Concurrency: Locks

CS 537: Introduction to Operating Systems

Louis Oliphant

University of Wisconsin - Madison

Fall 2023

Administrivia

- Project 4 due Oct 24th @ 11:59pm
- Exam 1
  - Taking me a bit to upload grades into Canvas, should be done over the weekend

Review: Threads

A **thread** is similar to a process in that it is a point of execution and can be scheduled. The main difference is threads **share virtual address space** (code and heap data). Each thread has its own call stack. Understand the **race condition** between threads accessing shared data.

Remember how to:

- Create a thread
- Wait for a thread to finish executing
- Pass arguments to a thread
- Get return values after a thread has finished

Review Threads: looping-race-nolock.s

```
# assumes %bx has loop count in it
.main
.top
# critical section
mov 2000, %ax  # get 'value' at address 2000
add $1, %ax    # increment it
mov %ax, 2000  # store it back

# see if we're still looping
sub  $1, %bx
test $0, %bx
jgt .top

halt
```

Review: x86.py

```
./x86.py -p looping-race-nolock.s -t 1 -a bx=2

 2000      ax          Thread 0
    0       0
    0       0   1000 mov 2000, %ax
    0       1   1001 add $1, %ax
    1       1   1002 mov %ax, 2000
    1       1   1003 sub  $1, %bx
    1       1   1004 test $0, %bx
    1       1   1005 jgt .top
    1       1   1000 mov 2000, %ax
    1       2   1001 add $1, %ax
    2       2   1002 mov %ax, 2000
    2       2   1003 sub  $1, %bx
    2       2   1004 test $0, %bx
    2       2   1005 jgt .top
    2       2   1006 halt
```

```
./x86.py -p looping-race-nolock.s -t 2 -a bx=2
2000      ax          Thread 0                Thread 1
    0       0
    0       0   1000 mov 2000, %ax
    0       1   1001 add $1, %ax
    1       1   1002 mov %ax, 2000
    1       1   1003 sub  $1, %bx
    1       1   1004 test $0, %bx
    1       1   1005 jgt .top
    1       1   1000 mov 2000, %ax
    1       2   1001 add $1, %ax
    2       2   1002 mov %ax, 2000
    2       2   1003 sub  $1, %bx
    2       2   1004 test $0, %bx
    2       2   1005 jgt .top
    2       2   1006 halt
    2       0   ----- Halt;Switch -----  ----- Halt;Switch -----
    2       2                            1000 mov 2000, %ax
    2       3                            1001 add $1, %ax
    3       3                            1002 mov %ax, 2000
    3       3                            1003 sub  $1, %bx
    3       3                            1004 test $0, %bx
    3       3                            1005 jgt .top
    3       3                            1000 mov 2000, %ax
    3       4                            1001 add $1, %ax
    4       4                            1002 mov %ax, 2000
    4       4                            1003 sub  $1, %bx
    4       4                            1004 test $0, %bx
    4       4                            1005 jgt .top
    4       4                            1006 halt
```

```
./x86.py -p looping-race-nolock.s -t 2 -a bx=2

2000 ax Thread 0                Thread 1
0    0
0    0  1000 mov 2000, %ax
0    1  1001 add $1, %ax
1    1  1002 mov %ax, 2000
1    1  1003 sub  $1, %bx
1    1  1004 test $0, %bx
1    1  1005 jgt .top
1    0  -- Interrupt ------  -- Interrupt -----
1    1                       1000 mov 2000, %ax
1    2                       1001 add $1, %ax
2    2                       1002 mov %ax, 2000
2    2                       1003 sub  $1, %bx
2    2                       1004 test $0, %bx
2    1  -- Interrupt ------  -- Interrupt -----
2    2  1000 mov 2000, %ax
2    3  1001 add $1, %ax
3    3  1002 mov %ax, 2000
3    2  -- Interrupt ------  -- Interrupt -----
3    2                       1005 jgt .top
3    3                       1000 mov 2000, %ax
```

```
2000 ax Thread 0                Thread 1
3    2  -- Interrupt ------  -- Interrupt -----
3    2                       1005 jgt .top
3    3                       1000 mov 2000, %ax
3    3  -- Interrupt ------  -- Interrupt -----
3    3  1003 sub  $1, %bx
3    3  1004 test $0, %bx
3    3  1005 jgt .top
3    3  1006 halt
3    3  -- Halt;Switch ----- -- Halt;Switch ---
3    3  -- Interrupt ------  -- Interrupt -----
3    4                       1001 add $1, %ax
4    4                       1002 mov %ax, 2000
4    4                       1003 sub  $1, %bx
4    4  -- Interrupt ------  -- Interrupt -----
4    4                       1004 test $0, %bx
4    4                       1005 jgt .top
4    4                       1006 halt
```

