CDA0017: Operating Systems

Donghyun Kang (donghyun@changwon.ac.kr)

NOSLab (https://noslab.github.io)

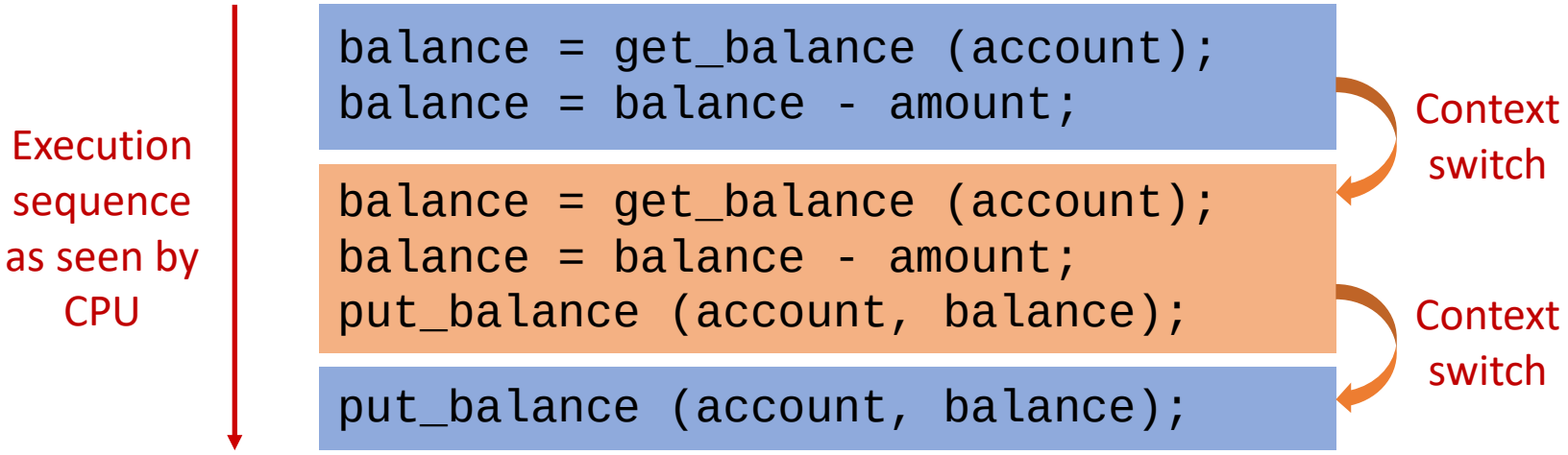
Changwon National University

예제

- 돈 인출 시스템
  - 2명의 친구가 100만원이 입금된 계좌를 공유하고 있음
  - 이때, 동시에 10만원씩 인출 한다면?

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

예제


Execution sequence as seen by CPU
balance = get_balance (account);
balance = balance - amount;
balance = get_balance (account);
balance = balance - amount;
put_balance (account, balance);
put_balance (account, balance);
Context switch
Context switch

