CMPSCI 577 Operating Systems Design and Implementation **Spring 2020**

Lecture 20: Apr 09, 2020

Lecturer: ***Ahmed Ali-Eldin*** *Scribe:* ***Pavan Reddy Bommana***

20.1 Need for Synchronization

Synchronization is needed in order to ensure that the shared resources are protected from concurrent access. The presence of multiprocessors will give rise to parallel execution, which needs synchronization. The presence of critical regions in the code and race conditions between two threads of execution is another reason.

20.1.1 Kernel example of race condition

A kernel queue with tasks is implemented as a linked list with two functions manipulating the queue. A Queue function that adds new tasks to the end of the queue. A dequeue function that removes the head of the queue for processing. Now imagine a situation where two processors are trying to queue two new tasks with no atomicity. The two processors are trying to make changes to the same resource by using queue function (critical region). This gives rise to a race condition.

20.1.2 Locking

Locking in the kernel is advisory and voluntary. Locking will help us ensure the atomicity of execution so that no race condition exists. Linux has a handful of locking mechanisms implemented that a kernel programmer can use. The behavior when the lock is unavailable because another thread already holds it is the most significant difference among different locking mechanisms. For example, busy-wait and sleep. Locks are implemented using atomic operations to ensure no race exists within the lock itself.

(i) Test-and-set: Used in most architectures. Tests if a lock is acquired. If so, then it it will exit either busy waiting or sleeping. In bot, then it sets the lock. It executes as an atomic instruction in the kernel.

(ii) Compare-and-exchange: Present in X86 architecture.

20.1.3 Causes of concurrency issues in kernel

1) Interrupts — An interrupt can occur asynchronously at almost any time, interrupting the currently executing code.

2) Softirqs and tasklets — The kernel can raise or schedule a softirq or tasklet at almost any time, interrupting the currently executing code.

3) Kernel preemption — Because the kernel is preemptive, one task in the kernel can preempt another.

4) Sleeping and synchronization with user-space — A task in the kernel can sleep and thus invoke the scheduler, resulting in the running of a new process.

5) Symmetrical multiprocessing — Two or more processors can execute kernel code at exactly the same time.

20.2 Kernel synchronization methods

20.2.1 Atomic Kernel operations

The group of operations which are guaranteed, by the kernel, to happen with atomicity. These operations either fully happen or do not happen at all. These can be divided into two types - i) Integer atomic kernel operations ii) Bitwise atomic kernel operations

20.2.1.1 Integer Atomic Kernel operations

These are special methods that guarantee atomicity for addition, reading variables etc., These methods work on some special data types called atomic data types (example *atomic_t*).

20.2.1.2 Bitwise Atomic Kernel operations

These are bunch of methods in the kernel that help in changing different bits in the memory or in data structures atomically. They can operate on any memory address. This allows the user to safely perform bitwise operations on the memory. The arguments to these functions are a pointer and a bit number. Bit zero is the least significant bit of the given address.
For each atomic bitwise operation, there is a non-atomic equivalent that can be used. Non-atomic operation is generally faster than atomic operation. So it is recommended to use the non-atomic operation if a lock already protects the data. The non-atomic operation starts with double underscores (example non-atomic version of test_bit() is __test_bit()).

20.2.2 Kernel Spin locks

A spin lock is a lock that can be held by at most one thread of execution. If a thread of execution attempts to acquire a spin lock while it is already held, the thread busy loops waiting for the lock to become available. If the waiting time is less than two context switches, then hold a spin lock, as a rule of thumb. Spin locks are mostly ssed in interrupt handlers after disabling local interrupts.

20.2.2.1 Reader-Writer Spin locks

In many operations, we have two different paths of operation, one of them is for readers and one of them is for writers. So lock usage can be clearly divided into reader and writer paths in these cases. Writing demands mutual exclusion, reading does not. Reading can happen concurrently as long as there are no writers trying to write. If there are multiple readers, then no writer should be able to write because readers should finish reading before writer starts writing. Only one writer can write at a time. These reader-writer spin locks are also called shared/exclusive or concurrent/exclusive locks sometimes.

20.2.3 Semaphores

Semaphores in Linux are sleeping locks. If a semaphore is not available, then the task sleeps in the wait queue. Once semaphore available, the kernel wakes up one of these sleeping tasks and tries to run it. Semaphores used for locks held for a long time. There are two types of Semaphores in the kernel - (i) Binary or mutex semaphore (ii) Counting semaphore (mostly initialized with count of one in the kernel)

20.2.3.1 Mutex

Only one task can hold the mutex at a time. You cannot lock a mutex in one context and then unlock it in another. This means that the mutex isn't suitable for more complicated synchronizations between kernel and user-space. Recursive locks and unlocks are not allowed. That is, you cannot recursively acquire the same mutex, and you cannot unlock an unlocked mutex. A process cannot exit while holding a mutex. A mutex cannot be acquired by an interrupt handler or bottom half.

20.2.3.2 Differences between Spin locks and Semaphores

Spin lock is preferred when low overhead locking or short lock hold time is the requirement. Mutex is preferred for longer lock holding time requirement. Spin lock is required when the requirement is to lock from interrupt context. Mutex is required when the requirement is to sleep while holding lock.

20.2.4 Completion Variables

It is basically another way to build a semaphore like locking mechanism. This is much more user-friendly than mutex or semaphore. One task waits on the completion variable while another task performs some work. When the other task has completed the work, it uses the completion variable to wake up any waiting tasks.

20.2.5 Big Kernel Lock (BKL)

It is a historical and obselete locking mechanism. The Big Kernel Lock is no longer part of Linux. It was a global spin-lock.

