Universidade Estadual de Campinas
Instituto de Computação
MC504 Sistemas Operacionais

T18 Thread Synchronization 2

Referência principal
Ch. 28 of *Operating Systems: Three Easy Pieces* by Remzi and Andrea Arpaci-Dusseau (pages.cs.wisc.edu/~remzi/OSTEP/)

Discutido em classe em 26 de setembro de 2018

Arthur João Catto, PhD

2º semestre de 2018

Can we build an efficient lock?

Is there any required hardware support?

Any required OS support?

A First Attempt: Controlling Interrupts

- Interrupts can be used to control access to critical sections.
  - This is a simple solution and one of the earliest means to provide mutual exclusion.
  - It was conceived for single-processor systems.

```
1    void lock() {
2        disable_interrupts();
3    }
4
5    void unlock() {
6        enable_interrupts();
7    }
```

Would you have any issues with this proposal?

Interrupt Control Raises Many Issues. . .

- It allows a thread to execute a privileged operation.
  - It is unwise to put too much trust in applications.
  - Greedy (or malicious) program could monopolize the processor.
- It creates a centralized point of control for all critical sections of a program.
- It does not work on multiprocessors.
- Interrupts may be lost if their control is disabled for long periods of time.
- In modern CPUs, instructions that mask or unmask interrupts tend to be slower than others.

Why do we need hardware support?

- Let's try to use a software *flag* to indicate whether the lock is held or not.

```
1   typedef struct __lock_t {
2       int flag;
3   } lock_t;
4
5   void init(lock_t *mutex) {
6       // 0 = lock available, 1 = lock held
7       mutex->flag = 0;
8   }
9
10  void lock(lock_t *mutex) {
11      while (mutex->flag == 1) // TEST the flag
12          ; // spin-wait (do nothing)
13      mutex->flag = 1; // now SET it !
14  }
15
16  void unlock(lock_t *mutex) {
17      mutex->flag = 0;
18  }
```

Do you have any issues with this code?

Our software attempt has two problems . . .

1. **Correctness**: There is no *mutual exclusion* (assume `flag == 0` initially)

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;`	

2. **Efficiency:** Spin-waiting wastes CPU time just waiting for another thread.

Getting Some Help from Hardware: Test-and-Set

- Since disabling interrupts and software flags did not work, system designers started to look for hardware support.
  - In the early 60's the Burroughs B5000 provided a simple solution, which is still widely adopted today, even for single CPU systems.
- Semantics of an atomic instruction to support the creation of simple locks:

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

  - It fetches the value pointed to by `ptr`, sets that value to `new` and returns the fetched value.
  - This sequence of operations is performed atomically.

A Working Spin Lock using TestAndSet

- To work correctly on *a single processor*, this solution requires a preemptive scheduler.

```
1   typedef struct __lock_t {
2       int flag;
3   } lock_t;
4
5   void init(lock_t *lock) {
6       // 0 = lock available, 1 = lock held
7       lock->flag = 0;
8   }
9
10  void lock(lock_t *lock) {
11      while (TestAndSet(&lock->flag, 1) == 1)
12          ;    // spin-wait
13  }
14
15  void unlock(lock_t *lock) {
16      lock->flag = 0;
17  }
```

Could you schedule the threads in such a way that it would not work?

Evaluating the Basic Spin Lock

- Correctness ✔
  - It allows only a single thread to enter the critical section at a time.
- Fairness ✘
  - It does not provide any guarantees of fairness.
  - Indeed, a spinning thread may spin forever.
- Performance ?
  - In the single CPU, performance overheads can be quite painful.
  - If the number of threads roughly equals the number of CPUs, spin locks work reasonably well.

Another Atomic Primitive: Compare-and-Swap

- Test whether the value at the address `ptr` is equal to `expected`.
  - *If so*, update the memory location pointed to by `ptr` with the `new` value.
  - *In either case*, return the actual value at that memory location.

