Lecture Notes - Concurrency and Synchronization

Debadatta Mishra
Indian Institute of Technology Kanpur

Synchronization and OS

- Two main aspects of OS synchronization design are---(i) provide synchronization constructs for the user space processes and threads, (ii) meet the synchronization requirements of the OS software itself. Even though synchronization is required to support concurrent access to any resource, we discuss synchronization of concurrent access to memory locations as the nature of the problem is same at a high level. Moreover, as most resources in computer systems are addressed through memory e.g., memory mapped I/O registers etc., shared memory is a reasonable abstraction for all shared resources.
- Multiple processes can share memory through shared memory APIs (e.g., shmget( ) etc.) by creating shared mapping through their virtual addresses. Concurrent access to variables allocated in the shared memory result in non-deterministic program behavior if the access is not synchronized. Similarly, threads of a single process share a common address space. Therefore, global/static variable or dynamically allocated memory from heap is shared across the threads. In both the scenarios, synchronization is required. The code segments/blocks accessing a shared memory area with at least one access being WRITE, the code blocks involved are called *critical sections*. For example, consider the following pseudocode,

```
static int count = 0;
void *thfunc( )
{
   int ctr=0;
   for(ctr=0; ctr<10000; ctr++)
         count++;
}
```

  When the above code is executed by two threads in a concurrent manner, the output is unpredictable as it depends on the order of scheduling, #of CPUs and the compiler generating assembly equivalent of the above code. For example, if *count*++ is compiled to {*MOV (count), R; ADD 1, R; and MOV R, (count)*}, the value of count can be maximum 20000 and minimum 2, irrespective of number of CPUs in the system. This is due to uncontrollable scheduling of two threads. The behavior will change in a uniprocessor system if the *count*++ is compiled to {*inc (count)*}. In short, the nondeterministic behavior makes the output incorrect and unpredictable.
- Many operating system data structures are accessed concurrently through system call handlers, exception handlers (on behalf of user processes like page faults) and interrupt handlers. For example, in ext4 file system, if two different processes create two different files at the same time, the inode bitmap structure is required to be updated in a concurrent manner. In a uniprocessor system, disabling preemption during system call handling can address concurrency issues by carefully designing

some unavoidable scheduling scenarios inside system call handler e.g., sleep, blocking read() etc. Another solution can be to disable preemption during access to the shared bitmap structure. However, in multiprocessor systems, explicit synchronization using mutual exclusion techniques like locks becomes unavoidable. Finally, concurrent access to data structures is required to handle interrupts and system calls, or to handle interrupts from multiple I/O devices. For example, network packet receive interrupt handler and recv( ) system call handler may access the same packet queue in a concurrent manner. To synchronize concurrent access from interrupt handlers and system call handlers in a uniprocessor system, disabling interrupts is required before executing the critical section. In multiprocessor systems, a combination of locks and interrupt disabling is required.

- When many concurrent execution threads try to enter into a critical section, only one of the threads must be allowed to enter the critical section and no other thread should be allowed an entry until the current thread leaves the critical section. This requirement is commonly known as *mutual exclusion*. On the other hand, another trivial requirement related to critical section is to allow a thread enter the CS if no other thread is executing the critical section. The last requirement is related to fairness aspect of the solution to the mutual exclusion problem where each contending thread should wait for equal amount of time on average. One metric to ensure fairness is to guarantee bounded number of attempts for any thread trying to enter the critical section.
- While designing solutions for mutual exclusion problem, one should bear in mind the tradeoffs of using different types of locks. Possible tradeoffs are---lock acquisition overhead vs. lock acquisition latency, lock granularity vs. code complexity and, CPU cycle wastage due to lock wait vs. context switch overhead. Most operating systems (including Linux) use architecture support as a building block to build synchronization constructs. However, in theory it is possible to implement locks without assuming any architecture support as we will see later.
- It may not be always required to use locks to protect critical sections. Several scenarios of critical section in the OS code along with the solutions are given below,

Contexts executing critical sections	Uniprocessor systems	Multiprocessor systems
System calls	Disable preemption	Locking
System calls, Interrupt handler	Disable interrupts	Locking + Interrupt disabling (local CPU)
Multiple interrupt handlers	Disable interrupts	Locking + Interrupt disabling (local CPU)

In multiprocessor systems, interrupt on local CPU is required to be disabled, failing which can result in deadlock. Moreover, the interrupt disabling and lock acquisition should be correctly ordered---first the local interrupt must be disabled before lock acquisition for correct behavior.