20.2.6 Sequential Locks

These locks are useful to provide a lightweight and scalable lock for use when we have many readers and a few writers. One drawback of conventional reader-writer lock is that they tend to lead to starvation for writers. Sequential locks are designed to mitigate this drawback. This works by maintaining a sequence counter. Whenever the data in question is written to, a lock is obtained by the writer and a sequence number is incremented. Here also, no two writers can acquire a lock together. The sequence counter starts at zero. Grabbing the write lock makes the value odd whereas releasing it makes it even (counter is incremented every time lock is acquired or released) because the lock starts at zero. Prior to and after reading the data, the sequence number is read. If the values are the same, a write did not begin in the middle of the read. If the values are even, a write is not underway.

20.3 Timers and Time Management

20.3.1 Kernel Notion of Time

The hardware provides a system timer that the kernel uses to gauge the passing of time. This system timer works off of an electronic time source, such as a digital clock or the frequency of the processor. This is architecture dependent. The system timer goes off (often called hitting or popping) at a pre-programmed frequency, called the tick rate. When the system timer goes off, it issues an interrupt which is called interrupt timer. Interrupt timer occurs every tick and has a special interrupt handler. This is important for many kernel functions, specially scheduling.

20.3.2 Tick Rate: HZ

The tick rate is programmed on system boot based on a static preprocessor define, HZ. It is important to note that, when writing kernel code, never assume that HZ has any given value as it is architecture dependant and also a user can change it any time.

20.3.2.1 HZ Tradeoffs

Pros of Higher values: Higher tick rate means higher the frequency of the timer interrupt. Consequently, the work it performs occurs more often. The accuracy of timed events improves. Process preemption occurs more accurately.

Cons of Higher values: A higher tick rate implies more frequent timer interrupts, which implies higher overhead. This means, the processor must spend more time executing the timer interrupt handler. This leads to more frequent thrashing of the processor's cache. This also leads to increased power consumption.

20.3.3 Tickless OS

In tickless OS mode, the system dynamically schedules the timer interrupt in accordance with pending timers. Instead of firing the timer interrupt at a fixed frequency, the interrupt is dynamically scheduled and rescheduled as needed. This saves a lot of power on idle systems.

20.3.4 Jiffies

A global variable holding the number of ticks that have occurred since the system booted. The system uptime can be calculated by using the expression jiffies/HZ seconds. Jiffies variable has data type an unsigned long. It is 32 bits in size on 32-bit architectures and 64-bits on 64-bit architectures. This 32 bit jiffies variable will cause overflow in a very short time if HZ is high. This causes problems in long running servers. It is to be noted that the overflow will not lead to a system crash but the kernel needs to do additional work to make sure that the overflow does not screw up the normal system operations. So even for 32-bit architectures, we create a 64-bit jiffies variable by gluing together two 32-bit memory locations. Kernel correctly handles the wrap-around for this 64-bit jiffies variable created.

20.3.5 Hardware clocks and Timers

20.3.5.1 The RTC

The real-time clock (RTC) provides a nonvolatile device for storing the system time. RTC continues to keep track of time even when the system is off by way of a small battery included on the system board. On the PC architecture, the RTC and the CMOS are integrated, and a single battery keeps the RTC running and the BIOS settings preserved. The kernel reads the RTC and uses it to initialize the wall time, which is stored in the xtime variable.

20.3.5.2 The System Timer

The system timer provides a mechanism for driving an interrupt at a periodic rate. This is architecture dependent. Different architectures implement system timer differently. Some architectures implement this via an electronic clock that oscillates at a programmable frequency. On x86, the primary system timer is the programmable interrupt timer (PIT). The PIT exists on all PC machines and has been driving interrupts since the days of DOS. The kernel programs the PIT on boot to drive the system timer interrupt (interrupt zero) at HZ frequency.

20.3.6 Timer Interrupt Handler

The timer interrupt is broken into two pieces: an architecture-dependent and an architecture-independent routine. The architecture dependant part does at least the following functions -

1) Obtain the xtime_lock lock, which protects access to jiffies_64 and the wall time value, xtime.

2) Acknowledge or reset the system timer as required.

3) Periodically save the updated wall time to the real time clock.

4) Call the architecture-independent timer routine, tick_periodic().

The architecture-independent routine, tick_periodic(), performs the following -

1) Increment the jiffies_64 count by one.

2) Update resource usages, such as consumed system and user time, for the currently running process.

3) Run any dynamic timers that have expired.

4) Execute scheduler_tick().

5) Update the wall time, which is stored in xtime.

6) Calculate the load average.

20.3.7 Dynamic Kernel Timers

Timers—sometimes called dynamic timers or kernel timers—are essential for managing the flow of time in kernel code. The workflow for these timers is as follows -

1) Initialize timer setup with an expiration time.

2) Specify a function to execute upon said expiration.

3) Activate the timer.

4) The given function runs after the timer expires.

5) Timers are not cyclic and the timer is destroyed after it expires.

20.3.7.1 Timers Implementation

Timers are stored in a linked list. To optimize insertions and realizing which timers expired, the kernel partitions timers into five groups based on their expiration value. Timers move down through the groups as their expiration time draws closer. The partitioning ensures that, in most executions of the timer softirq, the kernel has to do little work to find the expired timers.

20.3.8 Other methods of delaying execution

Sometimes there is a need for a short delay, or a simpler way to induce delays. The kernel provides three functions for microsecond, nanosecond, and millisecond delays, which do not use jiffies. These functions are *void udelay(unsigned long usecs)*, *void ndelay(unsigned long nsecs)*. *void mdelay(unsigned long msecs)* respectively.