Locks

Prof. Yongtae Kim

Computer Science and Engineering
Kyungpook National University

Locks: The Basic Idea & Pthread Locks

lock의 내부 메커니즘

- **To use a lock, some code around the critical section is required**

```
lock_t mutex; // some globally-allocated lock 'mutex'
...
lock(&mutex);
balance = balance + 1;
unlock(&mutex);
```

critical section

- **A lock is a variable that holds state of lock at any instant in time**
  - The state is either 1) available (or unlocked or free) that no thread holds the lock or 2) acquired (or locked or held) that exactly one thread holds the lock
- **Calling the routine `lock()` tries to acquire the lock**
  - If no other thread hold the lock, the thread will acquire it and become the owner
  - If another thread calls **`locks()`** on the same lock variable, it will not return and thus other threads are prevented from entering the critical section
  - Once the owner calls **`unlock()`**, the lock is available again

스레드 자원 공유
→ lock을 통해 원자적으로
수행 → 데이터 무결성 보장.

- **The name that the POSIX library uses for a lock is a mutex, as it is used to provide mutual exclusion between threads**

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Pthread_mutex_lock(&lock); // wrapper; exits on failure
balance = balance + 1;
Pthread_mutex_unlock(&lock);
```

→ 풀릴때 까지 기다림.
spin 같은 것이다.

Building and Evaluating a Lock

- **To build a working lock, hardware and OS need collaboration**
  - Over the years, a number of different hardware primitives have been added to the instruction sets of various computer architectures

    명령어 셋.
  - OS gets involved to complete building a sophisticated locking library

    OS의 도움을 받아야 함

- **To evaluate whether a lock works (and works well), some basic criteria need to be established**

  3가지를 고려해야 함

  상호배제.

  1) Mutual exclusion: does the lock work, preventing multiple threads from entering a critical section?

  공정성 기아 상태가 있어서는 안된다.

  2) Fairness: does each thread contending for the lock get a fair shot at acquiring it once it is free? (lock starvation)

  성능 시간이라는 자원의 overhead.

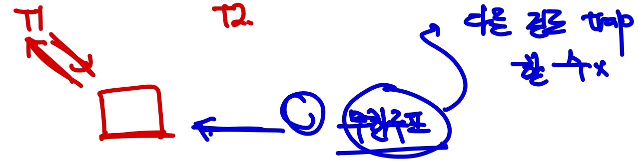
  3) Performance: the time overheads added by using the lock; considering three cases of no contention, multiple threads on a single CPU or multiple CPUs

  ① thread가 1개일 때 lock. → 오히려 lock/unlock 과정이 불필요

  ② CPU가 1개일 때 여러 thread가 경쟁하는 상태

  ③ CPU가 여러개일 때 여러 thread가 lock을 취하기 위해 경쟁하는 상태

Controlling Interrupts



- **One of the earliest solution to provide mutual exclusion was to disable interrupts for critical sections**

  쉬운 방법은 critical section에서 interrupt를 끄면 됨.

  – The code inside the critical section (will not be interrupted), thus will execute as if it's atomic; its simplicity is the main benefit but quite slow due to turning it on/off

  효율성 — 느림

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

  → OS가 모든 명령을 내려 줘야 함.

  신뢰가 중요

  – Since disabling interrupts requires a privileged operation, we need trust that this facility is not abused (e.g. infinite loop can monopolize the processor)

  → timer interrupt x.

  – This does not work on multiprocessors; each of multiple threads on different CPUs tries to enter the same critical section, disabling on a CPU doesn't matter

  멀티 프로세스 환경 문제 x.
  있어서 해결.

  – Turning off interrupts for long time can lead to interrupt loss

  cpu disk I/O interrupt 무시 x.

- **For these reason, turning off interrupts is only used in limited context as a mutual-exclusion primitive**

  → OS 자체 진행에서 lock, unlock은 사용할 때에나 사용할 수 있음

  – e.g.) OS itself will use interrupt masking to guarantee atomicity when accessing its own data structure, or at least to prevent certain messy interrupt handling

A Failed Attempt: Just Using Loads/Stores

→하드웨어의 지원은 받음.

- Let's try to build a simple lock by using a **single flag variable**

```
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;
}
```

flag

원자성이 보장 되지 않음. ⟸

Assuming flag = 0 to begin

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!)	

Thread2에서 lock 획득

context-switch할 때 진행시점 저장

둘 다 lock을 가질 수 있음

  - Using a simple variable (flag) to indicate if any thread has the lock
  - **lock()** to test whether flag = 0/1 and set the flag to 1 to acquire lock if flag = 0

    flag를 1로 설정하고 lock → flag 0으로 설정
  - **unlock()** to clear the flag that indicates that the lock is no longer held
- This software-only approach has two problems
  1) Correctness: multiple threads are able to enter the critical section (right figure)
  2) Performance: checking the flag continuously while flag=1; spin-waiting

     test-and-set | atomic exchange.

     cpu 사이클을 계속 아무 의미 없이 (소모중)
- To resolve issues, we need **hardware support** to implement lock

Building Working Spin Locks with Test-And-Set

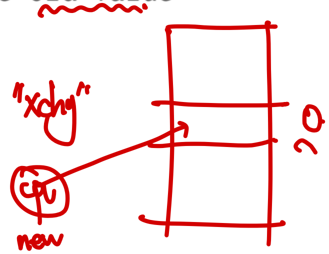
- **The simplest hardware support is test-and-set instruction**
  - It returns the old value pointed by **`old_ptr`**, and simultaneously updates the value **`new`**; the key is that this sequence is performed **ATOMICALLY**!

It is sometimes called **atomic exchange**
memory with register

→ 하나의 명령으로 수행

