4). This simplifies operations on tasks, regardless of their scheduling class. For example, the following function preempts the currently running task with a new task (where curr defines the currently running task, rq represents the red-black tree for CFS, and p is the next task to schedule) from ./kernel/sched.c:

```
static inline void check_preempt( struct rq *rq, struct task_struct
*p )
{
  rq->curr->sched_class->check_preempt_curr( rq, p );
}
```

If this task were using the fair scheduling class, check_preempt_curr() would resolve to check_preempt_wakeup(). You can see these relationships in ./kernel/sched_rt.c, ./kernel/sched_fair.c and ./kernel/sched_idle.c.

Scheduling classes are yet another interesting aspect of the scheduling changes, but the functionality grows with the addition of scheduling domains. These domains allow you to group one or more processors hierarchically for purposes load balancing and segregation. One or more processors can share scheduling policies (and load balance between them) or implement independent scheduling policies to intentionally segregate tasks [12].

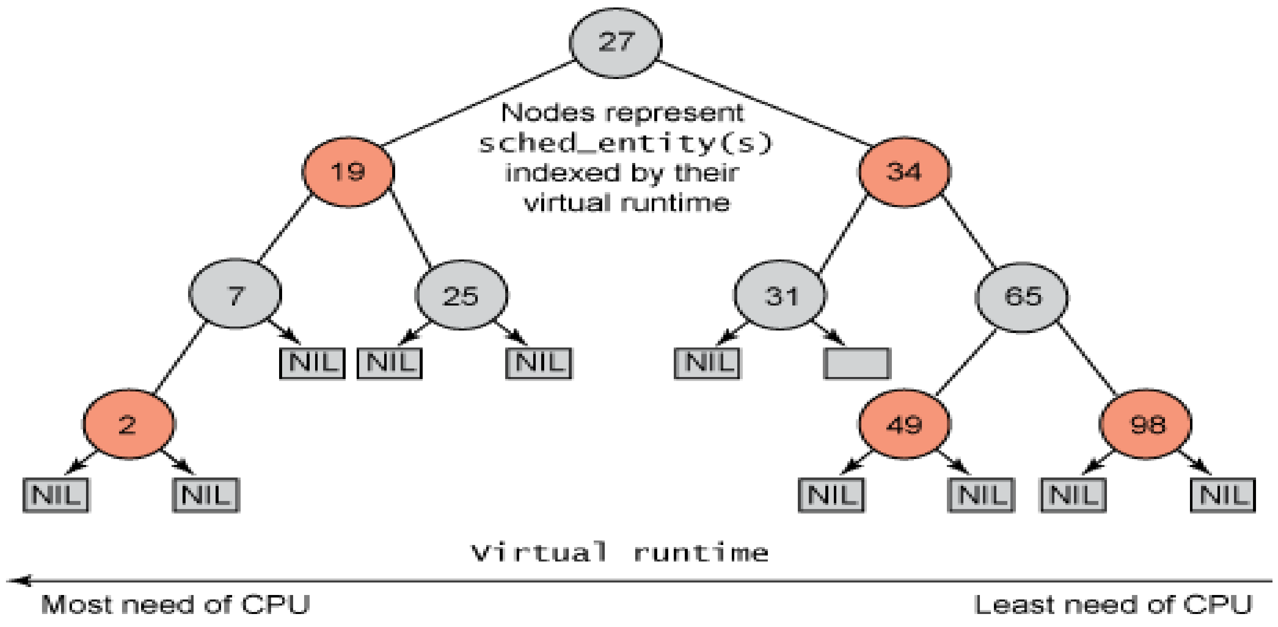

Figure 3.4: Graphical view of scheduling classes [12]

3.8 Multitasking in CFS

Let's try to understand what “ideal, precise, multitasking CPU” means, as the CFS tries to emulate this CPU. An “ideal, precise, multitasking CPU” is a hardware CPU that can run multiple processes at the same time (in parallel), giving each process an equal share of processor power (not time, but power). If a single process is running, it would receive 100% of the processor's power. With two processes, each would have exactly 50% of the physical power (in parallel). Similarly, with four processes running, each would get precisely 25% of physical CPU power in parallel and so on. Therefore, this CPU would be “fair” to all the tasks running in the system (Figure 3.5).

