Interrupts

Kook, Joongjin
(tipsiness@gmail.com)
Operating Systems Lab.
Soongsil University
2009.03. 31

Interrupt

- 하드웨어가 리눅스 커널 기능을 호출하는 방법
- **OS**에서는 디바이스를 효율적으로 제어하기 위해 인터럽트 사용
- 하드웨어를 실제로 사용할 수 있을 때 까지 **CPU**는 다른 작업 수행 가능
- 멀티태스킹 능력 향상
- 동시에 여러 디바이스에 대한 처리 가능

프로세스 컨텍스트 vs. 인터럽트 컨텍스트

프로세스 컨텍스트

- 커널이 실행중인 상태
- 시스템콜 또는 커널 스레드가 동작되는 상태
- current 매크로는 컨텍스트와 관련된 태스크를 가리킴
- 프로세스와 관련되어 있으므로 휴면하거나 스케줄러 호출 가능

인터럽트 컨텍스트

- 인터럽트 핸들러나 bottom half를 실행중인 상태
- current 매크로의 의미가 없음
- 관련된 프로세스가 없으므로 휴면 불가능(rescheduling 불가능)
- 휴면이 가능한 함수들은 인터럽트 컨텍스트에서 사용 불가
- 다른 코드를 중단시킨 상태에서 실행되므로 시간 제약을 가짐
- 빠르고 간단해야 하며, busy-wait이 발생하지 않도록 해야함

인터럽트 컨텍스트

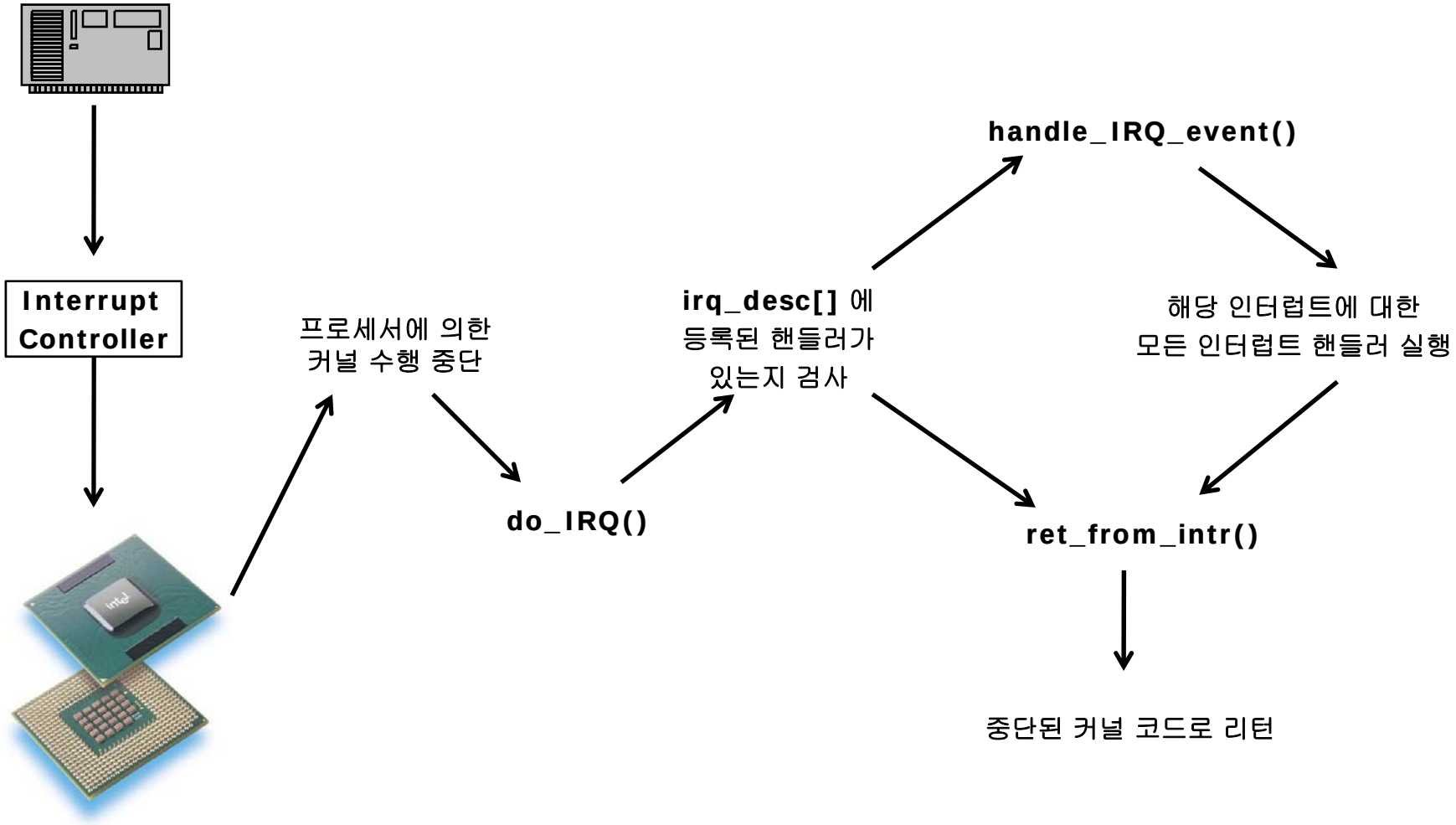
- 인터럽트 컨텍스트
  - 인터럽트 핸들러의 스택 설정은 환경 설정에 따라 조정 가능
  - 중단된 프로세스의 커널 스택 공유
  - 인터럽트 스택: 프로세서 별로 1페이지 크기의 독점적 스택 할당

