Lecture 10: Lock Implementation
Instructor: Chengyu Song
Slide contributions from
Nael Abu-Ghazaleh, Harsha Madhyvasta and Zhiyun Qian
Recap: Synchronization

- **Race Condition** can happen between threads with shared data
  - Concurrent
  - At least one write

- Synchronization is necessary to provide
  - Safety, liveness, performance

- **Mechanisms**
  - Building blocks: atomic read and write
  - Alternation, lock
Locks

- A lock is an object in memory providing two operations
  - `acquire()`: before entering the critical section
  - `release()`: after leaving a critical section

- Threads **pair calls** to `acquire()` and `release()`
  - Between `acquire()`/`release()`, the thread **holds** the lock
  - `acquire()`: does not return until any previous holder releases
  - What can happen if the calls are not paired?
Using Locks

withdraw (account, amount) {
    acquire(lock);
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    release(lock);
    return balance;
}

- Why is the “return” outside the critical section? Is this ok?
- What happens when a third thread calls acquire?
First try

```c
pthread_trylock(mutex) {
    if (mutex==0) {
        mutex= 1;
        return 1;
    } else return 0;
}
```

Thread 0, 1, ...

```c
...//time to access critical region
while(!pthread_trylock(mutex); // wait
<critical region>
pthread_unlock(mutex)
```

- Does this work? Assume reads/writes are atomic
- The lock itself is a critical region!
  - Chicken and egg
- Computer scientist struggled with how to create software locks
Second try

```c
int turn = 1;

while (true) {
    while (turn != 1) ;
    critical section
    turn = 2;
    outside of critical section
}

while (true) {
    while (turn != 2) ;
    critical section
    turn = 1;
    outside of critical section
}
```

This is called **alternation**

It satisfies mutex:

- If blue is in the critical section, then turn == 1 and if yellow is in the critical section then turn == 2
- (turn == 1) ≡ (turn != 2)

**Is there anything wrong with this solution?**
Third try – two variables

We added two variables to try to break the race for the same variable

Is there anything wrong with this solution?
Fourth try – set before you check

Is there anything wrong with this solution?
Fifth try – double check & back off

flag[0] = 1;
while (flag[1] != 0) {
    flag[0] = 0;
    wait a short time;
    flag[0] = 1;
}
critical section
flag[0]=0;
outside of critical section

flag[1] = 1;
while (flag[0] != 0) {
    flag[1] = 0;
    wait a short time;
    flag[1] = 1;
}
critical section
flag[1]=0;
outside of critical section
Six try – Dekker’s Algorithm

```
Bool flag[2];
Int turn = 1;
flag[0] = 1;
while (flag[1] != 0) {
    if(turn == 2) {
        flag[0] = 0;
        while (turn == 2);
        flag[0] = 1;
    } //if
} //while

critical section
flag[0] = 0;
turn = 2;
outside of critical section

flag[1] = 1;
while (flag[0] != 0) {
    if(turn == 1) {
        flag[1] = 0;
        while (turn == 1);
        flag[1] = 1;
    } //if
} //while

critical section
flag[1] = 0;
turn = 1;
outside of critical section
```
Peterson's Algorithm

```
int turn = 1;
bool try1 = false, try2 = false;
```

```
while (true) {
    try1 = true;
    turn = 2;
    while (try2 && turn != 1) ;
    critical section
    try1 = false;
    outside of critical section
}
```

```
while (true) {
    try2 = true;
    turn = 1;
    while (try1 && turn != 2) ;
    critical section
    try2 = false;
    outside of critical section
}
```

• This satisfies all the requirements
• Here's why...
Peterson's Algorithm: analysis

```plaintext
int turn = 1;
bool try1 = false, try2 = false;
```

while (true) {
    { ¬ try1 ∧ (turn == 1 v turn == 2) }
1 try1 = true;
    { try1 ∧ (turn == 1 v turn == 2) }
2 turn = 2;
    { try1 ∧ (turn == 1 v turn == 2) }
3 while (try2 && turn != 1) ;
    { try1 ∧ (turn == 1 v ¬ try2 v
        (try2 ∧ (yellow at 6 or at 7)) )
        critical section
4 try1 = false;
    {¬ try1 ∧ (turn == 1 v turn == 2) }
outside of critical section
}

while (true) {
    { ¬ try2 ∧ (turn == 1 v turn == 2) }
5 try2 = true;
    { try2 ∧ (turn == 1 v turn == 2) }
6 turn = 1;
    { try2 ∧ (turn == 1 v turn == 2) }
7 while (try1 && turn != 2) ;
    { try2 ∧ (turn == 2 v ¬ try1 v
        (try1 ∧ (blue at 2 or at 3)) )
        critical section
8 try2 = false;
    {¬ try2 ∧ (turn == 1 v turn == 2) }
outside of critical section
}
```

(blue at 4) ∧ try1 ∧ (turn == 1 v ¬ try2 v (try2 ∧ (yellow at 6 or at 7))
    ∧ (yellow at 8) ∧ try2 ∧ (turn == 2 v ¬ try1 v (try1 ∧ (blue at 2 or at 3))
... ⇒ (turn == 1 ∧ turn == 2)
Synchronization so far...

- We looked at how to build software locks
  - Difficult
  - Worse: it doesn’t really work
    - Compilers don’t think multi-threaded
    - Hardware reorders memory ops: memory consistency models
- Lets get help from the hardware!

