

# Locks

**Operating System (OS)**  
**안인규**

강의자료는 KAIST 원유집 교수님 OS 강의 자료를 참고했습니다.

# Locks: The Basic Idea

- Ensure that any **critical section** executes as if it were a single **atomic instruction**.
  - An example: the canonical update of a shared variable

```
balance = balance + 1;
```

- Add some code around the critical section

```
1  lock_t mutex; // some globally-allocated lock 'mutex'  
2  ...  
3  lock(&mutex);  
4  balance = balance + 1;  
5  unlock(&mutex);
```

# Locks: The Basic Idea

- Lock variable holds the state of the lock.
  - **available** (or **unlocked** or **free**)
    - No thread holds the lock (현재 lock 을 사용하는 thread 없음)
  - **Acquired** (or **locked** or **held**)
    - Exactly one thread holds the lock and presumably is in a critical section (누군가 lock을 잡고 있고, 다른 thread는 대기 중)

# The semantics of the lock()

- lock ()
  - **Try to** acquire the lock.
  - If no other thread holds the lock, the thread will **acquire** the lock.
  - **Enter** the *critical section*.
    - This thread is said to be the owner of the lock.
- Other threads are *prevented from* entering the critical section while the first thread that holds the lock is in there.

# Pthread Locks - mutex

- The name that the POSIX library uses for a lock.
  - Used to provide **mutual exclusion** between threads.

```
1  pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
2
3  Pthread_mutex_lock(&lock); // wrapper for pthread_mutex_lock()
4  balance = balance + 1;
5  Pthread_mutex_unlock(&lock);
```

- We may be using *different locks* to protect *different variables* → Increase **concurrency** (a more **fine-grained** approach).

# Building A Lock

- Efficient locks provided mutual exclusion at **low cost**
- Building a lock need some help from the **hardware** and the **OS**

# Evaluating locks – Basic criteria

- **Mutual exclusion**

- Does the lock work, preventing multiple threads from entering *a critical section*? (즉, 단 하나의 thread만 critical section에 들어가야 함)

- **Fairness**

- Does each thread contending for the lock get a fair shot at acquiring it once it is free? (Starvation) (즉, 특정 thread가 lock을 계속 획득하지 못하는 경우가 없어야 함)

- **Performance**

- The time overheads added by using the lock (lock을 잡고 푸는데 드는 비용이 작아야 함)

# Controlling Interrupts

- **Disable Interrupts** for critical sections
  - One of the earliest solutions used to provide mutual exclusion
  - Invented for single-processor systems.

```
1  void lock() {  
2      DisableInterrupts();  
3  }  
4  void unlock() {  
5      EnableInterrupts();  
6  }
```

- Problem:
  - Require too much *trust* in applications
    - Greedy (or malicious) program could monopolize (독점하다) the processor.
  - Do not work on **multiprocessors (다중 CPU)**
  - Turning off interrupts for extended periods of time **can lead to interrupts becoming lost** (can lead to serious systems problems)

# Why hardware support needed?

- **First attempt:** Using a *flag* denoting whether the lock is held or not.
  - The code below has problems (**no atomicity (원자성)**)

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

# Why hardware support needed? (Cont.)

- **Problem 1:** No Mutual Exclusion (assume `flag=0` to begin)

Thread1	Thread2
<pre>call lock() while (flag == 1) interrupt: switch to Thread 2</pre>	<pre>call lock() while (flag == 1) flag = 1; interrupt: switch to Thread 1</pre>
<pre>flag = 1; // set flag to 1 (too!)</pre>	

- **Problem 2:** Spin-waiting wastes time waiting for another thread.
- So, we need an atomic instruction supported by **Hardware!**
  - *test-and-set* instruction, also known as *atomic exchange*

# Test And Set (Atomic Exchange)

- An 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  }
```

- **return**(testing) old value pointed to by the `ptr`.
- *Simultaneously* **update**(setting) said value to `new`.
- This sequence of operations is **performed atomically**.

# A Simple Spin Lock using test-and-set

