We don’t need mfence for single threaded applications because no other thread can modify the memory. Even if a load gets reordered with a store (not on the same address as load), the thread will see the same value, before and after the store.

Lock prefix forces an instruction to work directly on cache passing over the store buffer. The hardware locks all the cache lines that a lock instruction is going to access before executing the instruction. Locking a cache line invalidates the cache line on other cores and prevent them from accessing the cache line. A lock prefix guarantees the atomicity of single instruction.

If multiple CPUs try to write and read (at least one write) a cache line simultaneously, then the hardware spends most of the time serving invalidation requests of cache lines. This event is also called cache line bouncing.

```
label1:
lock add $1, lockvar
cmp $1, lockvar
je label2
lock sub $1, lockvar
jmp label1
label2:
```

For example, if multiple threads execute the above code on different cores then most of the time the cores will be stalled due to the invalidation of cache lines. This problem is severe if the number of cores is very high (> 32). In this case, a CPU has to send an invalidation request to all the cores that make the cache coherence logic very slow. Due to cache line bouncing it is possible to write a single thread allocation that works better than a multi-threaded application running on 80 cores.

```
void acquire (struct spinlock *lk) {
    pushcli();
    while (atomic_xchg(&lk->locked, 1) != 0);
    __sync_synchronize ();
}
```

A spin lock implements busy waiting during the acquire logic. A spin lock is useful when the critical section is very small. Let us look at one possible implementation of a spin lock. atomic_xchg atomically swaps the content of an input memory location with the input value and returns the old value of the memory location. It uses “lock xchg” instruction to do it atomically. The lock prefix ensures that no other thread can access the input memory location when xchg is executing, and hence no other thread will see the value of “lk->locked” as zero after the atomic swap is completed.

Because this lock implementation is doing a write operation while waiting during lock acquisition, it may cause a lot of cache invalidation requests among waiters. This can be rewritten to reduce the number of writes.

```
void acquire (struct spinlock *lk) {
    pushcli();
    while (atomic_xchg(&lk->locked, 1) != 0) {
      while (lk->locked == 1);
    }
    __sync_synchronize ();
}
```

If the atomic_xchg returns one, a thread need not execute a locked instruction until the value of "lk->locked" is one. After seeing a zero value of the lock variable waiting threads may try to acquire it atomically. This implementation will do fewer cache invalidation compared to earlier implementation.

__sync_synchronize is "mfence" that would prevent the reordering of critical section instructions.

pushcli saves the old interrupt flag and disable interrupts. This allows interrupt handlers to use this spinlock implementation. Otherwise, a critical section protected by a spinlock can get interrupted and may try to acquire the same lock. This may cause a deadlock.

One problem with the spinlock implementation is fairness. If multiple threads are waiting for a lock, it is possible that a thread which arrived later gets the lock before previous threads. A ticket spin lock solves this problem by returning a unique token to waiting threads in the order of their arrival. Lock release sets the fields of the lock such the thread which has a specific token can acquire the lock.

```
struct lock {
   volatile unsigned head;   // initially 0
   volatile unsigned tail;      // initially 0
};
acquire:
oldtail = atomic_xadd (&lockvar->tail, 1);
while (oldtail != lockvar->head);
release:
lockvar->head++;
```

Ticket spin lock

atomic_xadd atomically adds 1 to the lockvar->tail and return the old value of lockvar->tail. The tail field of lockvar contains the current token number. When a thread tries to acquire a lock it atomically reads the current token value and increment the value of token by 1. The head field contains the value of the token that is allowed to acquire the lock. Notice that no two threads can have the same token because the token is atomically read and updated by the incoming threads. When a thread releases the lock, it increments the head to the token of next waiting thread.

A readers-writer lock allows multiple readers (threads who do not update the protected shared data) in the critical section. However, only one writer (thread who updates the protected shared data) is allowed to enter the critical section, when no other reader is executing in the critical section.

```
volatile unsigned lockvar = 10000;
read_acquire:
while (atomic_sub(&lockvar, 1) < 0) {
    atomic_add (&lockvar, 1);
    while (lockvar <= 0);
}
read_release:
atomic_add (&lockvar, 1);

write_acquire:
while (atomic_sub (&lockvar, 10000) != 0) {
   atomic_add (&lockvar, 10000);
   while (lockvar != 10000);
}
write_release:
atomic_add (&lockvar, 10000);
```

Readers-writer lock

One possible implementation of a readers-writer lock is shown above. atomic_sub and atomic_add atomically add and subtract an input value to the input memory location and return the updated value.

Notice, that multiple readers can execute in parallel. The writer waits until some readers are active. Finally, it sets the lockvar to zero before entering in the critical section to prevent readers and writers from entering the critical section. In this implementation, a writer may wait infinitely if readers are keep coming. To prevent this the writer may set the value of lockvar in such a way that new readers are not allowed when a writer is waiting.