22. Locks

Problem :

We would like to execute a series of instructions atomically,
but due to the presence of interrupts, we couldn't
=> LOCK is needed

thread

created by the programmer / scheduled by the OS
: Locks yield some of that control back to the programmer.
=> putting a lock aroud the code, the programmmer can guarantee that no more than a single thread can ever be active within that code
=> Locks help transform the chaos that is traditional OS scheduling into a more controlled activity

mutex : lock usesd by the POSIX library

How can we build an efficient lock

- provide **mutual exclusion**
- **fairness** : each thread contending for the lock get a fair shot at acquiring it once it is free (no starving)
- **performance** : the time overheads added by using lock

Controlling Interrupts

disable interrupts for critical sections
: one of the earliest solutions used to provice mutual exclusion

- invented for single-processor systems.

```
1  void lock() {
2      DisableInterrupts();
3  }
4  void unlock() {
5      EnableInterrupts();
6  }
```

Positive : simplicity

Negatives

- allow any calling thread to perform a privileged operation
  (turning interrupts on/off) => can you trust every calling threads?

- doesn't work on multiprocessors
  if multiple threads are running on different CPUs, and each try to enter the same critical section
  => doesn't matter whether interrupts are disabled
  interrupts are per CPU.
  => threads will be able to run on ther processors
  => could enter the critical section

- Turning off interrupts for extended periods of time can lead to interrupts becoming lost => can lead to serious systems problems

Used only in limited contexts as a mutual-exclusion primitive
Operating system itself will use interrupt masking to guarantee atomicity when accessing its own data structures / as the trust issue disappears inside the OS = always trusts itself to perform privileged operation

A Failed Attempt : Just Using Loads/Stores

why just using a single variable & accessing it via normal loads and stores is insufficient?

Correctness : No mutual exclusion

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**

Performance
the way a thread waits to acquire a lock that is already held:
endlessly checks the value of flag = **spin-waiting**

- the waste is exceptionally high on **a uniprocessor,**
  - the waiting thread using CPU during waiting
  - thread that holding lock cannot even run (at least, until a context switch occurs)!

=> system can get stuck

hardware support for locking : today all systems provide this type of support, even for single CPU systems

Building Working Spin Locks with Test-And-Set

Test-and-Set (or atomic exchange) instruction
: the simplest bit of hardware support

```
1  int TestAndSet(int *old_ptr, int new) {
2      int old = *old_ptr; // fetch old value at old_ptr
3      *old_ptr = new;     // store 'new' into old_ptr
4      return old;         // return the old value
5  }
```

hardware support :
the CPU guarantees that Test-and-Set function is performed atomically

it enables you to **"test"** the old value (which is what is returned)
while simultaneously **"setting"** the memory location to a new value

Making both the test (of the old lock value) and set (of the new value) a single atomic operation => ensure that only one thread acquires the lock

```
1  typedef struct __lock_t {
2      int flag;
3  } lock_t;
4
5  void init(lock_t *lock) {
6      // 0: lock is available, 1: lock is held
7      lock->flag = 0;
8  }
9
10 void lock(lock_t *lock) {
11     while (TestAndSet(&lock->flag, 1) == 1)
12         ; // spin-wait (do nothing)
13 }
14
15 void unlock(lock_t *lock) {
16     lock->flag = 0;
17 }
```

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

Spin lock is simplest type of lock to build
simply spins using CPU cycles, until the lock becomes available

For spin lock To work correctly on a single processor,
it requires a preemptive scheduler
= one that will interrupt a thread via a timer, in order to run a different thread, from time to time
Without preemption, spin locks don't work well on a single CPU, as a thread spinning on a CPU will never relinquish it.

Evalutating Spin Locks

correctness : does it provide mutual exclusion?
Yes. only allows a single thread to enter the critical section at a time

Fairness : not guaranteed. -> may lead to starvation
a thread spinning may spin forever, under contention.
no guarantee it will get the lock soon. just hoping and spinning.

Performance
considering

- with preemption
- there are multiple threads requesting the same lock

in the **single** CPU, performance overheads can be quite painful

- lock release get slower since it's interrupted by other threads
- each of threads will spin for the duration of a time slice before giving up the CPU (waiting for lock release), a waste of CPU cycles.

on **multiple** CPUs, works reasonably well
(if the number of threads roughly equals the number of CPUs)

- lock release get faster since it's less interrupted by other threads
- spinning to wait for a lock held on another processor doesn't waste many cycles => because it can proceed strictly parallel.

