15. Locks

critical section 구현하기 위해 사용하는 lock, mutex에 대해 더 배워보자!

▼ Evaluating Locks

1. Mutual exclusion : critical section을 위한 mutual exclusion을 보장해주는가?
2. Fairness : 여러 개의 thread가 진입할 때 starvation이 발생할 수 있는가?
3. Performance : 기존 sw의 성능이 lock 때문에 나빠지지는 않는가?

▼ implement Lock

▼ 1. Controlling Interrupts

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

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

lock을 구현할 수 있는 가장 쉬운 방법

- but, 많은 단점이 있는 code
  - 호출하는 thread는 root 권한으로만 실행 가능(priviliged operation)
  - multiprocessor에서 제대로 작동 x
    - core 별로 다르게 interrupt disable 해야 하기 때문에 context switch 발생 가능(timer interrupt 발생)
  - 성능 저하

⇒ 복잡한 interrupt handling situation들을 방지했을 때만 사용 가능한 ver

▼ 2. Spin Locks with Loads/Stores

```c
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;
}
```

↳lock을 얻음

- No mutual exclusion → th1, th2 모두 critical section 내부로 진입 ⇒ mutual exclusion 보장 x

Thread 1	Thread 2
Call lock()	
while (flag == 1) ?	
Context switch → flag 값을 바꾸지 않은 채로 thread 2 context switch	
	Call lock()
	while (flag == 1) ?
	flag = 1;
	Context switch
flag = 1;	

flag 값을 바꾸는 작업을 또 실행

→ 계속 cpu 자원을 waste하면서 spining

⇒ performance problem 발생!

▼ 3. Spin Locks with Test-and-Set

spin lock 문제 → mutual exclusion 보장 x

while문이

⇒ atomic하게 실행되지 않아서 flag 값을 세팅, 확인하는 과정에서 preemption 발생

⇒ atomic instruction 적용해보자!

↳Test-and-Set

pseudo code

```
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
}
```

```
typedef struct __lock_t { int flag; } lock_t;
void init(lock_t *lock) {
  lock->flag = 0;
}

void lock(lock_t *lock) {
  //어떤 특정 메모리 영역에 내가 원하는 value로 setting
  while (TestAndSet(&lock->flag, 1) == 1);
}

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

- Not fair ⇒ starvation 발생 가능
- single CPU case ⇒ ready queue 내부에서 time slice가 모두 소진될 때까지 기다려야 함
  - performance overhead 너무 커짐

▼ 4. Spin Locks with Compare-and-Swap

test and set 대신에 compare and swap 사용

pseudo code

```
int CompareAndSwap(int *ptr, int expected, int new) {
  int actual = *ptr;
  if (actual == expected) //기댓값과 같다면 새로운 값으로 바꿈
    *ptr = new;
  return actual; //ptr memory 영역에서 가지고 있던 원래 값으로 바꿈
}
```

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

1. lock acquire → flag가 0이길 expect
    a. set 1
    b. return flag
2. 원래의 flag 값
    a. 1 ⇒ 다른 thread가 이미 lock 얻어서 계속 while문 spin
    b. 0 ⇒ lock acquire success

⇒ but, spin lock을 사용하기에 fairness, performance 아직도 bad

▼ 5. Ticket Locks with Fetch-and-Add

- fairness 해결하기
    - fetch-and-add atomic instruction

```
//lock을 기다리는 순서대로 번호 부여 -> 그 순서대로 lock acquire
nt FetchAndAdd(int *ptr) {
  int old = *ptr;
  *ptr = old + 1;
  return old;
}
```

```
//ticket lock 기반으로 mutex lock 구현
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);
}

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

atomic instruction으로 구현하지 x → 동일한 번호를 가진 thread or 동시에

Hardware Support

- Locks with hw support → 너무 spin

⇛ 해결책

- yield() : spining 좀 완화 → cpu 자원 포기한다는 system call 호출

```
void lock(lock_t *lock) {
  while (TestAndSet(&lock->flag, 1) == 1)
    yield();
}
```

- still costly and unfair
  - yield해도 다시 그 thread 선택 가능(ex. vruntime이 가장 낮은 thread일 경우)

⇛ 여전히 spinnng 문제 존재

→ OS의 역할이 필요

+) 많은 thread가 RR에 의해 schedule되는 경우를 생각해보자.

▼ OS Support

- sleeping(어떤 thread를 재우는 기능) → spining 대신 사용

  ① in solaris

  - **park()** : calling thread가 sleep
  - **unpark(threadID)** : *threadID*인 thread wake

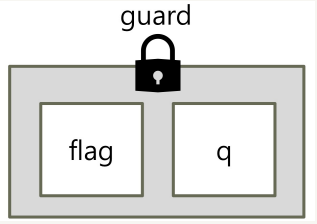
  ② in linux

  - **futex_wait(address, expected)** : address의 value = expected → process sleeping
  - **futex_wake(address)** : waiting queue에서 한 thread를 골라서 wake

- Locks with Queues
  - **futex_wake(address)** : waiting queue에서 한 thread를 골라서 wake

```
typedef struct __lock_t {
  int flag; // lock
  int guard; // spin-lock around the flag and
             // queue manipulations
  queue_t *q;
} lock_t;

void lock_init(lock_t *m) {
  m->flag = 0;
  m->guard = 0;
  queue_init(m->q);
}
```



*solaris 버전 system call 사용한 ver

```
void lock(lock_t *m) {
  while (TestAndSet(&m->guard, 1) == 1);
    if (m->flag == 0) ← flag를 1로 바꿈
      m->flag = 1; // lock is acquired
      m->guard = 0;
    } else { → 큐로 진입
      queue_add(m->q, gettid());
      setpark(); ⇒ another thread calls unpark before
      m->guard = 0; // park is actually called, the
      park(); // subsequent park returns immediately
    }
}

void unlock(lock_t *m) {
  while (TestAndSet(&m->guard, 1) == 1);
  if (queue_empty(m->q))
    m->flag = 0;
  else
    unpark(queue_remove(m->q));
  m->guard = 0;
}
```

*setpark가 없다면 race condition 발생 가능 ⇒명확히 정의는 thread 발생가능

이미 다른 thread가 park을 호출하기 전에 unpark을 호출했다면 연속적인 park가 즉각적으로 return 됨.

*성능개선도 가능한 ver