Synchronization techniques

- A close inspection of the following code attempting a software implementation of lock shows that it is incorrect.

  ```
  int lock=0;
  do_lock( )
  {
     while(lock);
     lock = 1;
  }
  do_unlock( )
  {
     lock = 0;
  }
  ```

  The comparison operation (in the while loop) can be true for multiple threads at the same time, both in uniprocessor and multiprocessor systems. In uniprocessor system, consider that two threads T1 and T2 are contending for the lock and T1 is currently scheduled when the lock becomes free. Further consider that T1 comes out of the while loop and gets descheduled by the OS and T2 gets scheduled. The while condition for T2 also becomes false and T2 acquires the lock before getting descheduled. T1 resumes its execution and also acquires the lock. This violates the mutual exclusion property. The issue in this implementation is with non-atomic test and set of the lock variable.
- One of the basic instruction used to implement a spinlock is atomic exchange (XCHG instruction in X86 systems). XCHG instruction swaps the content of a register and memory location atomically. A spinlock implemented using XCHG is explained in the slides and example code for your reference
- Coming back to our wrong spinlock implementation, *if the while loop condition checking and setting the lock variable if while condition is false* can be atomically executed, the lock implementation can address the mutual exclusion issue. One of the basic hardware instruction used to implement locks is CMPXCHG or CMPANDSWAP. A CMPXCHG instruction (X86) compares the content of a register with the memory content and swap them if they are equal. Refer to the slides and class example for implementation of spinlock using CMPXCHG.
- One major issue with spinlock implementations described so far is---they do not provide fairness guarantees as the implementations offer no bounds on the waiting time. To ensure fairness, some form of queueing/ordering is needed which requires additional variables along with the lock variable. One possible implementation makes use of another instruction fetch-and-add which fetches the content of the destination (generally a memory location) into the register before updating the memory location by adding the value in the register. One well known locking scheme with fairness (under certain assumptions) is ticket spinlocks. Ticket spinlocks makes use of two

shared variables i.e., *ticket* and *turn* where ticket is the value that is granted to a thread when it arrives and waits till its turn to enter the CS. Every thread performs a fetch-and-inc of the ticket variable using a local variable to copy old the value of ticket to a local variable (local turn) and waits till the global turn becomes equal to the value of local turn. The FIFO kind of movement of the ticket takes place when the global *turn* is incremented. Refer to the slides for the detailed implementation.

- In many real life scenarios, shared memory locations are accessed by many execution entities for the purpose of reading with at least one performing write operations. Assuming there are at least two different critical sections for read and write, mutual exclusion across the two critical sections (CS) using spinlocks is not a very good design. Specifically, in the cases when the number of concurrent read operations is significantly more than the number of concurrent write operations. In this scenario, at any given point of time, multiple reader threads can be allowed to access the CS. Read-write locks provide the flexibility of allowing multiple readers to execute the CS while providing exclusive access to the writer threads.
- A simple read-write lock can be implemented using a count variable and two spinlocks---(i) one of them is an exclusive lock acquired once by the readers irrespective of the number of readers or by every writer thread, (ii) the other lock (called the read lock) is used to protect the read count and ensure that the exclusive lock is locked by the first reader and released by the last reader. Please refer to the slides for implementation of a simple read-write lock. There are two disadvantages of using the above described read-write lock. First, two spinlocks are required; hence more operations are required. Second, the readers are temporarily serialized to acquire the read lock.
- One technique to implement a read-write lock using a single variable and supporting complete concurrency of readers can be to split the variable into two parts so that the reader-count can be maintained. For example in a 32-bit value the following construct can be used.
    - 0x01000000 --> unlocked
    - 0x00000000 --> locked for writing
    - 0x00FFFFFF --> one reader
    - 0x00FFFFFE --> two readers
    - .....

  When a reader wants to acquire lock, it decrements lock value. If the lock value is negative, it implies that the lock is acquired for writing or number of readers is more than 0xFFFFFF. Otherwise, the read-lock is granted. A write lock is granted only when the lock value is 0x01000000. Implementation of the read-write lock is left as an exercise.