Compare-And-Swap (or compare-and-exchange) instruction

- hardware primitive

test whether the value at the address specified by ptr is equal to expected

```
1  int CompareAndSwap(int *ptr, int expected, int new) {
2      int original = *ptr;
3      if (original == expected)
4          *ptr = new;
5      return original;
6  }
```

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

compare-and-swap is a more powerful instruction than test-and-set
: allows conditional updates, based on current memory state

```
1  void lock(lock_t *lock) {
2      while (CompareAndSwap(&lock->flag, 0, 1) == 1)
3          ; // spin
4  }
```

here, if flag is not 0, flag is not changed. / in test-and-set, it changed.
=> avoid redundant write

Load-Linked and Store-Conditional

hardware primitive to build locks and other concurrent structures

Load-Linked : typical load instruction

simply fetches a value from memory & places it in a register

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

Store-Conditional : only succeeds (and updates the value stored at the address just load-linked from) if no intervening store to the address has taken place.

```
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
    }
}
```

```
1  void lock(lock_t *lock) {
2      while (1) {
3          while (LoadLinked(&lock->flag) == 1)
4              ; // spin until it's zero
5          if (StoreConditional(&lock->flag, 1) == 1)
6              return; // if set-to-1 was success: done
7                      // otherwise: try again
8      }
9  }
10
11 void unlock(lock_t *lock) {
12     lock->flag = 0;
13 }
```

Figure 28.6: **Using LL/SC To Build A Lock**

Fetch-And-Add

- hardware primitive

atomically increments a value while returning the old value at a particular address

```
1  int FetchAndAdd(int *ptr) {
2      int old = *ptr;
3      *ptr = old + 1;
4      return old;
5  }
```

Ticket Lock

instead of a single value, uses a ticket and turn variable in combination to build a lock

```
1  typedef struct __lock_t {
2      int ticket;
3      int turn;
4  } lock_t;
5
6  void lock_init(lock_t *lock) {
7      lock->ticket = 0;
8      lock->turn   = 0;
9  }
10
11 void lock(lock_t *lock) {
12     int myturn = FetchAndAdd(&lock->ticket);
13     while (lock->turn != myturn)
14         ; // spin
15  }
16
17 void unlock(lock_t *lock) {
18     lock->turn = lock->turn + 1;
19 }
```

Figure 28.7: **Ticket Locks**

- ticket : total number of thread requiring lock. / total
- turn : current thread idx to hold lock / progress

Fairness guaranteed : ensures progress for all threads

Once a thread is assigned its ticket value, it will be scheduled at some point in the future.
/ a thread spinning on test-and-set could spin forever even as other threads acquire and release the lock

Less Code is better Code

if the same people had twice as much time,
they could produce as good of a system in half the code

Spinning : big waste

wastes an entire time slice doing nothing but checking a value that isn't going to change

Problem gets worse with N threads contending for a lock

N-1 time slices may be wasted in a similar manner (consider round robin scheduling)
simply spinning and waiting for a single thread to release the lock

Spinning : Correctness : Priority Inversion

Having high priority just ain't what it used to be.

what if lower priority thread holds the lock?
=> higher priority thread should wait
scheduler even doesn't give many CPU time to lower priority thread to end its' job and return the lock. since priority is lower than others.
=> more waiting until lock released.

Solution : **Priority inheritance**
a higher-priority thread waiting for a lower-priority thread can temporarily boost the lower thread's priority => enabling it to run and overcoming the inversion

or ensure all threads have the same priority

How to avoid Spinning

: How can we develop a lock that doesn't needlessly waste time spinning on the CPU?
=> Hardware support alone (working locks, fairness in lock acquisition) cannot solve the problem.
: What to do when a context switch occurs in a critical section, and threads start to spin endlessly, waiting for the interrupted (lock-holding) thread to be run again?
=> OS support

A Simple Approach : Just Yield

Don't spin. Just give up the CPU to another thread
yield() / an OS primitive
: a thread can call when it wants to give up the CPU and let another thread run (running -> ready) = **deschedules itself**

- Thread's state : running / ready / blocked

But run-and-yield approach is **still costly**
1. the cost of a **context switch** can be substantial => plenty of waste
   continuously do more context switch cause
   they are ready & but yielding
2. doesn't address **starvation**
   a thread may get caught in an endless yield loop while other threads repeatedly enter and exit the critical section

Using Queues : Sleeping Instead of Spinning

we must explicitly exert some control over which thread next gets to acquire the lock after the current holder releases it.
To do this, we need more OS support, as well as a queue to keep track of which threads are waiting to acquire the lock.

