Synchronization in Kernel

Pradipta De
pradipta.de@sunykorea.ac.kr
Today’s Topic

• Background review on Concurrency
• How does Linux tackle concurrency?
  – By synchronizing process/thread execution
What is synchronization?

• What is synchronization?
  – Processes/threads run in time-shared manner (single CPU) or on multiple CPUs (SMP)
    ➔ how to coordinate the operation such that we always arrive at deterministic result

  – Process can be scheduled out at any point in time
    • Interrupts can arrive any time
    • Should one disable interrupt while touching a shared resource?

  – Linux kernel is preemptible
    • But kernel preemption can be disabled
Concurrency and its Problem

• Concurrent execution means simultaneous execution (could be the same code block)
  – True concurrency: execution on multiple CPUs on multi-processor systems
  – Pseudo concurrency: time shared execution (on a uni-processor system), but processes run as if they are concurrent – multiple virtual CPUs

• Challenges
  – Modify shared variables or data structures
  – Wait to access shared resources
Definitions

Assume n processes (could be the same program or different program accessing some shared variable) active simultaneously

• **Critical Section**: portion of the code where shared data is manipulated
• **Race Condition**: if more than one process manipulates shared data simultaneously, leading to indeterminate output
• **Mutual Exclusion**: If one process is executing in critical, no other process should enter its critical section
Synchronization Scenarios

• Necessary
  – Computation outcome depends on how two kernel control paths are interleaved
    • Critical sections in exception handlers, interrupt handlers, kernel threads, deferrable functions
  – Examples:
    • Two different interrupt handlers access same data structure
      – Disable interrupts
    • Service routines of system calls access same data structure
      – Disable kernel preemption
Synchronization Scenarios

• Not Necessary
  – Interrupt handler disables the IRQ line
  – Interrupt handlers are non-preemptible
  – A kernel thread servicing interrupt cannot be preempted by system call routines, or deferrable functions
  – Softirqs and tasklets cannot interleave on a single CPU, and same tasklet cannot run on multiple CPUs
## Synchronization Techniques

<table>
<thead>
<tr>
<th>Technique</th>
<th>Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Per-CPU variables</td>
<td>Duplicate a data structure among the CPUs</td>
<td>All CPUs</td>
</tr>
<tr>
<td>Atomic operation</td>
<td>Atomic read-modify-write instruction to a counter</td>
<td>All CPUs</td>
</tr>
<tr>
<td>Memory barrier</td>
<td>Avoid instruction reordering</td>
<td>Local CPU or All CPUs</td>
</tr>
<tr>
<td>Spin lock</td>
<td>Lock with busy wait</td>
<td>All CPUs</td>
</tr>
<tr>
<td>Semaphore</td>
<td>Lock with blocking wait (sleep)</td>
<td>All CPUs</td>
</tr>
<tr>
<td>Seqlocks</td>
<td>Lock based on an access counter</td>
<td>All CPUs</td>
</tr>
<tr>
<td>Local interrupt disabling</td>
<td>Forbid interrupt handling on a single CPU</td>
<td>Local CPU</td>
</tr>
<tr>
<td>Local softirq disabling</td>
<td>Forbid deferrable function handling on a single CPU</td>
<td>Local CPU</td>
</tr>
<tr>
<td>Read-copy-update (RCU)</td>
<td>Lock-free access to shared data structures through pointers</td>
<td>All CPUs</td>
</tr>
</tbody>
</table>
Atomic Execution and Locks

- Execute critical section without interruption
- Implement atomic execution
  - Use a lock before entering critical section
  - Unlock after completing critical section
- Locks are a technique to override scheduling

- Evaluate locks
  - Mutual exclusion – basic requirement
  - Fairness – no starvation among threads contending for the same lock
  - Performance – time overheads

- Need Hardware support to implement locking primitives
  - Test and set instruction
  - Compare and exchange (on x86)
Optimization and Memory Barrier

- Optimizing compilers can reorder instructions
- When executing on multiple processors, memory accesses can be reordered

Synchronization primitives act as optimization and memory barriers

- An *optimization barrier* primitive ensures that the assembly language instructions corresponding to C statements placed before the primitive are not mixed by the compiler with assembly language instructions corresponding to C statements placed after the primitive.

- A *memory barrier* primitive ensures that the operations placed before the primitive are finished before starting the operations placed after the primitive.
Spinlock

Resources: C1, C2, C3
Processes: P0, P1, P2, P3, P4

Suitable for multiprocessor environments

• What is the problem with spinlocks?
  • Spinlock is a busy-wait lock ➔ wastes CPU cycles
  • Works fairly well in multi-processor env, but wasteful in uni-processor env
  • Kernel preemption is disabled in critical sections protected by spinlocks
  • Can the process in “busy wait” get preempted?
Spinlocks do not distinguish Read and Read/Write access. Multiple Read can proceed in parallel, the Write must be one at a time.