```
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
}
```

x86: **`xchg`**
RISC-V: **`amoswap`**
SPARC: **`ldstub`**




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

void init(lock_t *lock) {
    // 0: lock is available, 1: lock is held
    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;
}
```

T1 T2 preemtive 선점할 수 있다.

flag = 0

[illegible] T1. lock. flag = 0 T2 spin

  - flag = 0: **`TestAndSet()`** tests the flag and sets it to 1; return 0 and acquire lock
  - flag = 1: **`TestAndSet()`** tests the flag and sets it to 1 again; return 1 and spinning until the lock is released → when flag=0, then can acquire lock

↳ 이전값의 (old value)가 0이었으면 끝.

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

→ 혼자 계속 spin-wait하게 두면 안됨.
∴ 스케줄링을 해줘서 독점을 하지 못하도록 막아야 함

  - Without preemption, a thread spinning on a CPU will never relinquish it

Evaluating Spin Locks

정확성

- **The most important aspect of a lock is correctness:**
  - Does the spin lock provide mutual exclusion? Yes, we have a correct lock

    배타성 보장 TestAndSet:

- **The next aspect is fairness:**
  - How fair is a spin lock to a waiting thread?

    spin-lock은 공정성을 보장하지 x.
  - In other words, can it guarantee that a waiting thread will ever enter the critical section? Unfortunately, not fair; this may lead to starvation

t0 ~ t9 RR 20ms

lock. c t1~t9

- **The final aspect is performance:**

  spin-wait 20ms를 해야함.
  - In single CPU: the performance overheads can be quite painful; one of a lot of threads is holding the lock and the others keep spinning, a waste of CPU cycle
    e.g.) T0~T9 try to acquire a lock, T0 has the lock for 30ms, time slice: 20ms
    Single CPU: T0 runs for 20ms (lock) → T1~T9 spin one by one for 20ms
    → T1 releases the lock after 180ms → one of T1~T9 can obtain the lock

    → 바로 끝나자 마자 잡을 수 있다.
  - In multiple CPUs: spin locks work reasonably well (# of threads ≈ # of CPUs);
    e.g.) T0~T9 on CPU0~9 try to get a lock and T0 acquires the lock for 30 ms
    T0 runs and T1~T9 spins simultaneously for 30ms → one of T1~T9 will get the lock after 30ms (right after T1 releases the lock)

Compare-And-Swap

- **Another hardware primitive is compare-and-swap or compare-and-exchange (x86) instruction**
  - It tests whether the value at the address specified by `ptr` is equal to `expected`
  - If so, update the memory pointed by `ptr` with `new`, otherwise, do nothing
  - In either case, it returns the original value at that memory location

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int original = *ptr;
    if (original == expected)
        *ptr = new;
    return original;
}
```

→ 비교가 맞을 때만 바꿈.

state-condition

x86: **cmpxchg**

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

ptr

unlock 부분이라면 lock해제.

  - `CompareAndSwap()` simply checks if flag = 0 and if so, it atomically swaps in a 1 thus acquiring the lock

기본 메커니즘 자체는 동일.

  - Actually, compare-and-swap is more powerful instruction than test-and-set in lock-free synchronization, but their behavior is identical in terms of spin-lock
- **In x86, `cmpxchg` is the instruction for compare-and-swap**
  - If you want to see how to make a C-callable x86-version of compare-and-swap: https://github.com/remzi-arpacidusseau/ostep-code/tree/master/threads-locks

Load-Linked and Store-Conditional

- **The load-linked and store-conditional instruction pair can be used to build locks and other concurrent structures**
  - The load-linked instruction simply fetches a value from memory to a register
  - With no intervening store to the address, the store-conditional instruction returns 1 with updating the value at **`ptr`** to **`value`**, otherwise, it returns 0 with no update

RISC-V, MIPS, Alpha, PowerPC, ARM support these instructions

ptr.

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

→ load 명령어

lock T1 T2 T3

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

로드에 대한 값이 변경되지 않았을 때만.

0일 때 까지 spin.

