操作系统原理	October 24, 2011
Homework 5	
计算机 92 班	戴唯思 09055029

6.4

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

Spinlocks are locks where the thread simply *waits* in a loop (spin) repeatedly checking until the lock becomes available.

- Spinlocks *should not be used* on single-processor systems.

  In the best case, a spin lock on a single processor system will waste resources, slowing down the owner of the lock; in the worst case, it will deadlock the processor.

- Spinlocks *are not needed* in single-processor systems.

  Generally you should start with a mutex, and if profiling show it to be a bottleneck, you may want to consider a spinlock.

- Spinlocks are efficient when implemented on multi-processor systems.

6.9

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

Given semaphore $x = 0$, when a `wait()` and a `signal()` operation are done atomically, x changes to 1 then 0, or -1 then 0. If they are not executed atomically, x gets an uncertain value except at the initial step.

6.11

> **The Sleeping-Barber Problem**. A barbershop consists of a waiting room with n chairs and a barber room with one barber chair. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy but chairs are available, then the customer sits in one of the free chairs. If the barber is asleep, the customer wakes up the barber. Write a program to coordinate the barber and the customers.

```
class customer:
    def main(self):
        if barbershop.all_chairs_occupied:
            self.leave()
        self.sit_in(pick_one_from(barbershop.free_chairs))
```

```
class barber:
    def main(self):
        while True:
            while barbershop.chairs_occupied>0:
                self.serve(pick_one_from(barbershop.occupied_chairs))
            self.sleep(until=lambda:(barbershop.chairs_occupied==0))
```

Supplement 1

`P`, `V` described like this would cause starvation, because `V` resumes a process from the tail of the waiting queue, ones on the head would never be resumed.

Initialize `s` to 1. Invoke `P(S)` and `V(S)` separately before and after visits to the shared variable.

Supplement 2

```
def managed_write:
    Swait(wcount,1,1);
    Swait(Rcount, R, 0;
          mutex,1,1);
    write();
    Ssignal(mutex,1);
    Ssignal(wmutex,1);

def managed_read:
    Swait(wcount,w,0;
          Rcount,1,1);
    read();
    Ssignal(Rcount,1);
```

Supplement 3

4 processes: enter the room, students do the exam, students leave the room, the teacher leaves the room

Semaphores:

- gate=1
- exampaper=number of students
- teacher=1
- freeseat=number of students
- pack=0

Enter the room	Students do the exam	Leave(S)	Leave(T)
P(gate)	P(teacher)	P(own_exampaper)	P(pack)
Enters the room	Get the exam paper	V(own_exampaper)	V(pack)
V(gate)	P(exampaper)	P(gate)	P(gate)
V(freeseat)	V(teacher)	Leave the room	Leave the room
	Write on the exam paper	V(gate)	V(gate)
	P(teacher)		
	Hand in the exam paper		
	V(exampaper)		
	V(teacher)		

7.3

A possible solution for preventing deadlocks is to have a single, higher-order resource that must be requested before any other resource. For example, if multiple threads attempt to access the synchronization objects $A \ldots E$, deadlock is possible. We can prevent the deadlock by adding a sixth object F. Thenever a thread wants to require the lock for any object $A \ldots E$, it must first acquire the lock for object F. This solution is known as containment: the locks for objects $A \ldots E$ are contained within the lock for object F. Compare this scheme with the circular-wait scheme of section 7.4.4.

Table 1: Comparison of containment and circular-wait scheme

Containment	Circular-Wait
Both prevent the deadlock	
Inefficient	Efficient
One-at-a-time	Not restricted
Usually easy to implement	Usually hard to implement

7.11

Consider the following snapshot of a system:

	Allocation	*Max*	*Available*
	A B C D	*A B C D*	*A B C D*
P_0	0 0 1 2	0 0 1 2	1 5 2 0
P_1	1 0 0 0	1 7 5 0	
P_2	1 3 5 4	2 3 5 6	
P_3	0 6 3 2	0 6 5 2	
P_4	0 0 1 4	0 6 5 6	

Answer the following questions using the banker's algorithm:

a. What is the context of the matrix *Need*?

b. Is the system in a safe state?

c. If a request from process P_1 arrives for $(0, 4, 2, 0)$, can the request be greanted immediately?

a.

Need:

	A	*B*	*C*	*D*
P_0	0	0	0	0
P_1	0	7	5	0
P_2	1	0	0	2
P_3	0	0	2	0
P_4	0	6	4	2

b. Yes, the plan P_0, P_2, P_1, P_3, P_4 satisfies the safety requirements.

c. Yes, one possible sequence is P_0, P_2, P_3, P_1, P_4.

References

1. `https://secure.wikimedia.org/wikipedia/en/wiki/Spinlock`
2. `http://stackoverflow.com/questions/1025859/is-spin-lock-useful-in-a-single-processor-uni-core-architecture`
3. `http://comp.ist.utl.pt/ec-sc/0405/docs/ecos-2.0b1/doc/html/ref/kernel-spinlocks.html`
4. Thomas Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. IEEE Transactions on Parallel and Distributed Systems, vol. 1, num. 1, January 1990, pages 6 - 16. An earlier version appeared in Proc. 1989 International Conference on Parallel Processing (ICPP), August 1989. `http://www.cs.washington.edu/homes/tom/pubs/spinlock.pdf`
5. `http://www.moserware.com/2008/09/how-do-locks-lock.html`

6. http://wiki.osdev.org/Atomic_operation

7. http://www.cs.rpi.edu/~moorthy/Courses/os00/soln78.html

8. http://www.cs.umbc.edu/courses/undergraduate/421/spring06/homework1-soln.pdf