【OS】Day32

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

【Ch28】Locks

28.1 Locks: The Basic Idea

As an example, assume our critical section looks like this:

```
balance = balance + 1;
```

To use a lock, we add some code around the critical section like this:

```
lock_t mutex; //some globally-allocated lock 'mutex'

lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

A lock is just a variable, and thus to use one, we must declare a lock variable of some kind(such as mutex above).

This lock variable holds the state of the lock at any instant in time. It is either available(or unlocked or free) and thus no thread holds the lock, or acquired(or locked or held), and thus exactly one thread holds the lock and presumably is in a critical section.

We could store other information in the data type as well, such as which thread holds the lock, or a queue for ordering lock acquisition, but information like that is hidden from the user of the lock.

The semantics of the `lock()` and `unlock()` routines are simple. Calling the routine `lock()` tries to acquire the lock; if no other thread holds the lock(i.e. it is free), the thread will acquire the lock and enter the critical section; this thread is sometimes said to be the owner of the lock.

If another thread then calls `lock()` on that same lock variable(`mutex` in this example), it will not return while the lock is held by another thread; in this way, other threads are prevented from entering the critical section while the first thread that holds the lock is in there.

Once the owner of the lock calls `unlock()`, the lock is now available again.

If no other threads are waiting for the lock, the state of the lock is simply changed to free.

If there are waiting threads, one of them will notice this change of lock's state, acquire the lock, and enter the critical section.

28.2 Pthread Locks

The name that the POSIX library uses for a lock is a mutex, as it is used to provide mutual exclusion between threads, i.e., if one thread is in the critical section, it excludes the others from entering until it has completed the section.

When we see the following POSIX threads code, wee should understand that it is doing the same thing as above:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Pthread_mutext_lock(&lock); //wrapper; exits on failure
balance = balance + 1;
Pthread_mutex_unlock(&lock);
```

Here, the POSIX version passes a variable to lock and unlock, as we may be using different locks to protect different variables.
Doing so can increase concurrency: instead of one big lock that is used any time any critical section is accessed(a coarse-grained locking strategy), one will often protect

different data and data structures with different locks, thus allowing more threads to be in locked code at once(a more fine-grained approach).

28.4 Evaluating A Lock

To evaluate whether a lock works(and works well), we should establish some basic criteria.

- The first is whether the lock does its basic task, which is to provide mutual exclusion.
- The second is fairness. Does each thread contending for the lock get a fair shot at acquiring it once it is free? Another way to look at this is by examining the more extreme case: *does any thread contending for the lock starve while doing so, thus never obtaining it?*
- The final criterion is performance, specifically the time overheads added by using the lock.

28.5 Controlling Interrupts

One of the earlies solutions used to provide mutual exclusion was to disable interrupts for critical sections; this solution was invented for single-processor systems. The code would look like this:

```
void lock() {
  DisableInterrupts();
}

void unlock() {
  EnableInterrupts();
}
```

By turning off interrupts before entering a critical section, we ensure that the code inside the critical section will not be interrupted, and thus will execute as if it were atomic.

When we are finished, we re-enable interrupts and thus the program proceeds as usual.

The main positive of this approach is simplicity. The negatives, unfortunately, are many.

First, this approach requires us to allow any calling thread to perform a privileged operation(turning interrupts on and off), and thus trust that this facility is not abused. Any time we are required to trust an arbitrary program, we are probably in trouble.

Here, the trouble manifests in numerous ways:

1. A greedy program could call `lock()` at the beginning of its execution and thus monopolize the processor; worse, an errant or malicious program could call `lock()` and go into an endless loop.

   Using interrupt disabling as a general synchronization solution requires too much trust in applications.

2. The approach does not work on multiprocessors. If multiple threads are running on different CPUs, and each try to enter the same critical section, it does not matter whether interrupts are disables; threads will be able to run on other processors, and thus could enter the critical section.

3. Turning off interrupts for extended periods of time can lead to interrupts becoming lost, which can lead to serious system problems.

4. This approach can be inefficient. Code that masks or unmasks interrupts tends to be executed slowly by modern CPUs.