hw5

课后学习面向对象编程概念（类、对象、继承、多态）

b站学过

Practice Exercises

5.4 Explain why spinlocks are not appropriate for single-processor systems yet are often used in multiprocessor systems.

On a single-processor system, the thread holding a lock cannot be running while another thread is testing the lock, because only one thread/process can be running at a time. Therefore the thread will continue to spin and waste CPU cycles until its time slice end. That is why threads on a single-processer system should always sleep rather than spin, if they encounter a lock that is held.

On a multiprocessor system, multiple threads (the thread holding a lock, and the thread testing the lock) can actually be running at the same time. Therefore, it is appropriate for a thread waiting for lock to go into a spin wait, because the lock could be released while it is waiting.

5.5 Show that, if the `wait()` and `signal()` semaphore operations are not executed atomically, then mutual exclusion may be violated.

A wait operation atomically decrements the value associated with a semaphore. If two wait operations are executed on a semaphore when its value is 1, and if the two operations are not performed atomically, then it is possible that both operations might proceed to decrement the semaphore value, thereby violating mutual exclusion.

Exercises

5.10 Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single-processor system if the synchronization primitives are to be used in user-level programs.

If a user-level program is given the ability to disable interrupts, then it can disable the timer interrupt and prevent context switching from taking place, thereby allowing it to use the processor without letting other processes to execute.

5.11 Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor system.

Depends on how interrupts are implemented, but regardless of how, it is a poor choice of techniques.

Case 1 – interrupts are disabled for ONE processor only – result is that threads running on other processors could ignore the synchronization primitive and access the shared data.

Case 2 – interrupts are disabled for ALL processors – this means task dispatching, handling I/O completion, etc. is also disabled on ALL processors, so threads running on all CPUs can grind to a halt.

5.15 Consider how to implement a mutex lock using an atomic hardware instruction. Assume that the following structure defining the mutex lock is available:

```
typedef struct { int available; } lock;

void init(lock *mutex) {
    // available=0 -> lock is available, available=1 -> lock is unavailable
    mutex->available = 0;
}

void acquire(lock *mutex) {
    while (test_and_set(&(mutex->available))) // TEST the available
        ; // spin-wait (do nothing)
}

void release(lock *mutex) {
    mutex->available = 0;
}
```

5.17 Assume that a system has multiple processing cores. For each of the following scenarios, describe which is a better locking mechanism — a spinlock or a mutex lock where waiting processes sleep while waiting for the lock to become available:

(1) If the lock is to be held for a short duration, it makes more sense to use a spinlock as it may in fact be faster than using a mutex lock which requires suspending - and awakening - the waiting process.

(2) If it is to be held for a long duration, a mutex lock is preferable as this allows the processing core to schedule another process while the locked process waits.

(3) If the thread may be put to sleep while holding the lock, a mutex lock is definitely preferable as you wouldn’t want the waiting process to be spinning while waiting for the other process to wake up.

5.18 Assume that a context switch takes T time. Suggest an upper bound (in terms of T) for holding a spinlock. If the spinlock is held for any longer, a mutex lock (where waiting threads are put to sleep) is a better alternative.

The maximum time that can be tolerable in terms of time t will be <2xT Thus exceeding the maximum time, a mutex lock will be used instead of spin lock where the threads that are in wait mode are switched to sleep mode

5.20 Consider the code example for allocating and releasing processes shown in Figure 5.23.

a. Identify the race condition(s).

There is a race condition on the variable number of processes.

b. Assume you have a mutex lock named mutex with the operations acquire() and release(). Indicate where the locking needs to be placed to prevent the race condition(s).

A call to acquire() must be placed upon entering each function and a call to release() immediately before exiting each function.

c.Could we replace the integer variableint number_of_processes = 0 with the atomic integer atomic_t number_of_processes = 0 to prevent the race condition(s)

No, it would not help. The reason is because the race occurs in the allocate process() function where number of processes is first tested in the if statement, yet is updated afterwards, based upon the value of the test. It is possible that number of processes = 254 at the time of the test, yet because of the race condition, is set to 255 by another thread before it is incremented yet again.

5.23 Show how to implement the wait() and signal() semaphore operations in multiprocessor environments using the test_and_set() instruction. The solution should exhibit minimal busy waiting.

```
int guard = 0;
int semaphore value = 0;
wait() {
 while (TestAndSet(&guard) == 1)
     ;
 if (semaphore value == 0){
     atomically add process to a queue of processes
     waiting for the semaphore and set guard to 0;
 }
 else
 {
     semaphore value--;
     guard = 0;
 }
}
signal() {
 while (TestAndSet(&guard) == 1)
     ;
 if (semaphore value == 0 && there is a process on the wait queue)
     wake up the first process in the queue of waiting processes
 else
     semaphore value++;
 guard = 0;
}
```

5.28 Discuss the tradeoff between fairness and throughput of operations in the readers–writers problem. Propose a method for solving the readers–writers problem without causing starvation.

Throughput in the readers-writers problem is increased by favoring multiple readers as opposed to allowing a single writer to exclusively access the shared values. On the other hand, favoring readers could result in starvation for writers.When a reader arrives and notices that another reader is accessing the database, then it would enter the critical section only if there are no waiting writers.The starvation in the readers/writers problem could be avoided by keeping timestamps associated with waiting processes. When a writer is finished with its task, it would wake up the process that has been waiting for the longest duration.