Lecture Notes - Semaphores and classical problems

Debadatta Mishra
Indian Institute of Technology Kanpur

- Locking techniques allow exactly one thread to access the critical section. Consider a scenario when at most K threads concurrently accessing the shared memory structures is alright. For example, a finite array of size N can be accessed from a set of producer and consumer threads. More specifically, at most N concurrent producers are allowed if array is empty and at most N concurrent consumers are allowed if array is full. If mutual exclusion techniques are used, every access to the array must take a lock which implies that one thread is allowed at any point of time which is not very efficient. Using read-write locks does not help either as both producer and consumer accesses require modification of the array and associated counters.
- An example structure of a semaphore is shown below,

```
typedef struct semaphore{
                      int value;
                      Spinlock *L;
                      Queue *waitQ;
                      . . .
                    }
       int wait (sem_t *s) {
          s->value--;
         Wait if s->value <= 0
    }
    int post (sem_t *s) {
       s->value++;
      Wake up one if one or more are waiting
    }
```

  If the semaphore is initialized with 1, it is called a binary semaphore and can be used to implement mutual exclusion.
- Implementation of the wait and post functionalities require mutual exclusion (locks) as they involve increment/decrement along with condition checking which should be performed in atomic manner. A busy waiting implementation of a semaphore is shown in slide #6. However, to implement semaphores with sleep, the locks should be carefully placed as the wait and post requires sleep and wake-up. For example, consider the example implementation in slide #8 (buggy #3), the semaphore implementation is incorrect as it may lead to a deadlock because the wait implementation takes a lock before going to sleep. This problem can be addressed by releasing the lock before descheduling

itself (see slide #10). The assumption here is that the preemption is disabled while holding the lock, otherwise if an preemption occurs before the process state is set to be waiting after adding the process to wait queue, this can result in inconsistent behavior during scheduling.

- Semaphores can be used in an unconventional manner by initializing the semaphore to zero. For example, wait-for-child process can be implemented by initializing a semaphore to zero and the parent waiting on the semaphore. The child process invokes post when it is finished. Refer to the slide #11. Another example usage of a semaphore can be to appropriately order of execution. Consider the following example,

  ```
  A=0; B=0;
  Th0 {
          A = 1;
          printf("%d\n", B);
  }
  Th1 {
          B=1;
          printf("%d\n", A);
   }
  ```

  The possible outputs in this case can be {(01), (1,0), (1,1)}. To ensure an output (1,1), two binary semaphores can be used as shown in slide #14.
- One of the classical problems of concurrency is producer consumer problem as explained before. The solution presented in slide #16 using two semaphores (empty and used) does not offer a correct solution as the increment of producer and consumer counters (pctr and cctr, respectively) and access to the shared array in a concurrent manner can result in inconsistent states. Solution to the consistency problems can be addressed using a mutex as shown in slide #17. In this solution, there may be a deadlock in the following scenario: a producer threads takes the mutex and finds that the array is full and waits for empty semaphore. A consumer thread wants to consume and make an empty space, but it can not as it can not take the mutex. A correct solution using a single mutex and two semaphores can be implemented as shown in slide #18. However, this solution makes the producers and consumers serialized which is not very efficient. One way to address this issues is to use two mutexes, one for the producer threads and another for the consumer threads.
- Consider a modified version of producer consumer problem where a producer produces an array of bytes as a single element (instead of single integer) and the consumer consumes the elements. Employing mutual exclusion during the complete duration renders the implementation inefficient. One approach could be to take out the element production and consumption out of the critical section (see slide #20).

- Simply removing the copy operation out of the main critical section, the solution breaks. For example, the solution shown in slide#21 becomes incorrect when a producer at position K is descheduled and other producers complete processing positions K+1 and K+2. This may enable some consumers to consume the element at K which is still being produced. An approach to address this problem is to maintain a per-element mutex to allow more concurrency while maintaining correct mutual exclusion as shown in slide #22.
- Another synchronization construct is condition variables which is used to communicate between two threads. pthread_cond_wait(condition, mutex) atomically releases the mutex and waits on a condition variable without performing *any condition check.* When the condition is signaled by another thread using pthread_cond_signal(cond), the waiting thread resumes its execution holding the lock. Note that, pthread_cond_wait must be invoked holding the lock and the condition should be checked as spurious wakeup can occur. Also, call to pthread_cond_signal(cond) must hold the lock otherwise a waiter will wait indefinitely. Consider a case when pthread_cond_wait( ) has not added the thread to a wait queue and a wakeup call occurs around the same time after making the condition to be true. The waiting thread will never be woken up. An example implementation of condition variables is shown in slide #25.
- There are potentially two primary issues with concurrency---(i) incorrect mutual exclusion, (ii) deadlocks. Deadlocks can occur when threads are stuck even if the no thread is executing in critical section. For example, let us consider a typical transfer transaction shown in slide #26. The code can lead to deadlock if two accounts (A, B) are involved in two transfer transactions executing concurrently where for one transaction A is the source and B is the destination account and for the other transaction it is the reverse. One common technique to avoid these situations is to implement some form of ordering of locks. In this example, always taking a lock on an account with lower account number avoids the problem of deadlocks.
- Another classical problem is dining philosophers where a philosopher requires two forks before she can eat and goes back to thinking after finishing. If every philosopher picks up the left fork (or the right fork) first, there will be a deadlock. One way to break the tie is to assign unique numbers to forks and every philosopher picks up the lowest numbered fork first. That means the last philosopher picks up the right fork first. In general a deadlock occurs when all of the following conditions occur,
    - Exclusive access to the resource (CS), more than one threads are not allowed to access (This is almost unavoidable)
    - Hold-and-wait: There is a condition when threads hold one lock and wait for other lock to enter the CS, mostly a case when multiple locks

are used.

  - No resource preemption: Locks can not be forcibly removed from threads holding them
  - Circular wait: A cycle of threads requesting locks held by others. Specifically, a cycle in the directed graph G(V, E) where V is the set of processes, (v1, v2) ∈ E if v1 is waiting for a lock held by v2