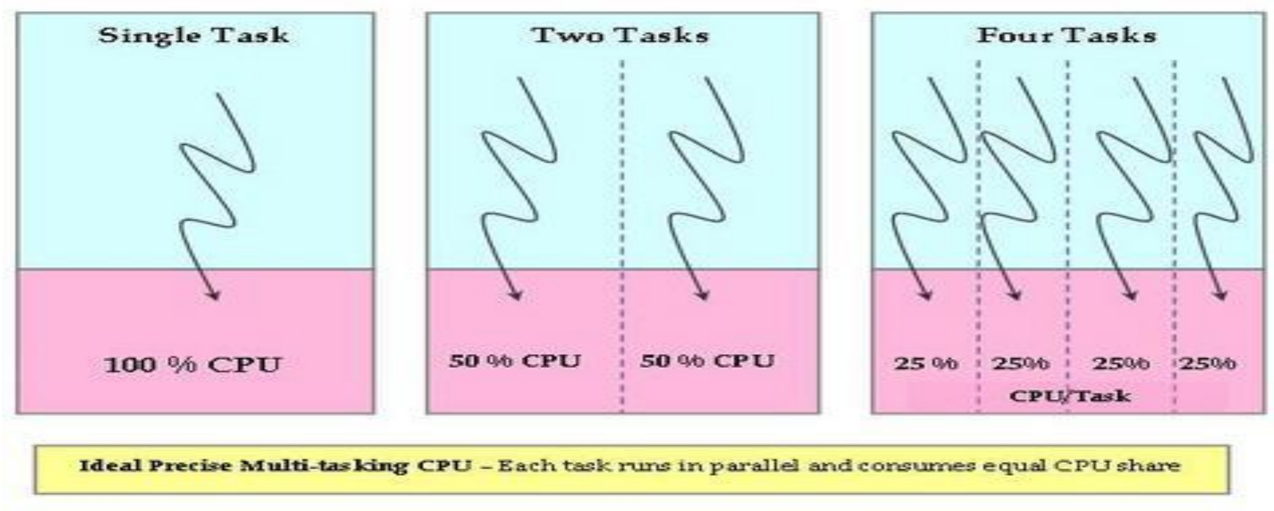
Single Task
100 % CPU
Two Tasks
50 % CPU
50 % CPU
Four Tasks
25 %
25%
25%
25%
CPU/Task
Ideal Precise Multi-tasking CPU – Each task runs in parallel and consumes equal CPU share

Figure 3.5: Ideal, Precise, Multitasking CPU [13]

Obviously, this ideal CPU is nonexistent, but the CFS tries to emulate such a processor in software. On an actual real-world processor, only one task can be allocated to a CPU at a particular time. Therefore, all other tasks wait during this period. So, while the currently running task gets 100% of the CPU power, all other tasks get 0% of the CPU power. This is obviously not fair (Figure 3.6).

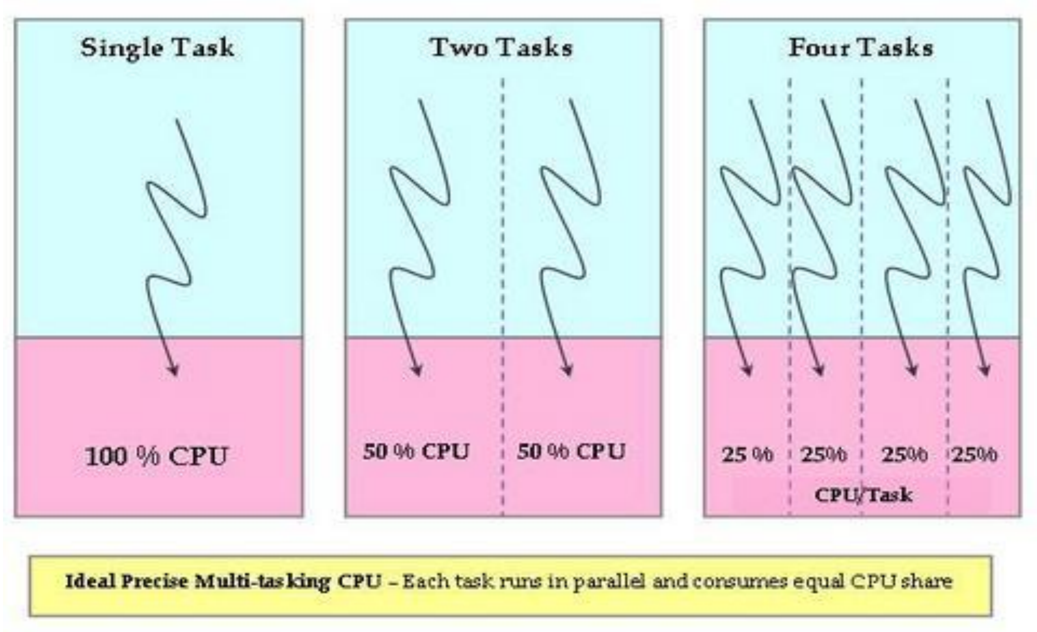
Single Task
Two Tasks
Four Tasks
100 % CPU
50 % CPU
50 % CPU
25 %
25%
25%
25%
CPU/Task
Ideal Precise Multi-tasking CPU – Each task runs in parallel and consumes equal CPU share