자원 공유

- 지역 변수의 경우 쓰레드 간 공유하지 않음
  - 각 변수는 stack 영역에 저장됨
  - 각 쓰레드는 독립된 stack 영역을 가짐

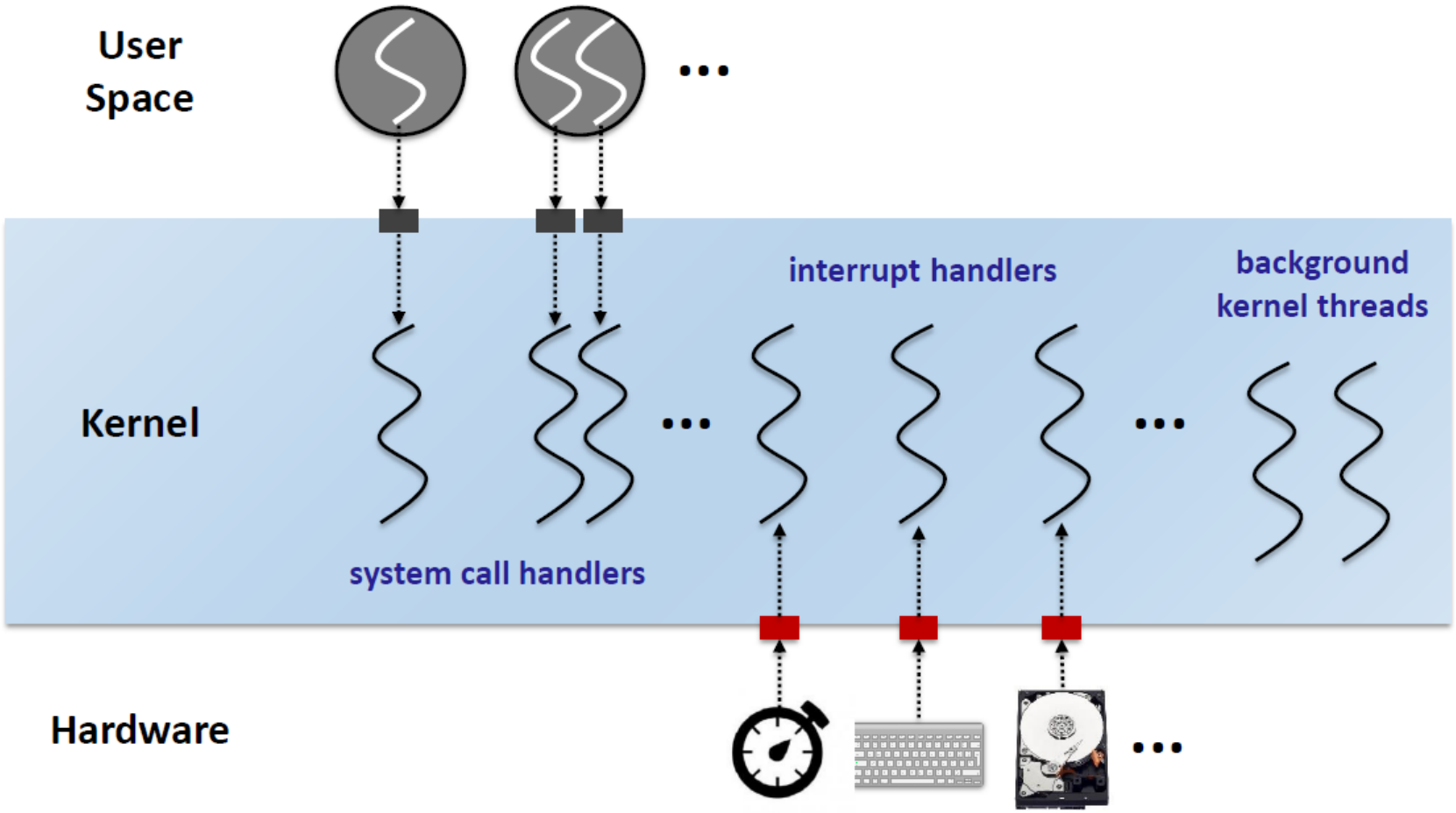
- 전역 변수의 경우 쓰레드 간 공유
  - 전역 변수, static 변수
- 동적 메모리 객체의 경우 쓰레드 간 공유
  - Heap

동기화 문제 발생

- 동시에 접근하는 데이터에 대한 결과의 오류 발생
  - 2개 이상의 쓰레드가 동시에 공유 데이터에 접근하는 경우 race condition이 발생함
  - 프로그램의 동작을 항상 보장할 수 없음
  - 디버깅이 어려움

- 공유 데이터 접근에 대한 동기화 메커니즘이 필요함
  - 병행성을 제공하면서 동기화 가능
  - 스케줄링 정책과 무관한 동기화 가능

커널의 병행성


User
Space
Kernel
interrupt handlers
background
kernel threads
system call handlers
Hardware

락

- 임계영역에 대한 하나의 명령어 처리를 보장하기 위한 방법

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

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

락

- 락 변수가 락의 상태(state)를 저장함
  - 사용 가능(available / unlocked / free)
    - 락을 획득한 쓰레드가 없는 상태
  - 사용 중 (acquired)
    - 하나의 쓰레드가 임계영역에 진입하였으며 락을 획득한 상태

Lock() / unlock()

