- we’ll examine a different type of scheduler known as a proportional-share scheduler, also sometimes referred to as a fair-share scheduler
    - Proportional-share is based around a simple concept: instead of optimizing for turnaround or response time, a scheduler might instead try to guarantee that each job obtain a certain percentage of CPU time
- How can we design a scheduler to share the CPU in a proportional manner? What are the key mechanisms for doing so? How effective are they?

Tickets:

- Underlying lottery scheduling is one very basic concept: tickets, which are used to represent the share of a resource that a process (or user or whatever) should receive.
- The percent of tickets that a process has represents its share of the system resource in question
    - Ex: Imagine two processes, A and B, and further that A has 75 tickets while B has only 25. Thus, what we would like is for A to receive 75% of the CPU and B the remaining 25%.
- Lottery scheduling achieves this probabilistically (but not deterministically) by holding a lottery every so often (say, every time slice).
    - the scheduler must know how many total tickets there are (in our example, there are 100). The scheduler then picks a winning ticket, which is a number from 0 to 99. Assuming A holds tickets 0 through 74 and B 75 through 99, the winning ticket simply determines whether A or B runs. The scheduler then loads the state of that winning process and runs it.

Ticket Mechanisms:

- Lottery scheduling also provides a number of mechanisms to manipulate tickets in different and sometimes useful ways. One way is with the concept of ticket currency.
- Currency allows a user with a set of tickets to allocate tickets among their own jobs in whatever currency they would like; the system then automatically converts said currency into the correct global value
- Another useful mechanism is ticket transfer. With transfers, a process can temporarily hand off its tickets to another process. This ability is especially useful in a client/server setting, where a client process sends a message to a server asking it to do some work on the client’s behalf.
    - To speed up the work, the client can pass the tickets to the server and thus try to maximize the performance of the server while the server is handling the client’s request. When finished, the server then transfers the tickets back to the client and all is as before.
- With ticket inflation, a process can temporarily raise or lower the number of tickets it owns
- due to the randomness of lottery scheduling, sometimes one job finishes before the other. To quantify this difference, we define a simple fairness metric, F which is simply the time the first job completes divided by the time that the second job completes

    - For example, if R = 10, and the first job finishes at time 10 (and the second job at 20), F = 10/20 = 0.5. When both jobs finish at nearly the same time, F will be quite close to 1. In this scenario, that is our goal: a perfectly fair scheduler would achieve F = 1.

Assign Tickets

- how to assign tickets to jobs? One approach is to assume that the users know best; in such a case, each user is handed some number of tickets, and a user can allocate tickets to any jobs they run as desired.

Stride Scheduling

- while randomness gets us a simple (and approximately correct) scheduler, it occasionally will not deliver the exact right proportions, especially over short time scales. For this reason, stride scheduling was invented which is a deterministic fair-share scheduler
- Stride scheduling is also straightforward. Each job in the system has a stride, which is inverse in proportion to the number of tickets it has. In our example above, with jobs A, B, and C, with 100, 50, and 250 tickets, respectively, we can compute the stride of each by dividing some large number by the number of tickets each process has been assigned.
    - For example, if we divide 10,000 by each of those ticket values, we obtain the following stride values for A, B, and C: 100, 200, and 40. We call this value the stride of each process; every time a process runs, we will increment a counter for it (called its pass value) by its stride to track its global progress.
- The scheduler then uses the stride and pass to determine which process should run next. The basic idea is simple: at any given time, pick the process to run that has the lowest pass value so far; when you run a process, increment its pass counter by its stride.
- lottery scheduling has one nice property that stride scheduling does not: no global state.
    - Imagine a new job enters in the middle of our stride scheduling example above; what should its pass value be? Should it be set to 0? If so, it will monopolize the CPU.
    - With lottery scheduling, there is no global state per process; we simply add a new process with whatever tickets it has, update the single global variable to track how many total tickets we have, and go from there

The Linux Completely Fair Scheduler (CFS

- The current Linux approach achieves similar goals in an alternate manner. The scheduler, entitled the Completely Fair Scheduler (or CFS)], implements fairshare scheduling, but does so in a highly efficient and scalable manner.
    - CFS aims to spend very little time making scheduling decisions
- Whereas most schedulers are based around the concept of a fixed time slice, CFS operates a bit differently. Its goal is simple: to fairly divide a CPU evenly among all competing processes. It does so through a simple counting-based technique known as virtual runtime (vruntime).
    - As each process runs, it accumulates vruntime. In the most basic case, each process's vruntime increases at the same rate, in proportion with physical (real)