Figure 3.6: Actual Hardware CPU [13]

The CFS tries to eliminate this unfairness from the system. The CFS tries to keep track of the fair share of the CPU that would have been available to each process in the system. So, CFS runs a fair clock at a fraction of real CPU clock speed. The fair clock's rate of increase is calculated by dividing the wall time (in nanoseconds) by the total number of processes waiting. The resulting value is the amount of CPU time to which each process is entitled [13].

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, is used to rank processes for scheduling and to determine the amount of time the process is allowed to execute before being preempted. The process with the longest wait time (that is, with the gravest need of CPU) is picked by the scheduler and assigned to the CPU. When this process is running, its wait time decreases, while the time of other waiting tasks increases (as they were waiting). This essentially means that after some time, there will be another task with the largest wait time (in gravest need of the CPU), and the currently running task will be preempted. Using this principle, CFS tries to be fair to all tasks and always tries to have a system with zero wait time for each process—

each process has an equal share of the CPU (something an “ideal, precise, multitasking CPU” would have done) [13].

In order for the CFS to emulate an “ideal, precise, multitasking CPU” by giving each runnable process an equal slice of execution time, CFS needs to have the following:

1. A mechanism to calculate what the fair CPU share is per process. This is achieved by using a system-wide runqueue fair_clock variable (cfs_rq->fair_clock). This fair clock runs at a fraction of real time, so that it runs at the ideal pace for a single task when there are N runnable tasks in the system. For example, if you have four runnable tasks, fair_clock increases at one-fourth of the speed of wall time (which means 25% fair CPU power).

2. A mechanism to keep track of the time for which each process was waiting while the CPU was assigned to the currently running task. This wait time is accumulated in the per-process variable wait_runtime (process->wait_runtime) [13].

4.0 Conclusion

In this report, we have seen that how completely fair scheduler allocates CPU time to various tasks. On real hardware, we can run only a single task at once, so while that one task runs, the other tasks that are waiting for the CPU are at a disadvantage - the current task gets an unfair amount of CPU time i.e. scheduler operates on individual tasks and strives to provide fair CPU time to each task [14]. Sometimes, it may be desirable to group tasks and provide fair CPU time to each such task group. For example, it may be desirable to first provide fair CPU time to each user on the system and then to each task belonging to a user. CONFIG_FAIR_GROUP_SCHED strives to achieve exactly that [14]. Therefore, there is a 'group scheduling extension for CFS' has been proposed by the designers of CFS. It lets tasks be grouped and divides CPU time fairly among such groups.
This group scheduling group tasks according to the user id and thus CPU bandwidth between two users are divided in the ratio of their CPU shares. For example: if you would like user "root" to get twice the bandwidth of user "guest", then set the cpu_share for both the users such that *"root"'s* cpu_share is twice *"guest"'s* cpu_share. Hence, it concludes that completely fair scheduler is advantageous over previous version of Linux schedulers as it provides **group scheduling and multitasking** which was not present in the previous versions of Linux schedulers.

5.0 Future Scope

The Linux kernel can scale to 4096 processors. Not all the processors have to share the same bus architecture and memory. Rather, a machine could consist of multiple 16 to 24 processor systems connected by a high-speed bus. This is the world of NUMA, as each system has its own memory that can be shared with other systems. The Completely Fair Scheduler (CFS) scaled to large NUMA machines, but created problems with systems that had less than 16 cores. CFS also scales well to the high core count of new GPUs (Graphic Processing Unit), which can also run non-graphical processes. For mobile devices with less than 16 cores, it creates problems with heating [15].