Quiz 9: Intro to Threads

https://tinyurl.com/cs537-fa23-q9

Locks (Programmer's Perspective)

```
#include <pthread.h>

pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x=x+1; //or whatever your critical section is
pthread_mutex_unlock(&lock);
```

Use different locks to protect different variables / data structures

Rest of lecture is to understand how locks are built (hardware and OS support)
Next lecture on using locks with different data structures

Lock Implementation Goals

1. **Correctness**
    - *mutual exclusion*
      Only one thread in critical section at a time
    - *Progress* (deadlock-free)
      If several simultaneous requests, must allow one to proceed
    - *Bounded* (starvation-free)
      Must eventually allow each waiting thread to enter
2. **Fairness** – Each thread waits for same amount of time
3. **Performance** – CPU is not used unnecessarily
    - case 1 – no contention
    - case 2 – multiple threads contending, single CPU
    - case 3 – multiple threads contending, multiple CPUs

Earliest Solution: Disable Interrupts

```
void lock() {
   DisableInterrupts();
}

void unlock() {
   EnableInterrupts();
}
```

On single CPU, thread assured no other thread will interfere (including OS)

This approach used sparingly by OS itself

Disadvantages

- User program has control of CPU, could `lock()` and run forever
- Doesn't work on multiprocessor systems
- Can lead to lost interrupts (imagine OS not being notified of I/O completion)
- Very inefficient

Failed Attempt: Using Loads/Stores

Why doesn't this work?

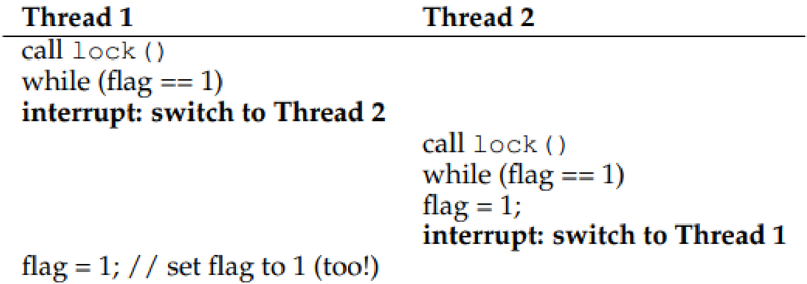
```
typedef struct __lock_t {int flag; } lock_t;

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
      ;
   mutex->flag = 1;
}

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

Failed Attempt: Reason



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!)	

No Mutual Exclusion!

Wasteful **Spin-waiting**

Hardware Support: Test-and-Set

on x86 it is the locked version of the atomic exchange (`xchg`)

Happens Atomically:

```
//xchg(int *addr, int newVal)
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
}
```

```
movl 4(%esp), %edx
movl 8(%esp), %eax
xchgl (%edx), %eax
ret
```

Lock Implementation with TestAndSet

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

void init(lock_t *lock) {
   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;
}
```

Other Atomic HW Instructions

```
int CompareAndSwap(int *addr, int expected, int new) {
   int actual = *addr;
   if (actual == expected)
      *addr = new;
   return actual;
}
```

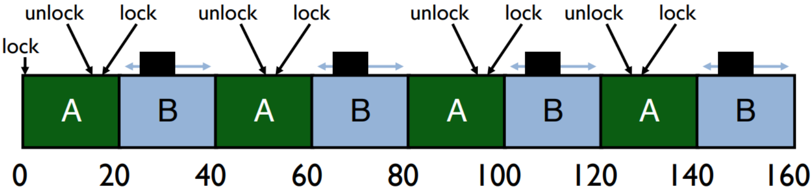
```
void lock(lock_t *lock) {
   while (CompareAndSwap(&lock->flag,0,1)== 1)
      ; //spin
}
```

A Pair of Atomic HW Instructions

```
int LoadLinked(int *ptr) {
   return *ptr;
}
int StoreConditional(int *ptr, int value) {
   if (no one has updated *ptr since the LoadLinked to this address) {
      *ptr = value;
      return 1; //success
   } else {
      return 0; //failed to update
   }
}
```

```
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-it-to-1 was a success: all done
                  //otherwise: try it all over again
   }
}
void unlock(lock_t *lock) {
   lock->flag = 0;
}
```

Basic Spinlocks Are Unfair



Scheduler is unaware of locks/unlocks!