Multiple readers are allowed access to the shared resource. Only one writer is allowed access to shared resource. Writer must explicitly access the rw-spinlock.
Sequential Lock (seqlock)

• Request to acquire read and write lock have equal priority \(\Rightarrow\) if read locks are held, then write cannot proceed
• Can we allow write to proceed while reads are in progress, and then fix the reads?
• Seqlock allows writer to proceed when readers are active
  – Reader may be forced to read same data multiple times to get valid value

• Design of Seqlock:
  – Maintain a sequence counter
  – When data is written to, acquire lock and increment a sequence counter
  – Before and after reading data, sequence number is checked to ensure that write did not start in between
  – Seq number starts from 0
    • Even seq number implies write is not in progress (seq no starts from 0)
  – If a writer holds the lock, then reader cannot enter
  – Only one writer can hold the lock
• What is a semaphore?
  – A semaphore is an object, initialized to an integer value, and provides two routines: acquire (wait) and release (post)

```c
int sem_wait(sem_t *s) {
    wait until value of semaphore s is greater than 0
    decrement the value of semaphore s by 1
}

int sem_post(sem_t *s) {
    increment the value of semaphore s by 1
    if there are 1 or more threads waiting, wake 1
}
```
Semaphore

- What does semaphore achieve?
  - A process can sleep instead of busy waiting for a lock → better CPU utilization if the lock is held for a long period

- Binary Semaphore
  - Initialize to 1 (similar to spinlock)
  - Allows only one process to hold the lock
  - In Linux, called as “mutex”

- Counting Semaphore
  - Initialize to non-zero value greater than 1

Since semaphore allows a thread to sleep, interrupt handlers and deferrable functions cannot use them
Read/write Semaphore

- Similar to RW-spinlocks, except that the process/thread waiting for the lock is suspended.
- Writer has exclusive access; multiple readers can have simultaneous access.
- Kernel handles all processes waiting for read/write semaphores in a FIFO order.
Read-Copy-Update

- Allows many readers and many writers to proceed simultaneously
  - Improvement over seqlock
- RCU is a lock-free data structure
  - No shared lock or counter

- When can it be used?
  - Protect dynamically allocated data structures and referenced using pointers (Why ?)
  - Process cannot sleep inside critical region protected by RCU
Read-Copy-Update

• How it works:
  – Reader gets the lock (disables interrupt), dereferences pointer and starts reading till it finishes (no sleep in between), finally release lock (enable interrupt)
  – Writer dereferences the pointer, and makes a copy of the whole data structure, then modifies copy; changes the pointer to updated copy
    • When can writer change the pointer to updated copy ?

• Complications
  – Readers may be reading the old copy ➔ writer must wait before changing the pointer
  – Waits till each CPU performs a process switch (calls schedule) [note: readers cannot sleep in between]
Local Interrupt Disable

• Disable interrupt to ensure that a sequence of kernel statements are treated as critical section
  – Does not protect against concurrent access to data structures by interrupt handlers running on other CPUs
  – Couple local interrupt disabling with spinlocks
## Choosing Synchronization Primitives

<table>
<thead>
<tr>
<th>Kernel control paths accessing the data structure</th>
<th>UP protection</th>
<th>MP further protection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exceptions</td>
<td>Semaphore</td>
<td>None</td>
</tr>
<tr>
<td>Interrupts</td>
<td>Local interrupt disabling</td>
<td>Spin lock</td>
</tr>
<tr>
<td>Deferrable functions</td>
<td>None</td>
<td>None or spin lock (see Table 5-9)</td>
</tr>
<tr>
<td>Exceptions + Interrupts</td>
<td>Local interrupt disabling</td>
<td>Spin lock</td>
</tr>
<tr>
<td>Exceptions + Deferrable functions</td>
<td>Local softirq disabling</td>
<td>Spin lock</td>
</tr>
<tr>
<td>Interrupts + Deferrable functions</td>
<td>Local interrupt disabling</td>
<td>Spin lock</td>
</tr>
<tr>
<td>Exceptions + Interrupts + Deferrable functions</td>
<td>Local interrupt disabling</td>
<td>Spin lock</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Deferrable functions accessing the data structure</th>
<th>Protection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Softirqs</td>
<td>Spin lock</td>
</tr>
<tr>
<td>One tasklet</td>
<td>None</td>
</tr>
<tr>
<td>Many tasklets</td>
<td>Spin lock</td>
</tr>
</tbody>
</table>
Avoiding Deadlocks

Solution: Order locking of resources
No automatic technique ➔ programmer agreed upon convention
Big Kernel Lock

- A global kernel lock
- Linux 2.0: Spinlock ensuring that only one processor at a time could run in Kernel mode
- Linux 2.2, 2.4: A large number of kernel data structures protected by different spinlocks
- Linux 2.6: BKL protects old code
Putting It Together

- Managing concurrent access to shared resource by kernel threads on platforms with
  - Single CPU
  - Multiple CPUs