Sistemas Operativos: Concurrency

Locks

Pedro F. Souto (pfs@fe.up.pt)

March 18, 2020

Roadmap

Using locks to prevent race conditions

Question Is there a systematic way to prevent race conditions?

Answer The closest is the concept of *monitor*

- Use *abstract data types* (ADT), a kind of class, to structure your code;
- Add a lock to ensure mutual exclusion in the execution of the functions of the ADT:
  - The first step in each function is to acquire that lock
  - The last step in each function is to release that lock

The Counter ADT

```
typedef struct __counter_t {
    int value;
} counter_t;

void init(counter_t *c) {
    c->value = 0;
}

void increment(counter_t *c) {
    c->value++;
}

void decrement(counter_t *c) {
    c->value--;
}

int get(counter_t *c) {
    return c->value;
}
```

The Counter Monitor

```
typedef struct __counter_t {
    int               value;
    pthread_mutex_t lock;
} counter_t;

void init(counter_t *c) {
    c->value = 0;
    Pthread_mutex_init(&c->lock, NULL);
}

void increment(counter_t *c) {
    Pthread_mutex_lock(&c->lock);
    c->value++;
    Pthread_mutex_unlock(&c->lock);
}

void decrement(counter_t *c) {
    Pthread_mutex_lock(&c->lock);
    c->value--;
    Pthread_mutex_unlock(&c->lock);
}
```

Monitors

- This is not the only way to design thread-safe programs
- Actually, the use of monitors may raise some modularity issues
  - Namely deadlocks as we will see later
- Furthermore, the performance may not be the best
  - If code uses several ADTs, may need to acquire the locks on all of them
- Nevertheless, this is a rather useful pattern
  - `java.util.concurrent` is a Java package, i.e. library, that provides thread-safe versions of many classes in the `java.util` package
  - Its implementation relies on the concept of monitor
- Problem 2 of this week's problem set asks you to develop a thread-safe stack using this pattern

Roadmap

Locks Usage

Implementation

Implementation Goals

Mutual exclusion that is the main purpose of locks

Fairness threads should have a fair chance to acquire a lock

- This is not always desired
- But at least threads should not starve

Performance There many facets of this

Without contention

With contention On uniprocessors vs. multiprocessors/multicores

Approaches

- Software-based solutions
  - More of an intellectual exercise: they are not efficient
- Controlling Interrupts
- Atomic read-modify-write instructions

Controlling Interrupts

Idea Disable interrupts upon entering a critical section and re-enable them upon exit

```
void lock() {
DisableInterrupt();
}
void unlock() {
EnableInterrupt();
}
```

Rationale By disabling interrupts, the thread will execute without interruption the critical section.

Advantage This is a simple solution that works

Controlling Interrupts: Issues

Requires the OS to trust applications If the application does not call `unlock()` the OS will not be able to regain control

- The only way out is to restart the system

Reduces responsiveness while the interrupts are disabled, the system cannot respond to interrupts

- May lead to lost interrupts, e.g. of the timer

Only works on uniprocessors

- Nothing prevents threads executing on other cores from entering interfering critical sections
- Furthermore, disabling interrupts operates on a single core;

Low performance

- On modern HW, interrupt-related instructions are slower than atomic read-modify-write instructions

Concluding On modern OSs controlling interrupts is done only at the kernel, mostly to prevent interrupt handling code from being interrupted.

Attempt to Implement Locks with Ordinary Instructions

```
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;
}
```

Issues

Performance threads must busy wait

Correctness i.e. does not ensure mutual exclusion always – can you see why?

Attempt to Implement Locks with Ordinary Instructions

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!)	

Issues

Solution Use atomic read-modify-write instructions

Atomic Exchange (Test-And-Set)

- This is the simplest atomic read-modify-write instruction: in the Intel32 ISA it is known as `xchg`

```
int TestAndSet(int *old_ptr, new) {
    int old = *old_ptr;
    *old_ptr = new;
    return old;
}
```

  - Remzi called it `TestAndSet()` because it allows testing the value and possibly modifying it (but there are instructions that do exactly that, Remzi calls them `CompareAndSwap()`)
- The key is that this instruction is atomic:
  - The reading and the modification of the memory whose address is `old_ptr` is done in an indivisible way

Question How can we use this instruction to solve the problem with our last attempt?

Spin-Lock Implementation with Atomic TestAndSet

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

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

Analysis

Entering a CS when the lock is not held

Entering a CS when the lock is held by another thread

Spin-Lock with Atomic TestAndSet: Evaluation

Correctness as long as the scheduler is preemptive

Fairness there is no guarantee

- Depends on the scheduler

Performance

Uniprocessor Poor

- While a thread spins waiting, other threads cannot run

Multiprocessor May be good, if:

- There is low contention
- Lock is held by a thread on another core/processor
- Critical section is short

Spin-Lock Performance Issues

- If the thread inside a CS is preempted/interrupted
  - The thread trying to enter the CS will be forced to spin
    - Possibly until it is preempted
- The higher the contention, i.e. the higher the number of threads trying to enter the CS, the worse the performance.

Solution (first try) Yield the CPU if lock is held by another thread

Lock with `yield()`

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

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

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

Issues

Performance On high contention

- Too many `yield()`'s

Fairness depends on the scheduler

Locks: Avoiding Busy-Waiting

- Need more control on which thread gets the lock
  - Requires OS support
- E.g. Solaris offers two system-calls:

  `park()` similar to `sleep()`

  `unpark()` kind of wakeup

Idea Before "parking" add the thread to a queue of threads waiting for the lock

- Upon unlocking, "unpark" the thread at the head of the queue

Lock with qeues and `park()`

```
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 t
    m->guard = 0;
}
```

- But this has a race-condition known as **lost-wakeup**

Fixing the lost-wakeup or wakeup/waiting race

- Use the `setpark()` system call:
  - It tells the OS that the thread is about to call `park()` (may be `prepare_park()` would be clearer)

```
queue_add(m->q, gettid());
setpark(); // tell the kernel thread is abou
m->guard = 0;
park();
```

  - If there is a call to `unpark()` between `setpark()` and `park()`, the latter returns immediately
- `park()/unpark()/setpark()` are OS-specific
- Linux offers the `futex()` system call with the same purpose:

`futex_wait(address, expected)` blocks the thread, if the value @ `address` is `expected` (otherwise, does not block);

`futex_wake(address)` wakes one thread that is waiting on `address`

mutex_lock() with futex (lowlevellock.h)

```
void mutex_lock (int *mutex) {
  int v;
  /* Bit 31 was clear, we got the mutex (this is the fastpath)  */
  if (atomic_bit_test_set (mutex, 31) == 0)
    return;
  atomic_increment (mutex);
  while (1) {
      if (atomic_bit_test_set (mutex, 31) == 0) {
          atomic_decrement (mutex);
          return;
      }
      /* We have to wait now. First make sure the futex value
         we are monitoring is truly negative (i.e. locked). */
      v = *mutex;
      if (v >= 0)
        continue;
      futex_wait (mutex, v);
  }
}
void mutex_unlock (int *mutex) {
  /* Adding 0x80000000 to the counter results in 0 if and only if
    there are not other interested threads */
  if (atomic_add_zero (mutex, 0x80000000))
    return;

  /* There are other threads waiting for this mutex,
     wake one of them up.  */
  futex_wake (mutex);
}
```