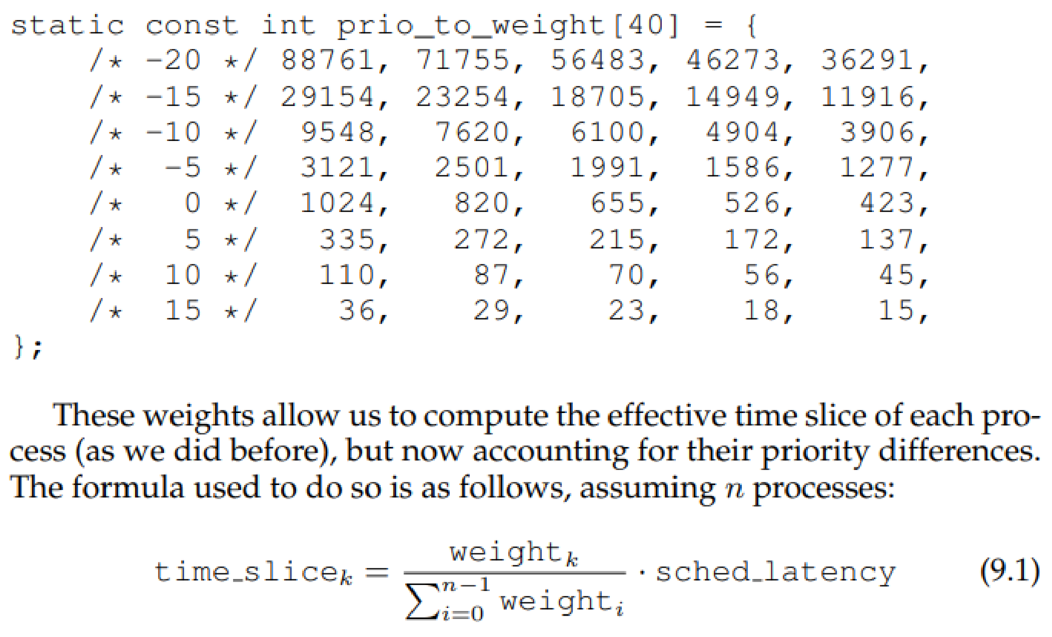
        time. When a scheduling decision occurs, CFS will pick the process with the lowest vruntime to run next.

    - This raises a question: how does the scheduler know when to stop the currently running process, and run the next one?
        - if CFS switches too often, fairness is increased, as CFS will ensure that each process receives its share of CPU even over miniscule time windows, but at the cost of performance (too much context switching);
        - if CFS switches less often, performance is increased (reduced context switching), but at the cost of near-term fairness.
    - CFS manages this tension through various control parameters. The first is scheduled latency. CFS uses this value to determine how long one process should run before considering a switch (effectively determining its time slice but in a dynamic fashion).
        - A typical sched latency value is 48 (milliseconds); CFS divides this value by the number (n) of processes running on the CPU to determine the time slice for a process, and thus ensures that over this period of time, CFS will be completely fair.
        - For example, if there are n = 4 processes running, CFS divides the value of sched latency by n to arrive at a per-process time slice of 12 ms. CFS then schedules the first job and runs it until it has used 12 ms of (virtual) runtime, and then checks to see if there is a job with lower vruntime to run instead.
    - what if there are “too many” processes running? Wouldn’t that lead to too small of a time slice, and thus too many context switches?
        - To address this issue, CFS adds another parameter, min granularity, which is usually set to a value like 6 ms. CFS will never set the time slice of a process to less than this value, ensuring that not too much time is spent in scheduling overhead
        - if there are ten processes running, our original calculation would divide sched latency by ten to determine the time slice (result: 4.8 ms).
            - However, because of min granularity, CFS will set the time slice of each process to 6 ms instead. Although CFS won’t (quite) be perfectly fair over the target scheduling latency (sched latency) of 48 ms, it will be close, while still achieving high CPU efficiency.
    - CFS utilizes a periodic timer interrupt, which means it can only make decisions at fixed time intervals.
    - This interrupt goes off frequently (e.g., every 1 ms), giving CFS a chance to wake up and determine if the current job has reached the end of its run. If a job has a time slice that is not a perfect multiple of the timer interrupt interval, that is OK; CFS tracks vruntime precisely, which means that over the long haul, it will eventually approximate ideal sharing of the CPU.

Weighting (Niceness)

- CFS also enables controls over process priority, enabling users or administrators to give some processes a higher share of the CPU.

