Operating System: Locks

Sang Ho Choi (shchoi@kw.ac.kr)
School of Computer & Information Engineering
KwangWoon University

The Classic Example

100만원
공유

- Withdrawing money from a bank account
  - Suppose you and your girl (or boy) friend share a bank account with a balance of 1,000,000won
  - What happens if both go to separate ATM machines and simultaneously withdraw 100,000won from the account?

10만원 동시인출

```
int withdraw (account, amount)
{
    balance = get_balance (account);
    balance = balance - amount;
    put_balance (account, balance);
    return balance;
}
```

The Classic Example (Cont'd)

- The execution of the two threads can be interleaved, assuming preemptive scheduling:

boy

Execution sequence as seen by CPU

```
balance = get_balance (account);
balance = balance - amount;
```

Context switch

중간에 끼어듦

```
balance = get_balance (account);
balance = balance - amount;
put_balance (account, balance);
```

Context switch

```
put_balance (account, balance);
```

girl

The Real Example

```
extern int g;
void inc()
{
    g++;
}
```

```
movl 0x1000, %eax
addl $1, %eax
movl %eax, 0x1000
```

Thread T1		Thread T2
`movl 0x1000, %eax`		
`addl $1, %eax`		
	Context switch	
		`movl 0x1000, %eax`
		`addl $1, %eax`
		`movl %eax, 0x1000`
	Context switch	
`movl %eax, 0x1000`		

Sharing Resources

- Local variables are not shared among threads
  - Refer to data on the stack
  - Each thread has its own stack
  - Never pass/share/store a pointer to a local variable on another thread's stack

thread마다 고유한 stack을 가지니 공유X

포인터로도 공유안됨

- Global variables are shared among threads
  - Stored in data segment, accessible by any thread

전역은 가능

- Dynamic objects are shared among threads
  - Stored in the heap, shared through the pointers

동적할당도 가능.

- Also, processes can share memory (shmem)

process 간에는 공유메모리 공유

Synchronization Problem

동기화문제가 발생하면 결과를 예측X.

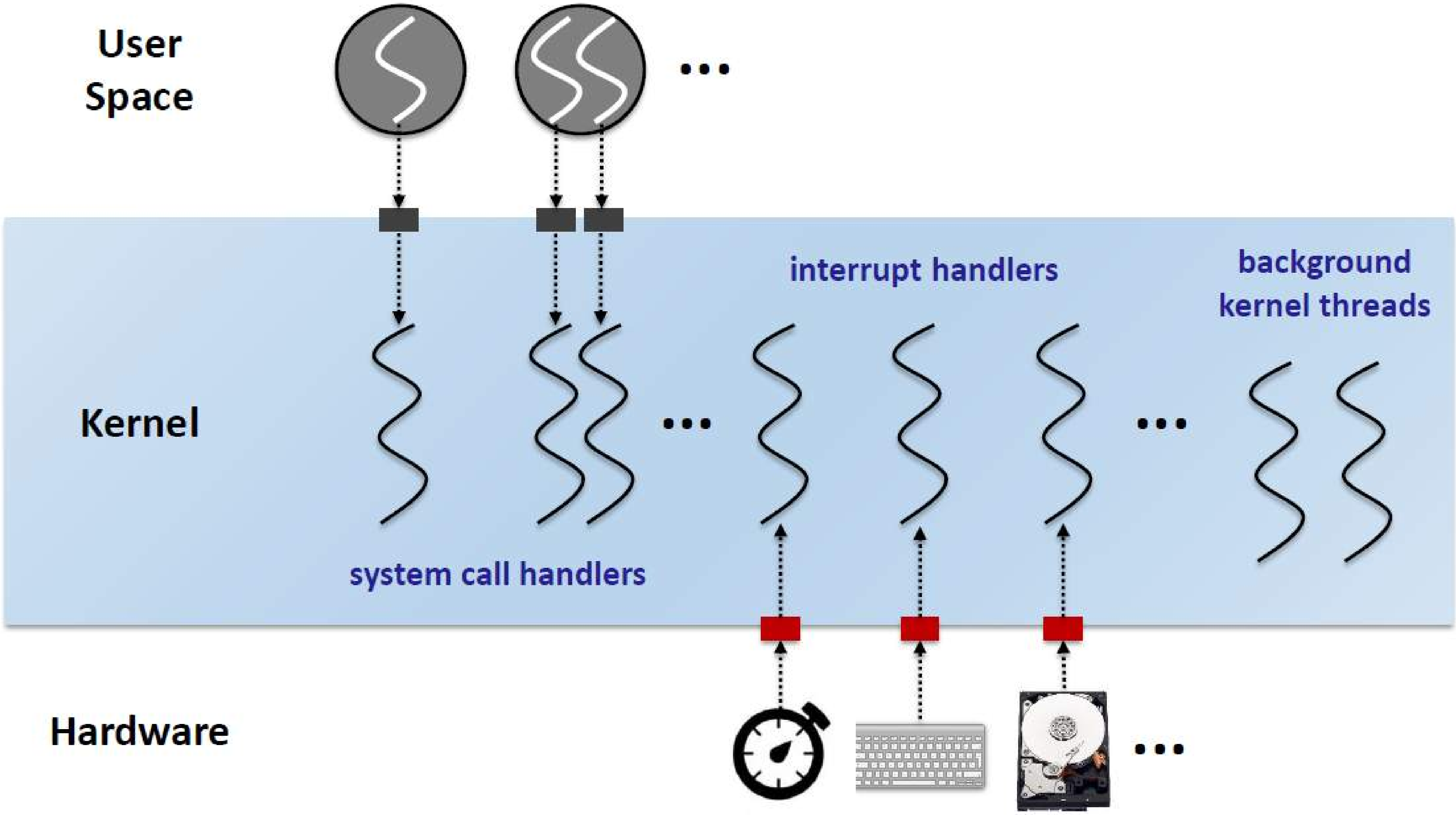
- Concurrency leads to non-deterministic results
  - Two or more concurrent threads accessing a shared resource create a race condition
  - The output of the program is not deterministic; it varies from run to run even with same inputs, depending on timing
  - Hard to debug ("Heisenbugs")