```
1  typedef struct __lock_t {
2      int flag;
3  } lock_t;
4
5  void init(lock_t *lock) {
6      // 0 indicates that lock is available,
7      // 1 that it is held
8      lock->flag = 0;
9  }
10
11 void lock(lock_t *lock) {
12     while (TestAndSet(&lock->flag, 1) == 1)
13         ; // spin-wait
14 }
15
16 void unlock(lock_t *lock) {
17     lock->flag = 0;
18 }
```

- **Note:** To work correctly on *a single processor*, it requires a preemptive scheduler.

# Evaluating Spin Locks

- **Correctness:** yes
  - The spin lock only allows a single thread to entry the critical section.
- **Fairness:** no
  - Spin locks don't provide any fairness guarantees.
  - Indeed, a thread spinning may spin *forever* (오래 기다린 thread 가 우선권을 갖는 보장이 없음)
- **Performance:**
  - In the single CPU, performance overheads can be quire *painful* (lock 을 얻기 위해 spin하며 CPU를 낭비 → 다른 lock을 잡은 thread 는 실행 기회를 얻지 못함)
  - If the number of threads roughly equals the number of CPUs, spin locks work *reasonably well* (multi core에서는 괜찮은 선택지일 수도)

# Compare-And-Swap (SPARC)

- 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  }
```

**Compare-and-Swap hardware atomic instruction (C-style)**

```
1  void lock(lock_t *lock) {
2      while (CompareAndSwap (&lock->flag, 0, 1) == 1)
3          ; // spin
4  }
```

**Spin lock with compare-and-swap**

# Compare-And-Exchange (x86)

- C-callable x86-version of compare-and-swap

```
1  char CompareAndSwap(int *ptr, int old, int new) {
2      unsigned char ret;
3
4      // Note that sete sets a 'byte' not the word
5      __asm__ __volatile__ (
6          " lock\n"           // Hardware Support (메모리 버스 잠금)
7          " cpxchgl %2,%1\n"
8          " sete %0\n"
9          : "=q" (ret), "=m" (*ptr)
10         : "r" (new), "m" (*ptr), "a" (old)
11         : "memory");
12     return ret;
13 }
```

# Load-Linked and Store-Conditional (MIPS)

```
1  int LoadLinked(int *ptr) {
2      return *ptr;
3  }
4
5  int StoreConditional(int *ptr, int value) {
6      if (no one has updated *ptr since the LoadLinked to this address) {
7          *ptr = value;
8          return 1; // success!
9      } else {
10         return 0; // failed to update
11     }
12 }
```

- 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 updates and 0 is returned.

# Load-Linked and Store-Conditional (Cont.)

```
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
11 void unlock(lock_t *lock) {
12     lock->flag = 0;
13 }
```

**Using LL/SC To Build A Lock**

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

**A more concise form of the lock() using LL/SC**

# Evaluating Spin Locks

- **Correctness:** yes
  - The spin lock only allows a single thread to entry the critical section ([atomicity](#))
- **Fairness:** no
  - Spin locks don't provide any fairness guarantees.
  - Indeed, a thread spinning may spin *forever* (오래 기다린 thread 가 우선권을 갖는 보장이 없음)
- **Performance:**
  - In the single CPU, performance overheads can be quire *painful* (lock 을 얻기 위해 spin하며 CPU를 낭비 → 다른 lock을 잡은 thread 는 실행 기회를 얻지 못함)
  - If the number of threads roughly equals the number of CPUs, spin locks work *reasonably well* ([multi core에서는 괜찮은 선택지일 수도](#))

# Fetch-And-Add

- **Atomically increment** a value while returning the old value at a particular address.

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

**Fetch-And-Add Hardware atomic instruction (C-style)**

# Ticket Lock

- **Ticket lock** can be built with fetch-and add.
  - Ensure progress for all threads. → **fairness**

```
1  typedef struct __lock_t {
2      int ticket;
3      int turn;
4  } lock_t;
5
6  void lock_init(lock_t *lock) {
7      lock->ticket = 0;
8      lock->turn = 0;
9  }
10
11 void lock(lock_t *lock) {
12     int myturn = FetchAndAdd(&lock->ticket);
13     while (lock->turn != myturn)
14         ; // spin
15 }
16 void unlock(lock_t *lock) {
17     FetchAndAdd(&lock->turn);
18 }
```

# So Much Spinning

- Hardware-based spin locks are **simple** and they work.
- In some cases, these solutions can be quite **inefficient**.
  - Any time a thread gets caught *spinning*, it **wastes an entire time slice** doing nothing but checking a value.
  - **Blocked state (sleep)!**

How To Avoid *Spinning*?  
We'll need **OS Support** too!