- . It does this not with tickets, but through a classic UNIX mechanism known as the nice level of a process. The nice parameter can be set anywhere from -20 to +19 for a process, with a default of 0.
    - Positive nice values imply lower priority and negative values imply higher priority; when you're too nice, you just don't get as much (scheduling) attention, alas.
- CFS maps the nice value of each process to a weight, as shown here:



```
static const int prio_to_weight[40] = {
    /* -20 */ 88761, 71755, 56483, 46273, 36291,
    /* -15 */ 29154, 23254, 18705, 14949, 11916,
    /* -10 */  9548,  7620,  6100,  4904,  3906,
    /*  -5 */  3121,  2501,  1991,  1586,  1277,
    /*   0 */  1024,   820,   655,   526,   423,
    /*   5 */   335,   272,   215,   172,   137,
    /*  10 */   110,    87,    70,    56,    45,
    /*  15 */    36,    29,    23,    18,    15,
};
```

These weights allow us to compute the effective time slice of each process (as we did before), but now accounting for their priority differences. The formula used to do so is as follows, assuming n processes:

$$\texttt{time\_slice}_k = \frac{\texttt{weight}_k}{\sum_{i=0}^{n-1} \texttt{weight}_i} \cdot \texttt{sched\_latency} \qquad (9.1)$$

- Assume there are two jobs, A and B. A, because it's our most precious job, is given a higher priority by assigning it a nice value of -5; B, because we hates it3 , just has the default priority (nice value equal to 0).
    - This means weightA (from the table) is 3121, whereas weightB is 1024. If you then compute the time slice of each job, you'll find that A's time slice is about 3 4 of sched latency (hence, 36 ms), and B's about 1 4 (hence, 12 ms).
- Here is the new formula, which takes the actual run time that process i has accrued (runtimei) and scales it inversely by the weight of the process, by dividing the default weight of 1024 (weight0) by its weight, weighti .
    - vruntimei = vruntimei + (weight0/weighti) · runtimei

Red Black Trees
- One major focus of CFS is efficiency
- when the scheduler has to find the next job to run, it should do so as quickly as possible.
- Simple data structures like lists don't scale: modern systems sometimes are comprised of 1000s of processes, and thus searching through a long-list every so many milliseconds is wasteful.
- CFS addresses this by keeping processes in a red-black tree, a type of balanced tree

- in contrast to a simple binary tree (which can degenerate to list-like performance under worst-case insertion patterns), balanced trees do a little extra work to maintain low depths, and thus ensure that operations are logarithmic (and not linear) in time.
- CFS does not keep all processes in this structure; rather, only running (or runnable) processes are kept therein. If a process goes to sleep (say, waiting on an I/O to complete, or for a network packet to arrive), it is removed from the tree and kept track of elsewhere
- Assume there are ten jobs, and that they have the following values of vruntime: 1, 5, 9, 10, 14, 18, 17, 21, 22, and 24. If we kept these jobs in an ordered list, finding the next job to run would be simple: just remove the first element.
    - However, when placing that job back into the list (in order), we would have to scan the list, looking for the right spot to insert it, an O(n) operation.
    - Processes are ordered in the tree by vruntime, and most operations (such as insertion and deletion) are logarithmic in time, i.e., O(log n). When n is in the thousands, logarithmic is noticeably more efficient than linear.

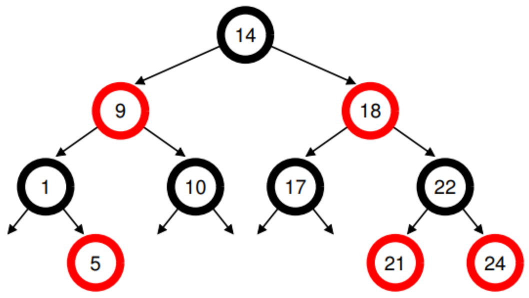


Figure 9.5: **CFS Red-Black Tree**

Dealing With I/O And Sleeping Processes

- One problem with picking the lowest vruntime to run next arises with jobs that have gone to sleep for a long period of time
- Imagine two processes, A and B, one of which (A) runs continuously, and the other (B) which has gone to sleep for a long period of time (say, 10 seconds).
- When B wakes up, its vruntime will be 10 seconds behind A's, and thus (if we're not careful), B will now monopolize the CPU for the next 10 seconds while it catches up, effectively starving A.
- CFS handles this case by altering the vruntime of a job when it wakes up. Specifically, CFS sets the vruntime of that job to the minimum value found in the tree (remember, the tree only contains running jobs)
    - In this way, CFS avoids starvation, but not without a cost: jobs that sleep for short periods of time frequently do not ever get their fair share of the CPU

Summary:

- Lottery uses randomness in a clever way to achieve proportional share; stride does so deterministically. CFS, the only "real" scheduler, is a bit like weighted round-robin with dynamic time slices, but built to scale and perform well under load;
- jobs that perform I/O occasionally may not get their fair share of CPU.
- Another issue is that they leave open the hard problem of ticket or priority assignment, i.e., how do you know how many tickets your browser should be allocated, or to what nice value to set your text editor?

- Other general-purpose schedulers (such as the MLFQ we discussed previously, and other similar Linux schedulers) handle these issues automatically and thus may be more easily deployed.