- Lock()
  - 락을 획득하기 위한 시도
  - 락을 획득한 쓰레드가 존재하지 않는 경우, 호출한 쓰레드가 락 획득
    - 임계 영역으로 진입함
  - 락을 획득한 쓰레드를 소유자라고 부름

- Unlock()
  - 락 소유자가 락을 해제함
  - Lock()을 호출한 쓰레드을 깨우는 동작 포함

락 구현

- 락 초기화

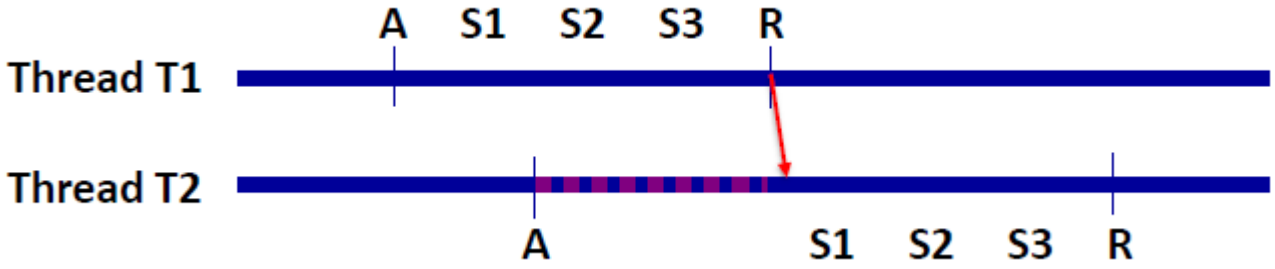
- Lock() 을 통해 임계 영역 진입 → unlock() 을 통해 락 해제
  - Lock() 호출 시, lock을 획득할 때까지 대기

- 하나의 1개의 쓰레드만 임계영역 진입을 보장

예제

```
A     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



락의 평가

- 정확성(correctness)
  - 상호 배제: 하나의 쓰레드만 특정 시점에 임계 영역에 진입 가능
  - 데드락 (deadlock) 이슈: 락을 획득한 쓰레드가 다른 락을 획득하기 위해 기다리는 상태에서 발생
  - Starvation 이슈: 락을 획득하지 못하는 경우 발생
- 공정성(fairness)
  - 모든 쓰레드가 락을 획득하기 위해 공정한 기회를 부여 받았는지 평가
- 성능 (performance)
  - 락 사용을 위한 시간적인 오버헤드가 평가

락의 구현 방식

- 인터럽트 기반
- 소프트웨어 알고리즘 기반
  - Dekker's algorithm (1962)
  - Peterson's algorithm (1981)
  - Lamport's Bakery algorithm for more than two processes (1974)
- 하드웨어 명령어 기반
  - Test-And-Set
  - Compare-And-Swap
  - Load-Linked (LL) and Store-Conditional (SC)
  - Fetch-And-Add

인터럽트 제어

- 인터럽트 기반 상호 배제 지원
  - 쓰레드에게 인터럽트 관련 특권(privileged) 제공
  - 임계 영역에 진입할 때 인터럽트 비활성화(disable)
  - Single-processor 환경에서 사용 가능
  - 외부 이벤트 처리 불가능
    - 예: 문맥 교환(context switch)

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

인터럽트 제어

- 장점
  - 단순함
  - Single-processor에서 효율적임
- 단점
  - 사용자의 쓰레드에 대한 신뢰가 필요함
  - Multi-processor 환경에서 사용 불가
  - 인터럽트 활성화/비활성화 코드는 상대적으로 느림

하드웨어 명령어의 필요성

- Flag 기반 락 획득 및 해제

```
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   }
```

하드웨어 명령어의 필요성

- **Problem 1**: **상호 배제 불가능**(assume `flag=0` to begin)

Thread1	Thread2
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!)	

- **Problem 2**: Spin-waiting 기반 CPU 소모

- 즉, 하드웨어 기반 원자적 락 획득 / 해제 명령어가 필요함
  - *test-and-set* instruction

Test And Set (원자적 교체)

- 단순한 방식으로 락 구현을 지원함

```
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    }
```

  - 원자적으로 두개의 동작이 이루어짐
    - Return: 이전 값 반환
    - 동시에 메모리에 새로운 값 설정

Test-and-set 기반 스핀 락

