Team Apol

INTROOS S23 - Nieva, Perez, Pol

3. How is a linux kernel structured?

Linux is limited by hardware functionality. Its OS consists of two separable parts: the system programs and the kernel. The kernel consists of everything below the system-call interface and above the physical hardware, which includes signals terminal handling, character I/O system, terminal drivers, file system, swapping block I/O system, disk and tape drivers, CPU scheduling, page replacement, demand paging and virtual memory. In the kernel, there are five major subsystems which includes the process scheduler (sched), the memory manager (mm), the virtual file system (vfs), the network interface (net), and the inter-process communication (ipc). The kernel provides these and the other operating-system functions, in this case a large number of functions all in one level.

4. What is the role of a kernel scheduler? How do you write a linux kernel scheduler?

Since a kernel or an operating system is expected to run multiple processes, and CPU can only run one instruction at a time, a scheduler is needed to to coordinate which processes should run. The following role of a kernel scheduler are:

1. Provide an equal use of the CPU between currently running processes.
2. Choose an appropriate process to be run next considering the policies, and process priorities implemented.
3. Balance processes between multiple cores.

The scheduler can perform the scheduling of a task in constant time. It is simply a grouping of functions that operate on given data structures. Its job is to pick a new task to run and switch to it. It checks the state of the currently running task and other tasks in a CPU’s run queue to see if scheduling and load balancing is necessary.

A scheduler should not be called during an atomic period. After that, it disables pre-emption and determines the length of time that the task to be scheduled out has been running. After a process has been initialized and placed on a run queue, at some time, it should have access to the CPU to execute. Find the highest-priority queue with a runnable process then find the first process on the queue. Create two functions that are responsible for passing CPU control to different processes which are the scheduler and a system timer that the kernel periodically calls and marks processes as needing rescheduling.

When a timer event occurs, the current process is put on hold and the Linux kernel itself takes control of the CPU. When the timer event finishes, the Linux kernel normally passes control back to the process that was put on hold. However, when the held process has been marked as needing rescheduling, the kernel calls the schedule function and choose which process to activate instead of the process that was executing before the kernel took control.

The time is reduced if a task has high interactivity credit since it would be undesirable for a task that usually waits on I/O to lose interactivity status due to a single long period of CPU usage. Next, if the function is entered off of a kernel pre-emption interruptible tasks with a signal pending get a state of uninterruptible tasks get removed from the queue. This is because if a task can be interrupted and it has a signal pending, it needs to handle that signal. Tasks that are not interruptible should not be on the queue.

At this point, it is time to look for the next task to run. If there are no runnable tasks in the queue, an attempt at load balancing is made. If balancing does not bring any runnable tasks, then a switch to the idle task is made. If there are runnable tasks in the queue but not in the active priority array, then the active and retired priority arrays are swapped.

At this point there is a runnable task in the active priority array. Next, the active priority array’s bitmap is checked to find the highest priority level with a runnable task. After that, dependent sleeping tasks on virtual SMT CPU’s are given a chance to run. If there is a dependent sleeper (which might only happen on an SMT system), the current CPU (which is a virtual CPU sharing physical CPU resources with the virtual CPU that has a dependent sleeper) switches to idle so the dependent sleeper can wake up and do what it needs to do.

If there has not been a switch to the idle task for one reason or another at this point, a check is performed to see if the task chosen to run next is a priority task and has been woken up. If it is not a priority task and was woken up, it is given a slightly higher sleep average and its dynamic priority is recalculated. This is a way to give another small bonus to sleeping tasks. Once this check has been performed and a bonus possible awarded, the wakeup flag is cleared.

Now the schedule function is ready to make an actual task switch. This point in the algorithm is a goto target, and whatever task is pointed to by the next variable is switched to. Earlier decisions to schedule the idle task had simply set next to the idle task and skipped to this point. Here, the previous task has its resched flag cleared, context switch statistical variables are updated, and the previous task gets its run time deducted from its sleep avg. Also, an interactive credit is deducted from the previous task if its sleeping average dropped below 0 and its credit is neither too high nor too low. This is because if its sleeping average is less than 0 it must not have been sleeping very much. With this setup complete, the actual context switch is made so long as the previous task and the new task are not the same task. After the context switch, pre-emption is reenabled since it was disabled during the scheduling algorithm. The final part of the schedule function checks to see if pre-emption was requested during the time in which pre-emption was disabled, and reschedules if it was.

References:

<http://criticalblue.com/news/wp-content/uploads/2013/12/linux_scheduler_notes_final.pdf>

<http://oss.org.cn/ossdocs/linux/kernel/a2/#_Toc411854244>

https://www.cs.columbia.edu/~smb/classes/s06-4118/l13.pdf

http://joshaas.net/linux/linux\_cpu\_scheduler.pdf

http://www.pearsonhighered.com/samplechapter/0131181637.pdf