Kernel Preemption

Last Updated on Fri, 07 Jan 2022 | Linux Kernel Architecture

As described above, the scheduler is invoked before returning to user mode after system calls or at certain designated points in the kernel. This ensures that the kernel, unlike user processes, cannot be interrupted unless it explicitly wants to be. This behavior can be problematic if the kernel is in the middle of a relatively long operation — this may well be the case with filesystem, or memory-management-related tasks. The kernel is executing on behalf of a specific process for a long amount of time, and other processes do not get to run in the meantime. This may result in deteriorating system latency, which users experience as "sluggish" response. Video and audio dropouts may also occur in multimedia applications if they are denied CPU time for too long.

These problems can be resolved by compiling the kernel with support for kernel preemption. This allows not only userspace applications but also the kernel to be interrupted if a high-priority process has some things to do. Keep in mind that kernel preemption and preemption of userland tasks by other userland tasks are two different concepts!

Kernel preemption was added during the development of kernel 2.5. Although astonishingly few changes were required to make the kernel preemptible, the mechanism is not as easy to implement as preemption of tasks running in userspace. If the kernel cannot complete certain actions in a single operation — manipulation of data structures, for instance — race conditions may occur and render the system inconsistent. The same problems arise on multiprocessor systems discussed in Chapter 5.

The kernel may not, therefore, be interrupted at all points. Fortunately, most of these points have already been identified by SMP implementation, and this information can be reused to implement kernel preemption. Problematic sections of the kernel that may only be accessed by one processor at a time are protected by so-called spinlocks: The first processor to arrive at a dangerous (also called critical) region acquires the lock, and releases the lock once the region is left again. Another processor that wants to access the region in the meantime has to wait until the first user has released the lock. Only then can it acquire the lock and enter the dangerous region.

If the kernel can be preempted, even uniprocessor systems will behave like SMP systems. Consider that the kernel is working inside a critical region when it is preempted. The next task also operates in kernel mode, and unfortunately also wants to access the same critical region. This is effectively equivalent to two processors working in the critical region at the same time and must be prevented. Every time the kernel is inside a critical region, kernel preemption must be disabled.

How does the kernel keep track of whether it can be preempted or not? Recall that each task in the system is equipped with an architecture-specific instance of struct thread_info. The structure also includes a preemption counter:

<asm-arch/thread_info.h>

struct thread_info {

int preempt_count; /* 0 => preemptable, <0 => BUG */

The value of this element determines whether the kernel is currently at a position where it may be interrupted. If preempt_count is zero, the kernel can be interrupted, otherwise not. The value must not be manipulated directly, but only with the auxiliary functions dec_preempt_count and inc_preempt_count, which, respectively, decrement and increment the counter. inc_preempt_count is invoked each time the kernel enters an important area where preemption is forbidden. When this area is exited, dec_ preempt_count decrements the value of the preemption counter by 1. Because the kernel can enter some important areas via different routes — particularly via nested routes — a simple Boolean variable would not be sufficient for preempt_count. When multiple dangerous regions are entered one after another, it must be made sure that all of them have been left before the kernel can be preempted again.

The dec_preempt_count and inc_preempt_count calls are integrated in the synchronization operations for SMP systems (see Chapter 5). They are, in any case, already present at all relevant points of the kernel so that the preemption mechanism can make best use of them simply by reusing the existing infrastructure.

Some more routines are provided for preemption handling:

▫ preempt_disable disables preemption by calling inc_preempt_count. Additionally, the compiler is instructed to avoid certain memory optimizations that could lead to problems with the preemption mechanism.

▫ preempt_check_resched checks if scheduling is necessary and does so if required.

▫ preempt_enable enables kernel preemption, and additionally checks afterward if rescheduling is necessary with preempt_check_resched.

▫ preempt_disable_no_resched disables preemption, but does not reschedule.

At some points in the kernel, the protection by the normal SMP synchronization methods is not sufficient. This happens, for instance, when per-CPU variables are modified. On a real SMP system, this requires no form of protection because only one processor can by definition operate with the variable — every other CPU in the system has its own instance and does not need to fiddle with the instance of the current processor. However, kernel preemption would allow that two different code paths on the same processor would access the variable quasi-concurrently, which would have the same result as if two independent processors would manipulate the value. Preemption must therefore be explicitly disabled in these situations using manual incovations of preempt_disable and preempt_disable.

Note, however, that the get_cpu and put_cpu functions mentioned in the Introduction will automatically disable kernel preemption, so no extra precautions are necessary if per-CPU variables are accessed using this mechanism.

How does the kernel know if preemption is required? First of all, the tif_need_resched flag must be set to signalize that a process is waiting to get CPU time. This is honored by preempt_ check_resched:

<preempt.h>

if (unlikely(test_thread_flag(TIF_NEED_RESCHED))) \ preempt_schedule(); \

Recall that the function is called when preemption is re-enabled after it had been disabled, so this is a good time to check if a process wants to preempt the currently executing kernel code. If this is the case, it should be done as quickly as possible — without waiting for the next routine call of the scheduler.

The central function for the preemption mechanism is preempt_schedule. The simple desire that the kernel be preempted as indicated by tif_need_resched does not yet guarantee that this is possible — recall that the kernel could currently still be inside a critical region, and must not be disturbed. This is checked by preempt_reschedule:

kernel/sched.c asmlinkage void _sched preempt_schedule(void)

struct thread_info *ti = current_thread_info();

* If there is a non-zero preempt_count or interrupts are disabled,

* we do not want to preempt the current task. Just return..

if (unlikely(ti->preempt_count || irqs_disabled())) return;

If the preemption counter is greater than 0, then preemption is still disabled, and consequently the kernel may not be interrupted — the function terminates immediately. Neither is preemption possible if the kernel has disabled hardware IRQs at important points where processing must be completed in a single operation. irqs_disabled checks whether interrupts are disabled or not, and if they are disabled, the kernel must not be preempted.

The following steps are required if preemption is possible:

kernel/sched.c do {

add_preempt_count(PREEMPT_ACTIVE); schedule();

sub_preempt_count(PREEMPT_ACTIVE);

* Check again in case we missed a preemption opportunity

* between schedule and now.

} while (unlikely(test_thread_flag(TIF_NEED_RESCHED)));

Before the scheduler is invoked, the value of the preemption counter is set to preempt_active. This sets a flag bit in the preemption counter that has such a large value that it is never affected by the regular preemption counter increments as illustrated by Figure 2-30. It indicates to the schedule function that scheduling was not invoked in the normal way but as a result of a kernel preemption. After the kernel has rescheduled, code flow returns to the current task — possibly after some time has elapsed, because the preempting task will have run in between — the flag bit is removed again.

Preemption counter

PREEMPT_ACTIVE

Continue reading here: Figure 230 The perprocess preemption counter