```
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  }
```

- 선점형 스케줄러 사용시 사용 가능

스핀 락의 평가

- 정확성: OK
  - 스핀 락은 임계 영역에 하나의 쓰레드만 진입하도록 보장함
- 공정성: NOK
  - 락 획득에 대한 보장 불가
    - 무한 반복 가능
- 성능: NOK / OK
  - Single-processor 환경에서 CPU 소모
  - Multi-processor 환경에서 쓰레드의 개수가 같은 경우, 합리적으로 동작 가능

Compare-And-Swap

```
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)

- Ptr의 값이 expected 변수의 값과 일치하는지 여부 검사
  - 같은 경우, 새로운 값으로 변경
  - 다른 경우, none

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

Spin lock with compare-and-swap

Load-Linked and Store-Conditional

```
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  }
```

Load-linked And Store-conditional

- 임계 영역 진입을 위한 함수의 쌍 제공
  - Load-linked 명령 호출 후 StoreConditional 명령을 호출함으로써 값 변경
    - 성공: 새로운 값을 저장하고 1 반환
    - 실패: 0 반환

Load-Linked and Store-Conditional

```
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

Fetch-And-Add

- 원자적으로 특정 주소의 예전 값을 반환하면서 값을 증가시킴

```
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)

티켓 락(Ticket Lock)

- Fetch-And-Add 기반 티켓 락 구현
  - 티켓 / 차례의 조합으로 락 구현
  - 락을 획득하기 위해 순서를 기다림 → 공정성(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  }
```

과도한 스핀

- 하드웨어 기반의 락은 간단하고 제대로 제대로 동작함

- 그러나, 비효율적으로 수행될 수 있음
  - 특정 쓰레드가 락을 획득한 상태에서 문맥 교환이 발생하는 경우, 동일한 락을 획득하기 위한 다른 쓰레드는 CPU 시간을 소모함

운영체제의 도움이 필요함!!

간단한 접근법: 무조건 양보

- 락을 획득하기 위해 스핀하는 경우, CPU을 양보함
  - 운영체제는 쓰레드의 상태를 실행 → 대기로 변경
  - 무조건 양보는 starvation 문제를 발생시킴

```
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  }
```

Lock with Test-and-set and Yield

큐 사용

- 어떤 쓰레드가 다음으로 락을 획득할지 명시적으로 제어
  - `park()`
    - 쓰레드를 잠재움
  - `unpark(threadID)`
    - 특정 쓰레드 ID에 해당하는 쓰레드를 깨움

큐 사용

```
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;
5       m->guard = 0;
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();
19      }
20  }
21  …
```

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

큐 사용

```
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.)

깨우기/대기 겨쟁

- 쓰레드 B가 park() 을 호출하기 직전에 쓰레드 A가 락을 해제하면?
  - 쓰레드 B는 영원히 잠들 수 있음

- 솔라리스는 이 문제를 해결하기 위해 setprk() 추가
  - Park()가 호출되기 전에 다른 쓰레드가 unpark()를 먼저 호출
    - Park() 는 잠을 자는 대시 바로 반환

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

Code modification inside of `lock()`

Futex

- 리눅스는 park() / unpark()와 유사한 개념인 futex 을 제공함
  - `futex_wait(address, expected)`
    - 예상 값과 같다면, 쓰레드를 잠재움
    - 예상 값과 다르면, 즉시 반환
  - `futex_wake(address)`
    - 대기 큐의 쓰레드 중 하나를 깨움

Futex

- NPTL(Native POSIX Thread Library) 라이브러리의 Lowlevellock.h
  - 최상위 bit는 락 획득 여부 저장
  - 다른 bit는 락을 기다리는 쓰레드의 개수 저장

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

```
16        if (v >= 0)
17            continue;
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.)

2단계 락

- 2단계 락을 통해 스핀 락을 효율적으로 사용함
  - 1단계
    - 락 획득이 바로 가능하다고 생각하기 때문에 스핀하며 대기함
    - 첫 번째 회전으로 락 획득에 실패하면 2단계로 넘어감
  - 2단계
    - 락 획득을 위한 쓰레드를 잠재움
    - 락이 해제되는 경우 깨어남

Q&A