

CSIS0230A Principles of Operating Systems(Class A)

Notes for Tutorial 8

Peeking into the scheduler

Scheduling different threads to use the CPU is an important system function, too important to allow user programs to arbitrarily manipulate and see directly. That is, in principle. In fact, users sometimes know more than the OS kernel about how a program behaves, and can thus make better decision on how CPU time should be allocated to them. Although we cannot directly see the workings of the scheduler (i.e., the kernel will not “tell” a thread that the CPU is taken away), we can examine how it works **indirectly** by watching the environment of the computer. We will do it during the tutorial, so let’s see the techniques that allows the threads to affect the scheduler and examine the effects.

1. Scheduling policy

Scheduling is a very communist system: the government (OS kernel) completely decides the amount of CPU time enjoyed by each thread, in a way which is more or less fair. But the world is never really fair, as is illustrated by the fiction “Animal Farm” of the British writer *George Orwell*. The last rule in the seven commandments of the communist farm run by animals reads:

ALL ANIMALS ARE EQUAL
BUT SOME ANIMALS ARE MORE EQUAL THAN OTHERS

Task scheduling in Linux has similar rules. All tasks are equal: they all use the same scheduling policy. That is, except a privileged some, which uses scheduling policies that override everybody else, which can be requested by calling the *sched_setscheduler* system call. Its man page has a complete description of all the scheduling policies available to Linux.

The privileged tasks are said to be **real-time**. They are evil: when they are ready, no normal task can run. They are so evil that normal users cannot create such tasks at all—only the `root` user can. By restricting real-time tasks to the `root` user, an arbitrary user cannot lock up the computer by running a real-time task. Each such task has a priority: a real-time task with a higher priority has absolute priority over another real-time task with lower priority. When more than one task has the same priority, the task will give up the CPU after a while if the task uses the RR (Round-Robin) scheduling policy, and will continue until it waits if it uses the FIFO (First-In-First-Out) scheduling policy.

Normal tasks use none of these evil scheduling policies. Instead, they use the time-sharing “`SCHED_OTHER`” policy, which we will examine next.

2. The time-sharing scheduler

Let’s first see the economics of the time-sharing scheduling system. The currency of the system is called **ticks**, each of which can buy around 20 milliseconds of CPU time. Each task has a certain amount of ticks (the “remaining time quantum”) to get service from the CPU. When it is used up, the task will no longer be scheduled at all. When a task is created, half of the ticks of the parent is given up to the child so that it can start working.

When all tasks either used up their ticks or is waiting for events and thus cannot utilize its ticks, no task can run. At this time, the government (OS) allocates new ticks to all time-sharing tasks. We say a new **epoch** starts. The amount of ticks available to each task depends on its “nice-value”. Nice values vary from -20 to 19, with larger values meaning less ticks per epoch (being “nice”, the task doesn’t need as much CPU attention than other tasks). More exactly, the number of ticks

allocated to a task is called its “base time quantum”, and is always $20 - \text{nicevalue}$. E.g., a task with nice value of 5 has base time quantum of $20 - 5 = 15$, while a task with nice value of 19 has base time quantum of $20 - 19 = 1$. Most Linux systems are set up so that when the user logs in the system, the resulting shell process has a nice value of 0. The *nice* and *setpriority* system calls can be used to increase the niceness, and after that all children of that task will have the increased niceness. If you have a command to run and you want to run it with lower priority, you can run it like “`nice -n 10 a.out`”. However, in either cases, only `root` is allowed to decrease the niceness of tasks.

There is a special rule for waiting tasks: suppose an epoch starts when a task is waiting, with non-zero remaining time quantum. Base time quantum will be added to the task. But if no precaution is taken, the number of ticks owned by this task will increase to a huge value after a lot of epochs. When the task finish waiting, it would own so many ticks that all other tasks cannot run. To avoid this, the government takes away half of the ticks it has before allocating its base time quantum, effectively limiting the number of ticks of a task to be less than twice of its base time quantum.

This says about how much time a task can spend, but we still have the question about who to schedule first. Whenever the scheduler is run (e.g., a task needs to wait for an I/O operation or used up its ticks, or a task become ready because the I/O it waits for is completed), a task with positive number of ticks need to be chosen to use the CPU time. Linux uses a very unscientific yet very effective way to select a task: to use the sum of the base time quantum and the remaining time quantum of each task as its priority. E.g., if two tasks are available, one has base time quantum of 10 and remaining time quantum of 1, and the other has base time quantum of 5 and remaining time quantum of 3, then the first one will be executed (since $10 + 1$ is larger than $5 + 3$).

3. Tricks to know more about scheduling

All these are supposed to be invisible by the user. But as somebody learning the OS theory, we do need some way to peek at the scheduler. The information we want to know is (1) at any time, which task is running; and (2) which tasks has used up their ticks, and when ticks are allocated to tasks.

To find an answer to the first question, we need one simple thing: a counter global to the system which allows increment and getting value to be done atomically, without affecting the scheduler. Once we have this tool, different tasks can increment and get values from it. By inspecting the counter values obtained by each task, we can deduce the time (or at least the ordering) when each task executes. Luckily, we do have this device ready: the result of our tutorial 3.

To find an answer to the second question seems more difficult. But actually the Linux scheduling system already provides an answer to it. If a task has a base time quantum of 1, it will also has a remaining time quantum of at most 1 (after the ticks given by its parent is used up). So this task always has the lowest priority, and can be executed only when all other tasks had spent all their ticks (or are waiting). Thus the execution of this task gives a clear mark that a new epoch is about to begin.