FAQ/Preemption

Preemption under Linux

What is preemption ?

Wikipedia explains that the preemption is « The ability of the operating system to preempt or stop a currently scheduled task in favour of a higher priority task. The scheduling may be one of, but not limited to, process or I/O scheduling etc. »

Under Linux ?

Under Linux, user-space programs have always been preemptible : the kernel interrupts user-space programs to switch to other threads, using the regular clock tick. So, the kernel doesn't wait for user-space programs to explicitly release the processor (which is the case in cooperative multitasking). This means that an infinite loop in an user-space program cannot block the system.

However, until 2.6 kernels, the kernel itself was not preemtible : as soon as one thread has entered the kernel, it could not be preempted to execute an other thread. The processor could be used to execute another thread when a syscall was terminated, or when the current thread explictly asked the scheduler to run another thread using the `schedule()` function. This means that an infinite loop in the kernel code blocked the entire system, but this is not really a problem : the kernel code is designed so that there are no infinite loops.

However, this absence of preemption in the kernel caused several problems with regard to latency and scalability. So, kernel preemption has been introduced in 2.6 kernels, and one can enable or disable it using the `CONFIG_PREEMPT` option. If `CONFIG_PREEMPT` is enabled, then kernel code can be preempted everywhere, except when the code has disabled local interrupts. An infinite loop in the code can no longer block the entire system. If `CONFIG_PREEMPT` is disabled, then the 2.4 behaviour is restored.

Spinlocks and preemption

Under early 2.4 kernels, on an uni-processor machines, spinlocks operations were simply no-ops, because there was no preemption. Strictly speaking, protection against concurrent accesses was needed only for multi-processor machines (of course, only if the shared resource is not used at the same time by an interrupt handler). However, now that kernel code is preemptible in 2.6, spinlocks are no more no-ops, even on uni-processor machines.

So, your code that may worked correctly under 2.4 on an uni-processor machine will fail to work correctly on a 2.6 with preemption enabled if you didn't correctly took into account concurrency problems. Anyway, when coding in the kernel, you should not take care of the fact that the machine is uni-processor, multi-processor or whether the preemption is enabled or not. Simply use the Linux spinlocks, they'll do the correct job and they'll be as optimized as possible for the destination architecture.