In 2.6.26 kernel for Android, CFS caused problems for Google's applications run on Android, built to run on systems handling 5,000 threads on 16 core systems. So Google ported the old O(1) scheduler into their version of 2.6.26 [16].

According to Ingo Molnar noted, "people are regularly testing 3D smoothness, and they find CFS good enough and that matches my experience as well (as limited as it may be). In general my impression is that CFS and SD are roughly on par when it comes to 3D smoothness."

These factors shows that though CFS creates problems for less than 16 core systems, it has a great future as almost all the latest mobile devices and embedded systems are coming with the system configuration that support CFS; therefore, in the upcoming devices and operating systems CFS is going to be used. Even, present Linux operating systems are using CFS due to its ability to allocate equal amount of CPU time to each task. In case of 3D games, CFS has already made a mark, games like war of warcraft are using CFS because CFS provides smoothness in 3D gaming as compared to SD scheduler (previous version of Linux scheduler).

6.0 List of Figures

Figure 1.1: Queuing-diagram representation of process scheduling [2] 4

Figure 1.2: Addition of medium-term scheduling to the queuing diagram [2].................5

Figure 1.3: The process states [2]...7

Figure 2.1: Timeline for Linux Versions [4] 9

Figure 2.2: A short history of Linux kernel major releases [5]................................ 11

Figure 2.3: The fundamental architecture of Linux operating system [5].................... 11

Figure 2.4: The Linux 2.6 scheduler runqueue structure [5] 15

Figure 3.1: Example of a red-black tree [12] 21

Figure 3.2: Modular Scheduler [13]... 23

Figure 3.3: Structure hierarchy for tasks and the red-black tree [12] 25

Figure 3.4: Graphical view of scheduling classes [12] 27

Figure 3.5: Ideal, Precise, Multitasking CPU [13]... 27

Figure 3.6: Actual Hardware CPU [13] 28

7.0 References

[1] http://en.wikipedia.org/wiki/Scheduling_(computing)

[2] Abraham Silberchatz (Yale University), Peter Baer Galvin (Corporate Technologies, Inc.), Greg Gagne (Westminster College)- Operating System Concepts seventh edition.

[3] http://en.wikipedia.org/wiki/Linux

[4] http://en.wikipedia.org/wiki/Linux_kernel

[5] http://www.ibm.com/developerworks/linux/library/l-linux-kernel/

[6] http://www.kernel.org/doc/index-old.html

[7] http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/

[8] http://www.ibm.com/developerworks/linux/library/l-scheduler/

[9] Anton Kos Faculty of Electrical Engineering University of Ljubljana Slovenia, Jelena Miletic´ Faculty of Mathematics University of Belgrade Ministry of Science Stipendist Serbia and Montenegro and Jelena Miletic´ Faculty of Mathematics University of Belgrade Ministry of Science Stipendist Serbia and Montenegro- On Fairness of Deficit Round Robin Scheduler

[10] Josh Aas- 2005 Silicon Graphics, Inc. (SGI)-Understanding the Linux 2.6.8.1 CPU Scheduler, 17th February 2005.

[11] Shen Wang Department of Computer Science and Technology (Tsinghua University Beijing, P.R.China) and Yu Chen Wei Jiang Peng Li Ting Dai Yan Cui Department of Computer Science and Technology Tsinghua University Beijing, P.R.China- IEEE paper 2009-Fairness and Interactivity of Three CPU Schedulers in Linux

[12] http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/

[13] http://www.linuxjournal.com/magazine/completely-fair-scheduler?page=0,2

[14] http://hi.baidu.com/fengzanfeng/blog/item/0566abecd5fde15b79f05528.html

[15] http://www.bargincomputing.com/2010/08/a-good-reason-to-use-pclinuxos/?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+LowCostComputing+%28Low+Cost+Computing%29&utm_content=FeedBurner+user+view

[16] http://www.h-online.com/open/features/Android-versus-Linux-924563.html

8.0 Abbreviations

CPU	Central Processing unit
PCB	Program control Block
JVM	Java virtual Machine
NUMA	Non-Uniform Memory Access
GMOS	General Motor Operating System
FMS	Fortan Monitor System
SMP	Symmetric Multiprocessing
FIFO	First In First Out
GPS	Generalized Processor Sharing
CMP	Chip Multi- processor
CMT	Chip Multi- threading
CFS	Completely fair Scheduler
HTTP	Hypertext Transfer Protocol
SD	Staircase Deadline
I/0	Input/output