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

- This is enough for us to write a spin lock…

```
8   void lock(lock_t *lock) {
9       while (CompareAndSwap(&lock->flag, 0, 1) == 1)
10          ; // spin
11  }
```

MIPS, Alpha, PowerPC, ARM

Yet Another: Load-Linked and Store-Conditional

```
1.  int LoadLinked(int *ptr) {
2.      return *ptr;
3.  }

4.  int StoreConditional(int *ptr, int value) {
5.      if (no one has updated *ptr since the LoadLinked to this address) {
6.          *ptr = value;
7.          return 1; // success!
8.      } else {
9.          return 0; // failed to update
10.     }
11. }
```

- The store-conditional *only succeeds* if no intermittent store to the address has taken place.
  - **Success** → return `1` and update the value at `ptr` to `value`.
  - **Fail** → the value at `ptr` is not updated and `0` is returned.

A Lock Using Load-Linked and Store-Conditional

```
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-it-to-1 was a success: all done
7.                          // otherwise: try it all over again
8.      }
9.  }

10. void unlock(lock_t *lock) {
11.     lock->flag = 0;
12. }
```

A Shorter Lock Using LoadLinked and StoreConditional

```
1. void lock(lock_t *lock) {
2.     while (LoadLinked(&lock->flag) ||
3.            !StoreConditional(&lock->flag, 1))
4.         ;     // spin
5. }

6. void unlock(lock_t *lock) {
7.     lock->flag = 0;
8. }
```

Are both solutions equivalent?

And A Last One: Fetch-and-Add

- Atomically increment the value pointed to by **ptr**, while returning the **old** value.

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

- **FetchAndAdd** allows us to implement a so-called *ticket lock* (Mellor-Crummey and Scott, 1991), that can guarantee progress for all threads, as we'll see next.

Ticket Lock: Ensuring progress for all threads

```
1.  typedef struct __lock_t {
2.      int ticket;
3.      int turn;
4.  } lock_t;

5.  void lock_init(lock_t *lock) {
6.      lock->ticket = 0;
7.      lock->turn = 0;
8.  }

9.  void lock(lock_t *lock) {
10.     int myturn = FetchAndAdd(&lock->ticket);
11.     while (lock->turn != myturn)
12.         ; // spin
13. }

14. void unlock(lock_t *lock) {
15.     FetchAndAdd(&lock->turn);
16. }
```

Do you understand how this solution works?

Can you tell the main difference between this solution and the former ones?

Can All That Spinning Be Avoided?

- Hardware-based spin locks are simple and they work.
  - However, in some cases, such solutions can be quite inefficient.

Can you tell why?

- Any time a thread gets caught spinning, it wastes an entire time slice doing nothing but checking a value.

What might happen when there are n concurrent threads?

Can we develop a lock that does not needlessly waste time busy waiting on the CPU?

A Simple Approach: Just Yield Control

- When you are going to spin, give up the CPU to another thread.
  - OS system call moves the caller from the *running* to the *ready state*.

```
1.  void init(lock_t *ptr) {
2.      ptr->flag = 0;
3.  }

4.  void lock(lock_t *ptr) {
5.      while (TestAndSet(&ptr->flag, 1) == 1)
6.          yield();     // give up the CPU
7.  }

8.  void unlock(lock_t *ptr) {
9.      ptr->flag = 0;
10. }
```

- With many threads, context switching can become expensive, while the starvation problem still remains.

Using Queues: Sleeping Instead of Spinning

- Our previous approaches leave too much to chance.
- To exert some control over which thread will acquire the lock after its current owner releases it we need some OS support.
- First, let's create a queue to keep track of which threads are waiting to enter the lock.
- Second, let's adopt two primitives from Solaris (1993)
  - `park()` → put a calling thread to sleep
  - `unpark(threadID)` → wake up the particular thread designated by `threadID`.