똑같은 input에도 다른결과 디버깅 어려워

- We need synchronization mechanisms for controlling access to shared resources
  - Synchronization restricts the concurrency
  - Scheduling is not under programmer's control

동기화메커니즘 필요!!

Concurrency in the Kernel



Critical Section

- A critical section is a piece of code that accesses a shared resource, usually a variable or data structure

```
movl 0x1000, %eax
addl $1, %eax
movl %eax, 0x1000
```

Critical section

- Need mutual exclusion for critical sections
  - Execute the critical section atomically (all-or-nothing) critical section을 아예 다 실행하거나 아예 실행되지 않도록 하거나
  - Only one thread at a time can execute in the critical section 하나만 실행
  - All other threads are forced to wait on entry 다른건 기다려
  - When a thread leaves a critical section, another can enter 하나가 끝나면 다른거 ㄱㄱ

Locks: The Basic Idea

그래서 Lock을 이용하자

- 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

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

Locks: The Basic Idea

Lock 변수는 Lock의 상태를 가지고 있음

- Lock variable holds <u>the state of</u> the lock

두가지

  - **available** (or **unlocked** or **free**)
    - ➢ No thread holds the lock

lock을 소유한 thread가 없을때

  - **acquired** (or **locked** or **held**)

있을때

    - ➢ Exactly one thread holds the lock and presumably is in a critical section

The semantics of the lock()

- `lock()`
  - **Try to** acquire the lock lock을 얻으려하는 동작.
  - If <u>no other thread holds</u> the lock, the thread will **acquire** the lock 아무도 안가지고 있으면 호출한 쓰레드가 가져
  - Enter the *critical section*
    - ➢ This thread is said to be <u>the owner of</u> the lock
  - Other threads are *prevented from* entering the critical section while the first thread that holds the lock is in there 다른애가 못들어와
- `unlock()`
  - Once <u>the owner of</u> the lock calls `unlock()`, the lock is now available (free) again lock을 놓고
  - Wake up any thread waiting in `lock()` — 기다리던애들이 가져가게끔 함.

Using locks

lock 사용법

- Lock is initially free
- Call `lock()` before entering a critical section, and `unlock()` after leaving it
- `lock()` does not return until the caller holds the lock

  lock을 호출한 곳이 lock을 가지는 동안에는 반환 X.
- On failure, a thread can spin (spinlock) or block (mutex)
- At most one thread can hold a lock at a time

