Synchronization in Kernel

Pradipta De
pradipta.de@sunykorea.ac.kr
• 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 or on multiple CPUs → 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?
  - Even kernel threads can be scheduled out
    - Since Linux adopted the pre-emptible kernel model
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
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)
Spinlock

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

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
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

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
}
```

```c
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
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]
Deadlocks

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>acquire lock A</td>
<td>acquire lock B</td>
</tr>
<tr>
<td>try to acquire lock B</td>
<td>try to acquire lock A</td>
</tr>
<tr>
<td>wait for lock B</td>
<td>wait for lock A</td>
</tr>
</tbody>
</table>

Solution: Order locking of resources
No automatic technique → programmer agreed upon convention
Putting It Together

• Managing concurrent access to shared resource on
  – Single CPU
  – Multiple CPUs