```
while (true) {
  try1 = true;
  turn = 2;
  while (try2 && turn != 1) ;
  critical section
  try1 = false;
  outside of critical section
}
```

Compiler transforms to
```
try1 = false;
turn = 2;
while (true) {
  while (try2 && turn != 1) ;
  critical section
  outside of critical section
}
```
Hardware to the rescue

- Crux of the problem:
  - We get interrupted between checking the lock and setting it to 1
  - Software locks reordered by compiler/hardware

- Possible solutions?
  - Atomic instructions: create a new assembly language instruction that checks and sets a variable atomically
    » Cannot be interrupted!
    » How do we use them?
  - Disable interrupts altogether (no one else can interrupt us)
Atomic Instruction: Test-and-Set

- The semantics of test-and-set are:
  - Record the old value
  - Set the value to indicate available
  - Return the old value

- Hardware executes it atomically!

```c
bool test_and_set (bool *flag) {
    bool old = *flag;
    *flag = True;
    return old;
}
```

- When executing test-and-set on “flag”
  - What is value of flag afterwards if it was initially False? True?
  - What is the return result if flag was initially False? True?
Using Test-and-Set

- Here is our lock implementation with test-and-set:

```c
struct lock {
    int held = 0;
};

void acquire (lock) {
    while (test-and-set(&lock->held));
}

void release (lock) {
    lock->held = 0;
}
```

- When will the while return? What is the value of held?
- Does it satisfy critical region requirements? (mutex, progress, bounded wait, performance?)
Still a Spinlocks

- The problem with spinlocks is that they are wasteful
  - Although still useful in some cases; let's discuss advantages and disadvantages

- If a thread is spinning on a lock, then the scheduler thinks that this thread needs CPU and puts it on the ready queue

- If N threads are contending for the lock, the thread which holds the lock gets only 1/N’th of the CPU
Disabling Interrupts

- Another implementation of acquire/release is to disable interrupts:

```c
struct lock {
}
void acquire (lock) {
    disable interrupts;
}
void release (lock) {
    enable interrupts;
}
```

- Note that there is no state associated with the lock
- Can two threads disable interrupts simultaneously?
On Disabling Interrupts

- Disabling interrupts blocks notification of external events that could trigger a context switch (e.g., timer)
- In a “real” system, this is only available to the kernel
  - Why?

- Disabling interrupts is insufficient on a multiprocessor
  - Back to atomic instructions
- Like spinlocks, only want to disable interrupts to implement higher-level synchronization primitives
  - Don’t want interrupts disabled between acquire and release
Summarize Where We Are

- Goal: Use **mutual exclusion** to protect **critical sections** of code that access **shared resources**
- Method: Use locks (spinlocks or disable interrupts)
- Problem: Critical sections can be long

**Spinlocks:**
- Threads waiting to acquire lock spin in test-and-set loop
- Wastes CPU cycles
- Longer the CS, the longer the spin
- Greater the chance for lock holder to be interrupted
- Memory consistency model causes problems (out of scope of this class)

```plaintext
acquire(lock)
...
Critical section
...
release(lock)
```

**Disabling Interrupts:**
- Should not disable interrupts for long periods of time
- Can miss or delay important events (e.g., timer, I/O)
Higher-Level Synchronization

- Spinlocks and disabling interrupts are useful for short and simple critical sections
  - Can be wasteful otherwise
  - These primitives are “primitive” – don’t do anything besides mutual exclusion

- Need higher-level synchronization primitives that:
  - Block waiters
  - Leave interrupts enabled within the critical section

- All synchronization requires atomicity

- So we’ll use our atomic locks as primitives to implement them
Implementing Locks: Block Waiters

- Block waiters, interrupts enabled in critical sections

```c
struct lock {
    int held = 0;
    queue Q;
}

void acquire (lock) {
    Disable interrupts;
    if (lock->held) {
        put current thread on lock Q;
        block current thread;
    }
    lock->held = 1;
    Enable interrupts;
}

void release (lock) {
    Disable interrupts;
    if (Q)
        remove and unblock a waiting thread;
    else
        lock->held = 0;
    Enable interrupts;
}
```

---

`acquire(lock)`

- Interrupts Disabled
- Critical section

`release(lock)`

- Interrupts Disabled
- Interrupts Enabled
- Interrupts Disabled
Implementing Locks: Yield on Wait

- Instead of spinning, yield the CPU

```c
struct lock {
    int held = 0;
}

void acquire (lock) {
    while (test-and-set(&lock->held)) {
        thread_yield();
    }
}

void release (lock) {
    lock->held = 0;
}
```
Advanced Topics

- Test-and-set with back-off (exponential works best)
- Ticket lock
- Wait queue
  - Liveness requirement
  - Performance requirement
  - Array-based queue
  - List-based queue
    » MCS, JM Mellor-Crummey and ML Scott (2006 Edsger Dijkstra Prize)
Next class

- Semaphores
- Preparation
  - Read Module 31
  - Little Book on Semaphores