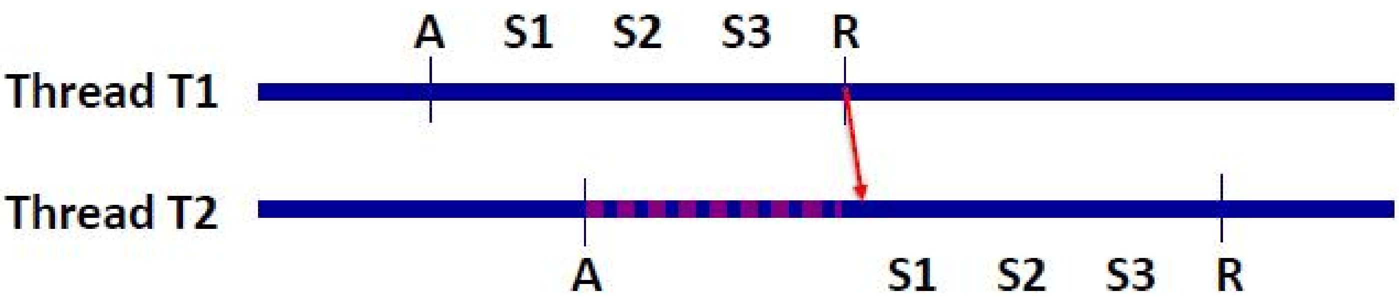
Using Locks (Cont.)

Acquire

Release

	`int withdraw (account, amount)`
	`{`
A	`lock (lock);`
S1	`balance = get_balance (account);`
S2	`balance = balance - amount;`
S3	`put_balance (account, balance);`
R	`unlock (lock);`
	`return balance;`
	`}`

Critical section



Requirements for Locks
Lock의 조건

- Correctness
  - Mutual exclusion: only one thread in critical section at a time
  - Progress (deadlock-free): if several threads want to enter the critical section, must allow one to proceed
    다수의 요청중 하나만 허용
  - Bounded waiting (starvation-free): must eventually allow each waiting thread to enter
    너무 오래 기다리게하면 X

- Fairness 공평성
  - Each thread gets a fair chance at acquiring the lock
    lock의 기회가 고르게 주어져야해

- Performance
  - The time overheads added by using the lock

lock 사용시의 overhead 체크

lock을 얻지못한애가 너무오래기다리면 성능이↓

Implementing Locks

위에서 하나씩

- Controlling interrupts

- Software-only algorithms
  - Dekker's algorithm (1962)
  - Peterson's algorithm (1981)
  - Lamport's Bakery algorithm for more than two processes (1974)

- Hardware atomic instructions
  - Test-And-Set
  - Compare-And-Swap
  - Load-Linked (LL) and Store-Conditional (SC)
  - Fetch-And-Add

Controlling Interrupts

임계 구역에서의 interrupt는 무시

- Disable Interrupts for critical sections
  - One of the earliest solutions used to provide mutual exclusion 초창기 구현 방식
  - Invented for <u>single-processor</u> systems single에서
  - Disabling interrupts blocks external events that could trigger a context switch (e.g. timer) 원리는 임계구역에서의 context switch 를 막으려고.
  - The code inside the critical section will not be interrupted
  - There is no state associated with the lock

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

Controlling Interrupts (Cont.)

특권을 주는거라

- Disable Interrupts for critical sections
  - Simple
  - Useful for a single-processor system
  - Problem:
    - ➢ Require too much *trust* in applications
      - ✓ Greedy (or malicious) program could monopolize the processor

program이 인위적으로 CPU 독점가능

독점

    - ➢ Do not work on multiprocessors
    - ➢ Turning off interrupts for extended periods of time can lead to interrupts becoming lost → 중요한 interrupt 을 놓칠 수 있음.

CPU 하나에서 interrupt 무시해봤자 다른 CPU에서 critical section 접근하면 그만임

Peterson's Solution

- Solution for two processes synchronization

- The two processes share two variables
  - int turn; 누가 들어갈지 결정
    - ➢ whose turn it is to enter the critical section
  - boolean flag[2]; 프로세스가 들어갈 준비가 됐는지?
    - ➢ indicates if a process is ready to enter the critical section
    - ➢ flag[i] = true, if process P_i is ready

Peterson's Solution (Cont.)

소프트웨어적으로 lock을 구현하는 예

flag[j] means "j want to enter".
'turn == j' means "This is j's turn"
➔ No operation

Entry section

```
while (true) {
    flag[i] = TRUE;
    turn = j;
    while ( flag[j] && turn == j);
    critical section

    flag[i] = FALSE;

    remainder section
}
```

Exit section

Algorithm for process P_i

Entry section

```
while (true) {
    flag[j] = TRUE;
    turn = i;
    while ( flag[i] && turn == i);
    critical section

    flag[j] = FALSE;

    remainder section
}
```

Exit section

Algorithm for process P_j

Why hardware support needed?

