Chapter 28 Locks

Introduce lock synchronization primitive to protect critical sections.

```
mu = sync.Mutex{}
```

```
mu.Lock()
counter = counter + 1 // code in the critical section
mu.Unlock()
```

Declare a lock variable: mutex (mutual exclusion)

- Holds the state of the lock at any instant in time
  - Available — unlocked and free
    - No thread holds the lock
  - Acquired — locked or held
    - Exactly one thread holds the lock
  - Could also hold other info in the lock/mutex data structure
    - Which thread holds the lock
    - A queue for ordering lock acquisitions

Semantics of Lock() and Unlock():

- Lock():
  - The first thread will acquire the lock (and can enter the CS)
  - Becomes owner of lock
  - Another thread calling Lock() on the same mutex
    - Will not return (block): thread is waiting
    - While other thread is in the CS
- Unlock():
  - Owner of mutex calls Unlock() (when leaving the CS)
    - Lock becomes free
  - If no other threads are waiting for the lock
    - State of lock is set to free
  - Otherwise: there are threads waiting stuck in the Lock() method
    - One of them will acquire the lock (and can enter the CS)

Lock Granularity

- Coarse-grained locking
  - One “big” lock
    - Covering large/entire data structure
- Fine-grained locking
  - Several locks
    - Each lock covering different positions of a data structure
  - Benefit: increased concurrency compared to a single “big” lock for all CSs

Thread API Guidelines (from Chapter 27)

- Keep it simple
  - Any code to lock or signal between threads should be *as simple as possible*
  - Tricky thread interactions lead to subtle bugs (deadlocks)
  - Applies to Go's channel-based interactions as well
- Minimize thread interactions
  - Keep the number of ways in which threads interact to a minimum
  - Each interaction should be carefully though out and constructed with well-known patterns
- Initialize locks and condition variables (in C)
  - Failure to initialize: sometimes works fine and sometimes fails in strange ways

Thread API Guidelines (continued)

- Check return codes:
  - Return codes in C and Unix contain info: should be checked
- Be careful about how you pass arguments to and return values from threads
  - If pass by reference to a variable allocated on the stack: you are doing it wrong!
- Each thread has its own stack
  - A thread’s locally allocated variables should be considered private
  - To share data between threads
    - Allocate space on the heap
    - Or use a global variable
- Always use condition variables (CVs) to signal between threads
  - Don’t use a simple flag!
  - Good alternative to CVs: Channels in Go.

28.4 Evaluating Locks

Evaluation criteria

- **Basic task/Correctness**: does it provide mutual exclusion?
- **Fairness**: does each thread contenting for the lock get a fair shot at acquiring the lock once it is free?
- **Performance**: time overheads added by using a lock. Cases to consider:
  - No contention
  - Multiple threads contending for the lock on
    - A single CPU
    - Multiple CPUs

28.5 Controlling Interrupts

Early solutions for single CPU systems:

```
func Lock() {
        DisableInterrupts()
}
```

```
func Unlock() {
        EnableInterrupts()
}
```

HW instruction to turn off interrupts
And re-enable interrupts again (unlock)

Pro/Con:
+ Easy to understand that no other thread or the OS can interrupt during the CS

- Requires calling thread to perform *privileged* instruction
  - Must trust thread / arbitrary program
  - Greedy program: call Lock() at beginning of execution and monopolize the CPU
  - Malicious program: call Lock() and enter infinite loop…
  - Only recourse: restart system
- Does not work on multiprocessor systems
  - Each CPU has their own interrupts
- Disabling interrupts for long time: lead to lost interrupts
  - CPU may miss I/O completion event
- Slow to disable / enable interrupts

28.6 Lock Implementation Using Loads/Stores

Let's use a single **flag** variable

- Access flag variable via normal memory load/store
- Will it be sufficient?

If a thread calls lock() when it is being held by another thread,
it will spin-wait until the thread calls unlock()

```
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**

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**

Two problems:

- Correctness
- Performance

Fig. 28.2 Shows an interleaving that both threads get the lock — and can enter the CS.

This means that we have no ***mutual exclusion****, violating the correctness.*

Performance: **spin-waiting** wastes time and CPU cycles waiting for another thread to release the lock

- On uniprocessor: the thread that the waiter is waiting for can't even run until a timer interrupt.

28.7 Spin Locks with Test-and-Set

HW support

- test-and-set instruction
- Intel x86: **xchg** instruction

The TestAndSet() func is performed atomically.

- Copies and returns the old value
- Update ptr with a provided new value