Lock With Queues, Test-and-set, Yield and Wakeup

```
1.  typedef struct __lock_t {
2.      int flag;
3.      int guard;
4.      queue_t *q; }
5.  lock_t;

6.  void lock_init(lock_t *m) {
7.      m->flag = 0;
8.      m->guard = 0;
9.      queue_init(m->q);
10. }

11. ...
```

Lock With Queues, Test-and-set, Yield and Wakeup

```
11. void lock(lock_t *m) {
12.     while (TestAndSet(&m->guard, 1) == 1)
13.         ; // acquire guard lock by spinning
14.     if (m->flag == 0) {
15.         m->flag = 1; // lock is acquired
16.         m->guard = 0;
17.     } else {
18.         queue_add(m->q, gettid());
19.         m->guard = 0;
20.         park();
21.     }
22. }

23. ...
```

Lock With Queues, Test-and-set, Yield and Wakeup

```
23. void unlock(lock_t *m) {
24.     while (TestAndSet(&m->guard, 1) == 1)
25.         ; // acquire guard lock by spinning
26.     if (queue_empty(m->q))
27.         m->flag = 0; // let go of lock; no one wants it
28.     else
29.         unpark(queue_remove(m->q)); // hold lock (for next thread!)
30.     m->guard = 0;
31. }
```

Some interesting things about the queueing model

1. Why did we combine test-and-set and the waiting queue?
2. Why did we use a waiting queue at all?
3. What is the purpose of the `guard` field in `lock_t`?
4. Aren't there spin-locks in `lock()` and `unlock()`?
   Isn't that what we were trying to eliminate?
5. What happens in `lock()` when a thread cannot acquire the lock?
6. In `lock()`, what if the lock release came after `park()` instead of before?
7. Look at `park()` – line 20 – carefully. Can you spot a threat nearby?

An unexpected race condition. . .

- There is a race condition just before the `park()` call on line 20.
  - Let's see what could happen there. Assume that. . .
    - There are two threads A and B.
    - Thread A is running but is descheduled just before calling `park()`.
    - Control is switched to thread B which is holding the lock.
    - Thread B releases the lock, finds the queue empty and does nothing more.

    Thread A will now sleep forever. . .
- This problem is sometimes called the **wakeup/waiting race.**

How to cure the wakeup/waiting race

- **Solaris** solved this problem by adding a third system call: **`setpark()`**.
  - By calling this routine, a thread indicates it *is about to* park.
  - If it happens to be interrupted and another thread calls **`unpark()`** before **`park()`** is actually called, the subsequent **`park()`** returns immediately instead of sleeping.
- Only a small change is needed within **`lock()`**…

```
17.     … else {
18.         queue_add(m->q, gettid());
19.
20.         m->guard = 0;
21.         park();
22.     }
```

Other approaches: Futex

- Linux provides a `futex` which is similar to Solaris' `park` and `unpark`.
  - `futex_wait(address, expected)`
    - Put the calling thread to sleep, if the value at `address` is equal to `expected`.
    - Otherwise, the call returns immediately.
  - `futex_wake(address)`
    - Wake up one thread that is waiting on the queue.
- The code snippet of the next slides comes from `lowlevellock.h` in the `nptl` library
  - The high bit of the integer `v` tracks whether the lock is held or not.
  - All the other bits give the number of waiters.

Implementing `mutex_lock`

```
1.  void mutex_lock(int *mutex) {
2.      int v;
3.      /* Bit 31 was clear, we got the mutex (this is the 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.         /* We have to wait now. First make sure the futex value
13.          we are monitoring is truly negative (i.e. locked). */
14.         v = *mutex;
15.         if (v >= 0)
16.             continue;
17.         futex_wait(mutex, v);
18.     }
19. }
```

Implementing `mutex_unlock`

```
1. void mutex_unlock(int *mutex) {
2.     /* Adding 0x80000000 to the counter results in 0 if and only if
3.      there are no other interested threads */
4.     if (atomic_add_zero(mutex, 0x80000000))
5.         return;
6.     /* There are other threads waiting for this mutex,
7.      wake one of them up */
8.     futex_wake(mutex);
9. }
```

Two-Phase Locks

- A two-phase lock realizes that spinning can be useful if the lock *is about to* be released.
  - **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, a second phase is entered,
  - **Second phase**
    - The caller is put to sleep.
    - The caller is only woken up when the lock becomes free later.

Rules for Using Locks

- Every lock is initially free.
- Always acquire a lock before accessing the shared data structure it protects.
  - Do it at the beginning of procedure!
- Always release a lock after finishing with the protected shared data.
  - Do it at the end of procedure!
  - Only the holder can release a lock.
  - DO NOT throw a lock for someone else to release.
- Never access shared data without a protecting lock.
  - It is DANGEROUS!