- park() : to put a calling thread to sleep
- unpark(threadID) : to wake a particular thread

```
1  typedef struct __lock_t {
2      int flag;
3      int guard;
4      queue_t *q;
5  } lock_t;
6
7  void lock_init(lock_t *m) {
8      m->flag  = 0;
9      m->guard = 0;
10     queue_init(m->q);
11 }
12
13 void lock(lock_t *m) {
14     while (TestAndSet(&m->guard, 1) == 1)
15         ; //acquire guard lock by spinning
16     if (m->flag == 0) {
17         m->flag = 1; // lock is acquired
18         m->guard = 0;
19     } else {
20         queue_add(m->q, gettid());
21         m->guard = 0;
22         park();
23     }
24 }
25
26 void unlock(lock_t *m) {
27     while (TestAndSet(&m->guard, 1) == 1)
28         ; //acquire guard lock by spinning
29     if (queue_empty(m->q))
30         m->flag = 0; // let go of lock; no one wants it
31     else
32         unpark(queue_remove(m->q)); // hold lock
33                                     // (for next thread!)
34     m->guard = 0;
35 }
```

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

guard : spin-lock around the flag / flag = current lock status (free or not)
m->guard = 0, then park(). else, you are holding spin-lock while u sleep

Entirely avoiding spin-waiting is not possible here
: a thread might be interrupted while acquiring or releasing the spin-lock (guard)
but once it get into spin-lock (guard),
trying to requiring actual lock (flag) and that lock is not available,
it goes to sleep not spin

=> However, the time spent spinning is quite limited
: just a few instructions inside the lock and unlock code for guard,
instead of the user-defined critical section (flag)

=> this approach may be reasonable

When unlock, we're not setting the critical section lock (flag) available

- critical section lock(flag) is set available when queue empty
- we just pass the lock directly from the thread releasing the lock to the next thread acquiring it. flag == 1
- waken (unparked) thread just proceed code which need lock. this is possible because no other thread is spinning or running concurrently trying to acquire the lock. They are all sleeping.

Wakeup / Waiting race

situation : thread 0 is holding the lock

- queue : []
- flag : 1
- guard : 0

thread 0) trying to request lock
thread 0) guard == 0, but flag == 1 => lock is not available
thread 0) added to queue => queue : [thread 0]
thread 0) after set guard = 0, about to park() to wait lock release

- context switch -

thread 1) was holding the lock
thread 1) released the lock -> entering to unlock
thread 1) since guard == 0, and queue is not empty,
unpark the first thread in queue : thread 0 * not even parked!
thread 1) set guard = 0

- context switch -

thread 0) run park() : sleep FOREVER

=> To Solve this

1. by adding a system call : **setpark()**
a thread can indicate it is about to park.

=> if it then happens to be interrupted and another thread calls unpark before park is acutually called, the subsequent park returns immediately instead of sleeping.

```
1       queue_add(m->q, gettid());
2       setpark(); // new code
3       m->guard = 0;
```

2. Let the kernel manage the **guard** and coordination
=> make "release the lock & wake up the next thread" atomic

Different OS, Different Support

futex : in-kernel functionality provided by Linux

- each futex has associated with it a specific physical memory location per-futex in-kernel queue
- Callers can use futex calls to sleep and wake as need be

- futex_wait (address, expected) : puts the calling thread to sleep
  - assuming the value at the address is equal to expected
- futex_wake (address) : wakes one thread that is waiting on the queue

***mutex is a 32-bit int used to encode two things:**

- Bit 31 (the highest bit): 1 → Locked -> v : - / 0 → Unlocked -> v : +
- Bits 0–30 (lower 31 bits): Encodes the waiter count, i.e., how many threads are interested in the lock.

atomic_bit_test_set : set the bit at Nth.

```
1  void mutex_lock (int *mutex) {
2    int v;
3    // Bit 31 was clear, we got the mutex (fastpath)
4    if (atomic_bit_test_set (mutex, 31) == 0)
5      return;
6    atomic_increment (mutex);
7    while (1) {
8        if (atomic_bit_test_set (mutex, 31) == 0) {
9            atomic_decrement (mutex);
10           return;
11       }
12       // Have to waitFirst to make sure futex value
13       // we are monitoring is negative (locked).
14       v = *mutex;
15       if (v >= 0)
16         continue;
17       futex_wait (mutex, v);
18   }
19 }
20
21 void mutex_unlock (int *mutex) {
22   // Adding 0x80000000 to counter results in 0 if and
23   // only if there are not other interested threads
24   if (atomic_add_zero (mutex, 0x80000000))
25     return;
26
27   // There are other threads waiting for this mutex,
28   // wake one of them up.
29   futex_wake (mutex);
30 }
```

Figure 28.10: **Linux-based Futex Locks**

Two-Phase Locks

spinning can be useful, particularly if the lock is about to be released.

- Sleeping and waking up (e.g., via park()/futex_wait()) involves **expensive context switches and scheduler involvement**, which can be costlier than just waiting a few CPU cycles.

So in the **first phase**, the lock spins for a while, hoping that it can acquire the lock.

If the lock is not acquired during the first spin phase => enter to second spin

Second Spin
the caller is put to sleep, and only woken up when the lock becomes free later.

Linux lock above is a form of such a lock, but it only spins once.