Test-And-Set

- a *test-and-set* (or *atomic exchange*) instruction fetches a value and replaces it with a new value in a single step

```
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
}
```

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

void init(lock_t *lock) {
    // 0: lock is available, 1: lock 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;
}
```

atomic test-and-set ensures no chance of another thread grabbing the lock between you seeing it is available and taking it

spin lock implemented with test-and-set

Spin Locks

- *spin lock* refers to the behavior of the thread waiting for the lock

```
while (TestAndSet(&lock->flag, 1) == 1)
    ; // spin-wait (do nothing)
```

  - ensures mutual exclusion
  - requires a preemptive scheduler
    - otherwise spinning thread(s) can tie up the CPU(s), preventing the lock holder from running (and ever relinquishing the lock)
  - not fair
    - no guarantees about which thread gets the lock if many are waiting, and any one thread can keep being unlucky
  - performance can be quite poor on a single CPU
    - waiting threads eat up CPU time spinning as the scheduler cycles between them
  - performance can be reasonable on multiple CPUs if each thread gets its own CPU
    - spinning threads don't impede the progress of the lock holder, and the time the lock is held is presumably short

Compare-and-Swap

- a *compare-and-swap* (or *compare-and-exchange*) instruction compares and conditionally replaces it with a new value in a single step

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

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

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

void lock(lock_t *lock) {
    while (CompareAndSwap(&lock->flag, 0, 1) == 1)
        ; // spin
}

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

spin lock implemented with compare-and-swap

(similar to test-and-set version)

Load-Linked and Store-Conditional

- *load-linked* fetches a value from memory into a register
- *store-conditional* only successfully updates the value in memory if no other store has occurred since the last load-linked

```
int LoadLinked(int *ptr) {
    return *ptr;
}

int StoreConditional(int *ptr, int value) {
    if (no update to *ptr since LL to this addr) {
        *ptr = value;
        return 1; // success!
    } else {
        return 0; // failed to update
    }
}
```

spin lock implemented with load-linked and store-conditional

```
void lock(lock_t *lock) {
    while (1) {
        while (LoadLinked(&lock->flag) == 1)
            ; // spin until it's zero
        if (StoreConditional(&lock->flag, 1) == 1)
            return; // if set-to-1 was success: done
                    // otherwise: try again
    }
}

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

detects if another thread changed the flag since we looked at it rather than ensuring no change occurred

Fetch-and-Add

- *fetch-and-add* fetches and increments a value in a single step

```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn   = 0;
}

void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
        ; // spin
 }

void unlock(lock_t *lock) {
    lock->turn = lock->turn + 1;
}
```

ticket lock implemented with fetch-and-add

atomic fetch-and-add means that each thread gets a unique ticket number

don't have to worry about being interrupted in the middle of unlock because the lock isn't relinquished `lock→turn` is set (the last step of the statement)

achieves fairness – it will eventually become each thread's turn (ticket numbers effectively establish a queue)

(Sort Of) Avoiding Spinning – Yield

- spinning threads can voluntarily give up the CPU

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

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

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

- performance
  - less CPU time spinning, more CPU time context switching (and running the scheduler)
- fairness
  - no guarantee that starvation is avoided

Avoiding Spinning – Sleeping

```
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;
}
```

- instead of spinning, sleep until the lock is available
  - waiting threads go into a queue (for fairness)
  - still utilizes spinning when waiting for the guard lock, but this should be of very limited duration

guard lock must be released before sleeping...why?

with the guard lock released, what can go wrong?

system calls (Solaris) –
`park` puts the calling thread to sleep
`unpark` wakes the specified thread

Solaris first released by Sun Microsystems in 1993 acquired by Oracle in 2010 now largely defunct

lock is transferred rather than released – why?

Avoiding Spinning – Sleeping

```
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());
                                setpark(); // new code
                                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;
}
```

`setpark` indicates the thread's desire to park – a subsequent `park()` will return immediately instead of sleeping if another thread calls `unpark()` first