# A Simple Approach: Just Yield

- When you are going to spin, **give up the CPU** to another thread.
  - OS system call moves the caller from the *running state* to the *ready state*.
  - The cost of a **context switch** can be substantial and the **starvation** problem still exists.

```
1  void init() {
2      flag = 0;
3  }
4
5  void lock() {
6      while (TestAndSet(&flag, 1) == 1)
7          yield(); // give up the CPU
8  }
9
10 void unlock() {
11     flag = 0;
12 }
```

# Using Queues: Sleeping Instead of Spinning

- **Queue** to keep track of which threads are waiting to enter the lock.
- `park()`
  - Put a calling thread to sleep
- `unpark(threadID)`
  - Wake a particular thread as designated by `threadID`.

# Using Queues: Sleeping Instead of Spinning

```
typedef struct __lock_t {  
    int flag;           // lock is acquired or not  
    int guard;         // to protect the queue  
    queue_t *q;  
} lock_t;
```

# Using Queues: Sleeping Instead of Spinning

```
1  typedef struct __lock_t { int flag; int guard; queue_t *q; } lock_t;
2
3  void lock_init(lock_t *m) {
4      m->flag = 0;          // 0: unlock, 1: lock
5      m->guard = 0;        // guard lock, 0: unlock, 1: lock
6      queue_init(m->q);
7  }
8
9  void lock(lock_t *m) {
10     while (TestAndSet(&m->guard, 1) == 1)
11         ; // acquire guard lock by spinning
12     if (m->flag == 0) {
13         m->flag = 1; // lock is acquired
14         m->guard = 0;
15     } else {
16         queue_add(m->q, gettid());
17         m->guard = 0;
18         park();          // Put a calling thread to sleep
19     }
20 }
21 ...
```

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

# Using Queues: Sleeping Instead of Spinning

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

**Lock With Queues, Test-and-set, Yield, And Wakeup (Cont.)**

# Wakeup/waiting race

- In case of releasing the lock (*thread A*) just before the call to `park()` (*thread B*) → Thread B would **sleep forever** (potentially) (즉, `park()`와 `unpark()` 사이의 타이밍이 꼬이면...)
- **Solaris (OS)** solves this problem by adding a third system call: `setpark()`.
  - By calling this routine, a thread can indicate 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.

```
1     queue_add(m->q, gettid());
2     setpark(); // new code
3     m->guard = 0;
4     park();
```

Code modification inside of `lock()`

# Futex

- Linux provides a **futex** (is similar to Solaris's `park` and `unpark`).
  - `futex_wait(address, expected)`
    - Put the calling thread to sleep
    - If the value at `address` is not equal to `expected`, the call returns immediately.
  - `futex_wake(address)`
    - Wake one thread that is waiting on the queue.

# Futex (Cont.)

- Snippet from `lowlevellock.h` in the **nptl** library
  - The high bit of the integer `v`: track whether the lock is held or not
  - All the other bits : the number of waiters

```
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; // 락 획득 성공
6      atomic_increment(mutex);
7      while (1) {
8          if (atomic_bit_test_set(mutex, 31) == 0) {
9              atomic_decrement(mutex);
10             return; // 락 획득 성공
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         ...
```

Linux-based Futex Locks

# Futex (Cont.)

```
16         if (v >= 0)
17             continue; // Unlock 상태일 수 있다
18         futex_wait(mutex, v);
19     }
20 }
21
22 void mutex_unlock(int *mutex) {
23     /* Adding 0x80000000 to the counter results in 0 if and only if
24        there are not other interested threads */
25     if (atomic_add_zero(mutex, 0x80000000))
26         return;
27     /* There are other threads waiting for this mutex,
28        wake one of them up */
29     futex_wake(mutex);
30 }
```

Linux-based Futex Locks (Cont.)

# 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,
    - 짧은 대기 상황에는 Spin이 빠름 (multi-CPU 환경에서)
  - **Second phase**
    - The caller is put to sleep.
    - The caller is only woken up when the lock becomes free later.
    - 긴 대기 상황에는 sleep (blocked)이 효과적

# Summary

- Hardware support
  - Test-and-set
  - Compare-and-swap (SPARC, x86)
  - Load-linked and store conditional (ll/sc, MIPS)
- OS support
  - park()/unpark()/setpark()
  - futex
  - Two-phase locks