Fairness: Ticket Locks – Based on Atomic HW Instruction

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

```
typedef struct __lock_t {
   int ticket; //thread's ticket number
   int turn; //whose turn it is
}
void lock_init(lock_t *lock) {
   lock->ticket = 0;
   lock->turn = 0;
}
void lock(lock_it *lock) {
   //first, reserve this thread's turn
   int myturn = FetchAndAdd(&lock->ticket);
   while (lock->turn != myturn)
      ; //spin until thread's turn
}

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

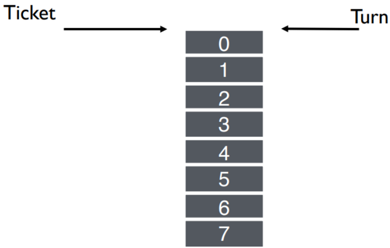
Ticket Lock Example

```
A lock():
B lock():
C lock():

A unlock():

A lock():
B unlock():

C unlock():
A unlock():
```



Spinlock Performance

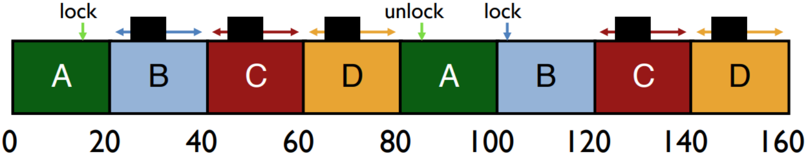
Fast when...

- many CPUs
- locks held a short time
- advantage: avoid context switch

Slow when...

- one CPU
- locks held a long time
- disadvantage: spinning is wasteful

CPU Scheduler is Ignorant of Spinlocks



CPU scheduler may run **B,C,D** instead of **A** even though **B,C,D** are waiting for **A**

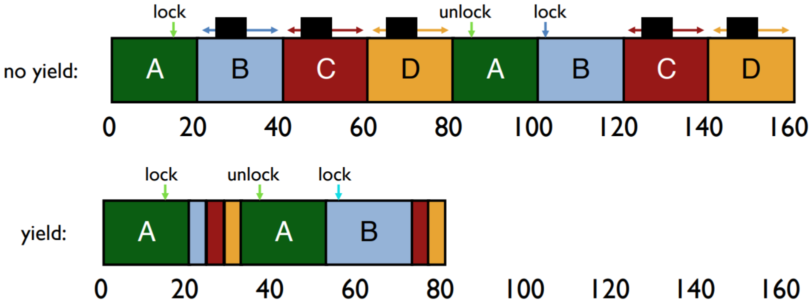
Ticket Lock With Yield (OS Call)

Instead of spinning, give up CPU with special yield() instruction

```
typedef struct __lock_t {
   int ticket; //thread's ticket number
   int turn; //whose turn it is
}
void lock_init(lock_t *lock) {
   lock->ticket = 0;
   lock->turn = 0;
}
void lock(lock_it *lock) {
   int myturn = FetchAndAdd(&lock->ticket); // Reserve turn
   while (lock->turn != myturn)
      yield(); //give up rest of time-slice
}

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

Yield Instead of Spin



Spinlock Performance

Waste of CPU cycles?

- Without yield: O(threads * **time_slice**)
- With yield: O(threads * **context_switch**)

Even with yield, spinning is slow with high thread contention

Next improvement: put thread on waiting queue and block instead of spinning

New OS call:

- `park()` – put calling thread to sleep
- `unpark()` – wake a particular thread

Lock Implementation

```c
typedef struct __lock_t {
   int flag;
   int guard;
   queue_t *q;
}

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();
   }
}
```

```c
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 for next
   m->guard = 0;
}
```

What would happen if release of guard came *after* the `park()`?

Think about possible **wakeup/waiting race condition** just before the call to park()

Add `setpark()` OS call to indicate *about* to park(). Add call to setpark() just before releasing guard:

```c
queue_add(m->q, gettid());
setpark();
m->guard = 0;
park();
```

Spin-Waiting vs. Blocking

Each approach is better under different circumstances:

- Uniprocessor
    - Waiting process is scheduled -> Process holding lock is not
    - Waiting process should always relinquish processor
    - Associate queue of waiters with each lock (as in previous implementation)
- Multiprocessor
    - Waiting process is scheduled -> Process holding lock might be
    - spin or block depends on how long, **t**, before lock is released:
        - Lock released quickly -> **Spin-wait**
        - Lock released slowly -> **Block**
        - Quick and slow are relative to context-switch cost, **C**