- First attempt: Using a *flag* denoting whether the lock is held or not
  - The code below has problems

```
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;
}
```

Why hardware support needed? (Cont.)

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

Thread1	Thread2
call `lock()` lock호출하고	
`while (flag == 1)` 반복문을 통과했고	
interrupt: switch to Thread 2 flag set하려는데 context switch	
	call `lock()`
	`while (flag == 1)` 여기서 flag를 set해버림.
	`flag = 1;`
	interrupt: switch to Thread 1
`flag = 1;` // set flag to 1 (too!)	

lock을 호출한 애가 flag를 set하고 나중에 들어오는 애를 while에 가둬야 하는데 둘다 통과한 꼴

– **Problem 2:** Spin-waiting wastes time waiting for another thread

이것에 낭비하는 시간이 아까워.

- So, we need an atomic instruction supported by Hardware!

명령어가 atomic할 필요가 있다.

  – *test-and-set* instruction, also known as *atomic exchange*

원자적으로 실행되게 하는 명령어

Test And Set (Atomic Exchange)

- An instruction to support the creation of simple locks

```c
int TestAndSet(int *ptr, int new) {
    int old = *ptr;   // fetch old value at ptr
    *ptr = new;       // store 'new' into ptr
    return old;       // return the old value
}
```

  - **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

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

void init(lock_t *lock) {
    // 0 indicates that lock is available,
    // 1 that it is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ;          // spin-wait
}

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

return 값.

나중에 flag 를 set 하는게 아닌 함수로 한번에 flag를 바꿔서 해결.

– **Note:** To work correctly on *a single processor*, it requires <u>a preemptive scheduler</u>

올바르게 작동하려면 선점스케줄해야 함.

Evaluating Spin Locks

- Correctness: yes
  - The spin lock only allows a single thread to entry the critical section   임계구역에 하나만 접근하는가? yes
- Fairness: no   공평한가? No 다른놈은 계속 기다려,
  - Spin locks <u>don't provide any fairness</u> guarantees
  - Indeed, a thread spinning may spin *forever*
- Performance:
  - In the single CPU, performance overheads can be quite *painful*
  - If the number of threads roughly equals the number of CPUs, spin locks work *reasonably well*   학습 톡톡

Compare-And-Swap

expected라는 변수추가

- 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

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

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

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

Spin lock with compare-and-swap

So Much Spinning

- Hardware-based spin locks are simple and they work

간단하지만 spinning 중 낭비되는 시간 ↑

- 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

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

OS의 지원으로 이를 피할수 있다

A Simple Approach: Just Yield

- When you are going to spin, give up the CPU to another thread spin에 있는 thread가 CPU를 포기하게끔
  - OS system call moves the caller from the *running state* to the *ready state* ready로 바뀜
  - The cost of a **context switch** can be substantial and the **starvation** problem still exists 하지만 여전히 context switch cost는 있고 계속 양보만해서 일을못하는 starvation도 생김.

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

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

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

Lock with Test-and-set and Yield

Using Queues: Sleeping Instead of Spinning

- Queue to keep track of which threads are <u>waiting</u> to enter the lock

lock에 접근하려고 기다리는 thread들을 queue 저장

- `park()`
  - Put a calling thread to sleep
- `unpark(threadID)`
  - Wake a particular thread as designated by `threadID`

Using Queues: Sleeping Instead of Spinning

```c
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();
    }
}
…
```

guard를 얻기 위한 spinning을 하지만 guard를 계속 0으로 초기화하므로 cpu overhead가 낮다. 그럼에도 guard spinning을 줄이는 이유는 다수의 thread가 lock을 하려 할 때 queue에 순차적으로 넣기 위함이다.

학습목록보기

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

Using Queues: Sleeping Instead of Spinning

```
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;
}
```

큐가 비었으면 flag는 0

아니면 wake up

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

Futex

위는 Solaris 이건 Linux

- 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

spinlock은 임계구역이 짧을때
sleeping은 // 길때 유용

아래는 두가지를 모두 사용하는방법

Two-Phase Locks

- A two-phase lock realizes that spinning can be useful if the lock *is about to* be released
  - First phase   처음에 spin 하다가 못얻으면 sleeping 시키기
    - ➢ 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,
  - Second phase
    - ➢ The caller is put to sleep
    - ➢ The caller is only woken up when the lock becomes free later