

Overview

- We look at sources of concurrency in the kernel
- We examine one source in particular, kernel preemption, (support introduced in Linux 2.6)
- Along the way we take a look at a selection of the locking facilities available to Linux kernel programmers to handle concurrency issues in the kernel



Concurrency in the Kernel

- The following are sources of concurrency in the kernel:
 1. Interrupt handling
 2. Preemption (since 2.6 the kernel is preemptible)
 3. SMP (true concurrency when we have multiple CPUs)
- All kernel code must take care that race conditions do not arise in this concurrent environment



UP Concurrency

- On a UP we need to worry about concurrency due to:
 1. Interrupt handling
 2. Preemption
- What must we do if we wish to update a data structure that is also referenced by an interrupt handler?
- What would happen if we simply put a spinlock or semaphore around the data structure and the nasty interrupt occurred?



Preemption

- It is a process's responsibility to put itself to sleep e.g. after issuing an I/O request that may take some time
- The process marks itself as waiting, places itself on a wait queue, removes itself from the runqueue and calls `schedule` to have another task run
- When the requested event occurs the processes on the wait queue are woken and test if the condition upon which they were waiting is now true
- If false the task returns to sleep (spurious wake-ups)
- If true the task is removed from the wait queue, placed on the runqueue and marked runnable



Preemption

- If the newly runnable task is of higher priority than the currently executing one, a flag, `need_resched`, is set and preemption will occur at the next available opportunity
- `need_resched` is also set when a process runs out of timeslice (set by `clock_tick`)
- Why not immediately preempt the running process?
- Even if our code is updating a kernel resource not acted upon by any interrupt handler we are not safe: preemption may run another task accessing the same resource
- We could disable interrupts but that's a heavy handed approach since interrupt handlers are not the problem
- Instead the programmer uses `preempt_disable` and `preempt_enable` to handle the problem



Kernel Preemption

- In a non-preemptible kernel (Linux used to work like this), kernel code must run to completion and a context switch can only occur:
 1. When a task explicitly calls `schedule` (putting itself to sleep)
 2. On returning to userland from an interrupt handler or system call
- In a preemptible kernel it is up to the kernel programmer to let the kernel know when it is safe to preempt: through calls to `preempt_disable` and `preempt_enable`
- These calls atomically increment and decrement a counter, `preempt_count`



Kernel Preemption

- When a call to `preempt_enable` makes the counter zero and `need_resched` is set, the current process is preempted
- Kernel preemption occurs when:
 1. Returning to kernel code from an interrupt handler (provided that `preempt_count` is zero) and `need_resched` is set
 2. The kernel code becomes preemptible again (`preempt_count` hits zero on a call to `preempt_enable`) and `need_resched` is set
 3. `schedule` is explicitly called



Userland Preemption

- It is always safe to preempt a task (and the `need_resched` flag is checked) when a process returns to userland
- A process returns to userland from:
 1. A system call
 2. An interrupt handler
- Clearly once execution has left the kernel a process can be safely preempted (it cannot be updating any kernel resources)



UP Concurrency Summary

- On a UP there are basically two sources of concurrency to worry about:
 1. Interrupts
 2. Preemption
- These can be handled by:
 1. Disabling and enabling interrupts
 2. `preempt_disable` and `preempt_enable`
- The second option is preferred when kernel code accesses a shared resource that is not accessed by an interrupt handler



SMP Concurrency

- The kernel is also required to run on SMP machines and there true concurrency becomes reality
- New locking mechanisms must be introduced to solve concurrency related issues
- Enabling/disabling interrupts on one CPU will not prevent interrupt handlers executing on another CPU from accessing the shared resource
- Enabling/disabling preemption on one CPU will not prevent processes executing on another CPU from accessing the shared resource
- As a result spinlocks and semaphores are added to the kernel locking options



Spinlocks and Semaphores

- Spinlocks provide a simple, fast, non-recursive locking mechanism for the kernel
- When a spinlock is unavailable the caller spins until the lock becomes available
- Spinlocks should be held only for very short periods of time and are therefore suitable for use only across short code segments
- Where locking is required for longer periods a semaphore should be used since lock contention will result in a process sleeping rather than uselessly spinning for a prolonged period



Spinlocks and Semaphores

- Care must be exercised by the programmer. . .
- She wishes to access a shared resource on an SMP machine and protects that resource with only a spinlock
- If that resource is also accessed by an interrupt handler then the following can happen:
 1. Process acquires spinlock
 2. Interrupt occurs, handler is invoked and process is suspended
 3. The handler executes in the context of the process that holds the lock and therefore spins waiting for the lock to become available
 4. We have deadlock because the only process that can release the spinlock is held up by the interrupt handler that will never return



Spinlocks and Semaphores

- What if we simply disabled interrupts on the CPU?
- To solve the latter problem the acquiring process must disable interrupts and acquire a spinlock
- Now the interrupt may still occur on the other CPU:
 1. The spinlock is unavailable and execution spins in the interrupt handler
 2. However, the process that owns the spinlock is not held up and will soon release the spinlock allowing the interrupt handler to return
- If the resource is never accessed by an interrupt handler then only the spinlock is required
- Clearly, programming in this environment is not straightforward!