

CSIS0230A Principles of Operating Systems(Class A)

Notes for Tutorial 11

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 few, which uses scheduling policies that override everybody else. Such policy 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 limiting real-time tasks to the `root` user, an arbitrary user cannot lock up the computer by running a real-time task (The `root` user can do so, but there are already thousands of ways for them to lock up the system anyway). Each real-time task has a priority from 0 to 99: a smaller priority value meaning higher priority, and no aging is performed. When more than one task has the same priority, preemption will happen only to tasks with the `SCHED_RR` (Round-Robin) scheduling policy, and will not happen to tasks with the `SCHED_FIFO` (First-In-First-Out) scheduling policy.

Normal tasks use neither 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 until it is recharged. When a task is created, half of the ticks of the parent is given up to the child so that the child can start working immediately without waiting for a recharge. Note that under this scheme, even the lowest priority task has a chance to run once a while, without the need of aging.

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) recharge all tasks by set the time quantum to

their **base time quantum**. We say a new **epoch** starts. The base time quantum (i.e., amount of ticks available to each task during one epoch) 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 base time quantum is $20 - \text{nice_value}$. 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 a 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. These system calls are defined as follows:

int nice(int inc)

Adds *inc* to the nice value for the calling task. Returns 0 normally.

int setpriority(int which, int who, int prio)

Set the nice value of a task or a user to *prio*. The *which* argument select between whether to set the nice value of a task (`PRIO_PROCESS`) or all tasks of a user (`PRIO_USER`), while the *who* argument specify the ID of the task or user to change priority. Returns 0 normally.

A new process created by `fork()` will have the same nice level as its parent. 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`”. On the other hand, only `root` is allowed to decrease the niceness of tasks.

This says about how much time a task can spend, but we still have the question about who to schedule first. Each task has a **dynamic priority**, which has a value in the range from 100 to 139. This is larger than the priority value of any real-time task, so they run only when there is no runnable real-time task. “Normally”, a task with nice level from -20 to 19 are given the dynamic priority of 100 to 139 respectively. So a task with lower nice level will run before a task with higher nice level. However, to give better responsiveness to interactive tasks, they are given lower priority values. In particular, the system keep track of the average amount of time that each task sleeps (**sleep average**). For tasks that sleep a lot, the dynamic **priority value is decreased temporarily** (to give it higher priority); and for tasks that sleep very little, the dynamic priority value is increased temporarily. The amount of increase or decrease is never more than 5. Since interactive tasks usually sleeps a lot, this heuristic has the tendency effect of scheduling them before non-interactive tasks.

If a task sleep a real lot, then it is identified as “interactive” by the kernel, and they can extend their time quantum for a short period of time. This allows for occasional CPU bursts of interactive tasks.

Interestingly, all these can be implemented in constant time. This is because there is a fixed number (140) of priorities. The kernel keeps 140 queues for active tasks, one for each priority, and 140 queues for inactive tasks, again one for each priority. Every time the scheduler is invoked, the 140 active queues are looked up to find the first queue that contains a task (constant time, done by finding the first bit set in a bit-vector of 140 bits), and the first task is scheduled. Once the task uses up its time slice, the time quantum is reset, and the task is moved to the corresponding inactive queue. When there is no job in the active queue, the two queues are swapped (constant time again), and a new epoch begins. This is implemented in `kernel/sched.c` of the kernel.

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 4.

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 5 smaller than then any other task, 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.