응답성

- **UNIX**에서는 응답성을 확보하기 위해 인터렙트 레벨 사용
  - 높은 응답성을 필요로 하는 인터럽트에게 높은 우선순위를 주는 방식

- 리눅스에서는 인터럽트 처리에 대해 높은 응답성을 필요로 하는 부분**(top half)**과 필요로 하지 않는 부분**(bottom half)**으로 구분
  - 응답성을 필요로 하지 않는 부분은 나중에 처리

인터럽트 핸들링


Interrupt Controller
프로세서에 의한 커널 수행 중단
do_IRQ()
irq_desc[] 에 등록된 핸들러가 있는지 검사
handle_IRQ_event()
해당 인터럽트에 대한 모든 인터럽트 핸들러 실행
ret_from_intr()
중단된 커널 코드로 리턴

하드웨어 인터럽트

- 인터럽트 핸들러 등록
  - *int request_irq(*
    ```
    unsigned int irq,
    irqreturn_t (*handler)(int, void *, struct pt_regs *),
    unsigned long irq_flags,
    const char * devname, void *dev_id
    )
    ```
  - *irq_flags:*
    - *SA_SHIRQ : Interrupt is shared*
    - *SA_INTERRUPT : Disable local interrupts while processing*
    - *SA_SAMPLE_RANDOM : The interrupt can be used for entropy → call add_interrupt_randomness()*

하드웨어 인터럽트

- 인터럽트 핸들러
  - *irqreturn_t (*handler)(int, void *, struct pt_regs *)*
  - *ex: irqreturn_t int_handler(*
    int irq,
    *void * dev_id,*
    *struct pt_regs *regs*
)

하드웨어 인터럽트

- 인터럽트 관련 함수
  - *local_irq_disable() : 인터럽트 비활성화*
  - *local_irq_enable() : 인터럽트 활성화*
  - *local_irq_save() : 현재 인터럽트 상태 저장*
  - *local_irq_restore() : 인터럽트 상태 복구*

  - *irq_disabled() : 로컬 프로세서의 인터럽트가 활성화된 경우 0을 리턴*
  - *in_interrupt() : 프로세스 컨텍스트이면 0*
  - *in_irq() : 인터럽트 핸들러를 실행중이면 0*