```
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
    }
}
```

만약 ①lock→flag를 통해 멀티 쓰레드에서 동시에 접근하더라도 정상작동

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

concise form

  - **`LoadLinked()`** spins waiting for the flag to be set to 0 (lock is not held)
  - **`StoreConditional()`** tries to acquire lock and atomically changes the flag to 1
  - If two threads call **`LoadLinked()`** and both get 0 (lock is free), one of them will get the lock with **`StoreConditional()`** and the other will fail since the winning thread updated the flag between its **`LoadLinked()`** and **`StoreConditional()`**

Fetch-And-Add

turn을 도입해서. / 모든 thread가 언젠가는 lock을 얻을 수 있게 되는걸.

- One final hardware primitive is **fetch-and-add instruction**
  - It atomically increments a value while returning the old value, at a particular address and can be used to build a ticket lock.

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

x86: **xadd**

```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn   = 0;
}

void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
        ; // spin
}

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

ticket … 10

turn

$T_0 = 0$ $T_1 = 1$ $T_2 = 2$ ⋮ $T_9 = 9$

turn이 0일때 가능

fairness가 보장된다.

  - `FetchAndAdd()` increases the ticket value that is considered the thread's turn
  - The globally shared `lock->turn` is used to determined the next turn
  - Unlock is done by incrementing the turn so next thread can have the lock if any
- This approach **ensures progress for all threads**

fairness

  - Once a thread has its ticket value, it will be scheduled at some point in future

Too Much Spinning

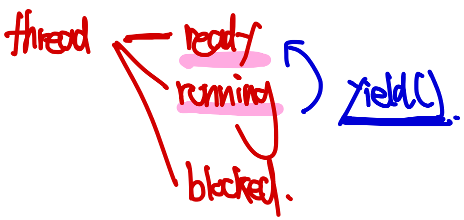
→ lock을 가지고 있지 않은 thread는 spin으로 자신 차례를 낭비함! → OS 개입해!

- **The hardware-based locks are simple and works well**
  - However, in some cases, these solutions can be quite inefficient
  - Threads not holding the lock keep spinning, wasting an entire time slice
  - To solve this issue, OS support is necessary

spin 하는 동안에는, 다른 스레드에게 CPU를 양보하는 것.

- **A simple approach to avoid spin is to yield**
  - OS primitive **`yield()`** is required to give up the CPU and let another thread run

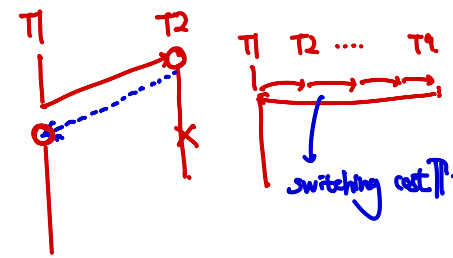
개념




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

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

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



sleep mode로 가면 다른 프로세스가 일을 할 수 있음.

  - Two threads: a thread has a lock and the other yields the CPU, then thread with the lock will run and finish faster its critical section; this works well

문제 해결X

  - Many threads: a thread holds a lock, each of the others execute this run-and-yield before getting the lock; this still has cost (context switch↑)

(이 문제 해결X) ← 문제 해결X "overhead ↑

  - Worse, this approach does not resolve the starvation problem at all

Using Queues: Sleeping instead of Spinning

→ 먼저 lock을 가진 스레드가 unlock한 후에 lock을 얻은 스레드를 직접해 준다면

- **To provide fairness, we need some explicit control for threads**
  - To do this, we will need queue to keep track of which threads are waiting for lock
  - Two calls: **park()** to sleep, **unpark(threadID)** to wake a thread with **threadID**

↳ 현재 스레드 잠재. ↳ threadID로 특정 스레드를 깨우는 함수

test-and-set + starvation 문제 해결.

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

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();
    }
}

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 thread!)
    m->guard = 0;
}
```

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

setpark() is added to resolve wakeup/waiting race

spin-wait 하는 시간이 제한됨.

= guard가 0이 될 때까지 spin

→ guard가 0이고 flag가 0이면 lock을 얻어오는 부분.

→ 대기하는 thread에게.

- The guard is used as spin lock around flag and queue; Acquiring the guard lock is first step and this lock is released within a few instructions → spin overhead↓
- When a thread can't acquire the lock, it is added to a queue, sleeps with guard=0
- What if T1 released the lock just before **park()** in T2? → T2 parks and sleeps forever (wakeup/waiting race) as no thread will call **unlock()** ➔ **setpark()** indicates it is about to park → **park()** returns immediately instead of sleeping

Different OS, Different Support

(운영체제에 따라 lock을 제공하는 scheme이 다름.)

- **OSes provide their own lock scheme in the thread library**
  - Linux provides a futex which provides some in-kernel functionality
  - Each futex has associated with it a specific physical memory location and a per-futex in-kernel queue
  - Futex offers two calls: **`futext_wait()`** and **`futext_wake()`**
  - Please see the Linux-based futex lock code at pp. 349

guard.

- **Two-phase lock realizes that spinning can be useful, particularly if the lock is about to be released**
  - 1st phase: spinning for a while, hoping that it can acquire the lock
  - 2nd phase: if the lock is not acquired in 1st spin phase, the caller sleeps when 2nd phase is entered and only woken up when the lock becomes free later

second phase에 들어가면 sleeping

계속 spin이 더 효율적인 수 있다.

flag