```
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new;     // store 'new' into old_ptr
    return old;         // return the old value
}
```

This TAS instruction makes it possible to implement a simple **spin lock**.

Allows to test the old value (returned) while setting the memory to the new value.

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

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

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

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

Figure 28.3: **A Simple Spin Lock Using Test-and-set**

First case:

- A thread calls lock() when no other threads hold the lock
    - Flag is 0
  - When thread calls TAS(flag, 1)
    - TAS returns old value of flag, which is 0
  - Thread will not spin since TAS(flag, 1) = 0 != 1
  - TAS will set flag to 1 (atomically)
    - Indicating that lock is now held
  - On exiting CS and call unlock()
    - The thread sets flag back to 0.

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

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

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

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

Figure 28.3: **A Simple Spin Lock Using Test-and-set**

Second case:

- A thread calls lock() when another thread hold the lock
  - Flag is 1
- When this thread calls TAS(flag, 1)
  - It returns the old value of flag which is 1
  - Hence the thread enters spin-waiting
    - Repeatedly checking the flag using the TAS instruction
  - Only when another thread sets the flag to 0
    - Will “this thread” call TAS() again, and hopefully returning 0.
    - While atomically setting flag to 1, and acquiring the lock and can enter the CS

Simplest type of lock: Spin Lock

Simply spins, wasting CPU cycles until lock becomes available.

To work on a single CPU, it requires a **preemptive scheduler**

- Interrupt threads periodically to run OS scheduler and replace threads…

28.8 Evaluating Spin Locks

Correctness: YES!

Fairness:

- Simple spin locks are not fair and may lead to starvation
- A thread may spin forever under contention
  - Other threads may grab the lock in front of a spinning thread

Performance:

- Single CPU case: Overheads can be quite “painful”
  - Thread holding lock preempted in CS
  - Scheduler runs all other threads
    - Each tries to acquire lock
    - Each thread spin for the duration of a time slice (until giving up)
    - Wasted an entire time slot

Performance:

- Multiple CPUs case: Can work reasonably well
  - Thread A on CPU 1
  - Thread B on CPU 2
  - A and B contend for the lock
    - If A gets the lock
    - B tries to get the lock, B will spin (on CPU2)
    - Assuming the CS is short
    - Lock quickly becomes available
    - And gets acquired by B

Spinning to wait for lock held by another processor can be effective.

28.9 Compare-And-Swap

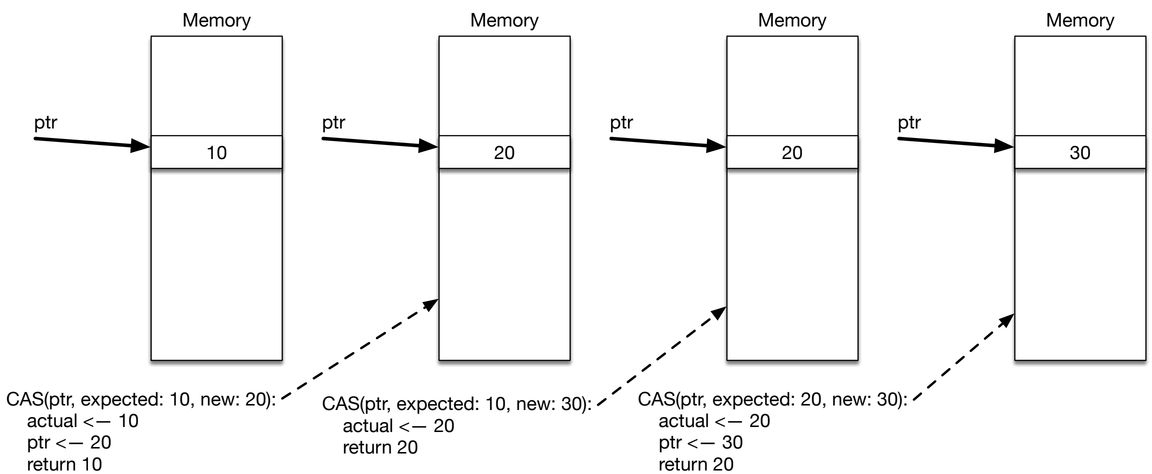
Slightly more powerful primitive than Test-And-Set.

Not needed for mutex locks.

For lock-free synchronization, we need CAS.

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int actual = *ptr;
    if (actual == expected)
        *ptr = new;
    return actual;
}
```

Figure 28.4: **Compare-and-swap**



28.13 A Simple Approach: Just Yield, Baby

First attempt:

- When you are going to spin
  - Instead: give up CPU to another thread
- Assume OS primitive: **yield()**

```
void init() {
    flag = 0;
}

void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}

void unlock() {
    flag = 0;
}
```

Recall: Thread state: **running, ready,** or **blocked.**

When a thread calls yield(), it deschedules itself.
Moving from **running** to **ready** state.

Example Two threads; A holds the lock; B want to get the lock.
Instead of B spinning, B simply yields the CPU, allowing A to run again and finish its CS.

Example. 100 threads contenting for a lock, repeatedly.

One thread acquires the lock — and is preempted before finishing its CS (and thus not releasing the lock)
The other 99 threads call Lock(), fint that it is held, and yield the CPU.
99 threads will run-and-yield before the thread holding the lock gets to run again.
Costly context switches (for no good reason).

28.14 Using Queues: Sleeping Instead of Spinning

Exert more control over which thread gets to acquire the lock next

- OS support
- Queue to keep track of threads waiting to acquire the lock

```
typedef struct __lock_t {
    int flag;
    int guard;
    queue_t *q;
} lock_t;

void lock_init(lock_t *m) {
    m->flag  = 0;
    m->guard = 0;
    queue_init(m->q);
}

void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}

void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (queue_empty(m->q))
        m->flag = 0; // let go of lock; no one wants it
    else
        unpark(queue_remove(m->q)); // hold lock (for next thread!)
    m->guard = 0;
}
```

Figure 28.9: **Lock With Queues, Test-and-set, Yield, And Wakeup**

Use two locks

- A **guard** lock to protect the lock_t data structure (queue and flag)
  - Lock() and Unlock() must acquire the **guard** lock
  - Before updating the queue and flag variables
- The lock itself is the flag variable
- The **guard** lock is a spinning lock
  - But the time spent spinning is limited
    to the few instructions in the lock() and unlock() code
- Much better than scenarios where user-defined and arbitrarily long CSs.

If flag = 0: we get the lock

Otherwise:

- Add ourselves to the queue
- Release the guard lock
- Park ourselves

Unlock():

- Acquire the guard lock
- If queue empty: release lock
- Else:
  - Pass lock onto the next thread waiting in queue: Unpark()
  - When the “unparked” thread is woken up by the scheduler, it will resume after the park(), line 23
- Release the guard lock