【OS】Day32(2)

Class	Operating System: Three Easy Pieces
Date	@February 5, 2022

【Ch28】Locks

28.6 A Failed Attempt: Just Using Loads/Stores

```
typedef struct __lock_t { int flag; } lock_t;

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

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;          // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

Figure 28.1: **First Attempt: A Simple Flag**

In this first attempt, the idea is quite simple: use a simple variable(flag) to indicate whether some thread has possession of a lock.

The first thread that enters the critical section will call `lock()`, which tests whether the flag is equal to 1(in this case, it is not), and then sets the flag to 1 to indicate that the thread now holds the lock. When finished, the thread calls `unlock()` and clears the flag, thus indicating that the lock is no longer held.

If another thread happens to call `lock()` while that first thread is in the critical section, it will simply spin-wait in the while loop for that thread to call `unlock()` and clear the flag.

Once that first thread does so, the waiting thread will fall out of the while loop, set the flag to 1 for itself, and proceed into the critical section.

Unfortunately, the code has two problems: one of correctness, and another of performance.

The correctness problem is simple to see once we get used to thinking about concurrent programming. Imagine the code interleaving in the figure above; assume `flag = 0` to begin.

We can easily produce a case where both threads set the flag to 1 and both threads are thus able to enter the critical section. This behaviour is what professionals call "bad".

The performance problem, which we will address more later on, is the fact that the way a thread waits to acquire a lock that is already held: it endlessly checks the value of flag, a technique known as spin-waiting.

Spin-waiting wastes time waiting for another thread to release a lock. The waste is exceptionally high on a uniprocessor, where the thread that the waiter is waiting for cannot even run.

Thread 1	**Thread 2**
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

Figure 28.2: **Trace: No Mutual Exclusion**