멀티프로세서 환경의 고려

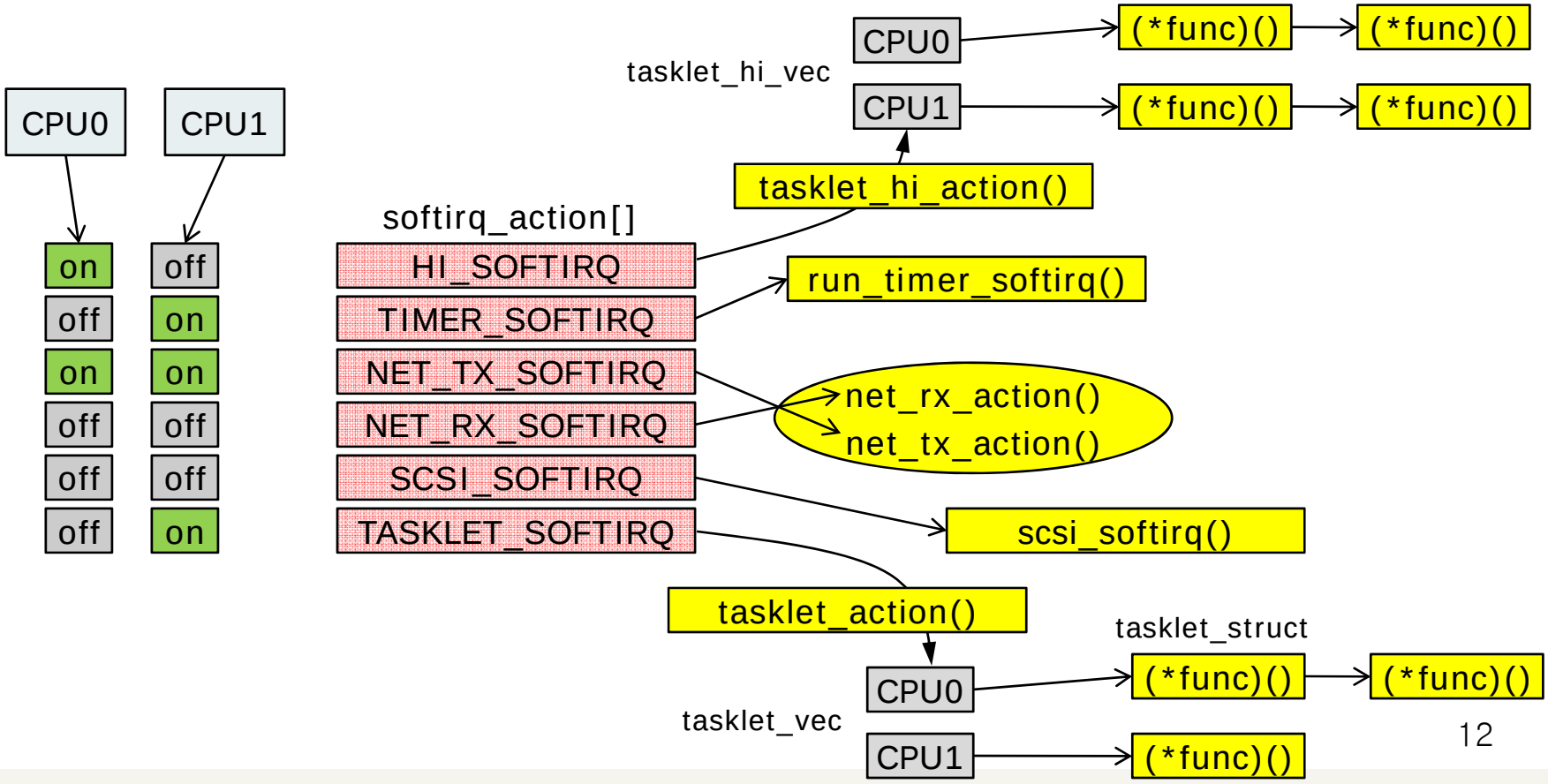
- **CPU**에 관계없이 모든 인터럽트 처리 가능
- 같은 **IRQ**가 아닌 한 **CPU**별로 각각의 하드웨어 인터럽트 핸들러 실행 가능
- 복수의 **CPU**에서 동시에 소프트 인터럽트 핸들러 수행 가능, 같은 종류의 소프트 인터럽트 핸들러인 경우에도 병렬로 실행 가능
- 소프트 인터럽트 핸들러는 그 요인으로 작용한 하드웨어 인터럽트 핸들러와 같은 **CPU**상에서 동작함으로써 캐시 효율 향상
- 특정 인터럽트의 처리를 명시적으로 특정 **CPU**에 할당 가능

커널이 취급하는 인터럽트

- 외부 장치 인터럽트
  - SCSI 호스트 어댑터, 네트워크 카드, 단말 장치 등이 생성하는 인터럽트
- 타이머 인터럽트
  - 타이머 컨트롤러가 주기적으로 발생시키는 인터럽트
  - 리눅스는 글로벌 타이머 인터럽트와 로컬 타이머 인터럽트 이용
- 프로세서 간 인터럽트
  - 멀티프로세서 환경에서 다른 CPU에 의한 이벤트 통지를 위한 인터럽트
- **NMI**
  - 통상의 인터럽트는 마스크에 의해 CPU가 인터럽트 수신을 제어할 수 있지만, NMI는 마스크가 불가한 인터럽트로 긴급한 상황에 대한 장해 대응 목적으로 사용

인터럽트 처리의 지연 – softirq(1)

- 소프트 인터럽트
  - 하드웨어 인터럽트 핸들러의 요청에 의해 발생
  - 하드웨어 인터럽트 핸들러에서 수행한 처리를 물려받아 동작



인터럽트 처리의 지연 – softirq(2)

- 소프트 인터럽트 제어
  - 시스템콜 처리와 소프트 인터럽트 핸들러 사이에서 경쟁하는 자원이 있을 경우 상호 배제 처리 필요
  - 하드웨어 인터럽트 금지를 통해 소프트 인터럽트 금지 가능
  - 소프트 인터럽트 제어 함수 (softirq, tasklet)
    - local_bh_disable : 소프트 인터럽트 핸들러의 실행 금지
    - local_bh_enable : 소프트 인터럽트 핸들러의 실행 허가

인터럽트 처리의 지연 – softirq(3)

- 소프트 인터럽트 실행
  - 소프트 인터럽트가 요청되면 소프트 인터럽트 핸들러가 호출됨
  - 하드웨어 인터럽트 핸들러 종료 시 do_softirq 함수 호출
  - do_softirq 함수는 대기 상태에 있는 소프트 인터럽트의 요청을 조사한 후 해당 핸들러를 순차적으로 호출
  - 소프트 인터럽트 핸들러가 실행 중일 때 동일한 하드웨어 인터럽트가 발생하면, 핸들러 종료 시 소프트 인터럽트 핸들러를 한 번 더 실행
  - 잦은 빈도로 발생되는 인터럽트 요청에 대해 ksoftirqd에 소프트웨어 인터럽트 핸들러의 처리 위탁 (소프트 인터럽트 핸들러의 실행은 커널 스레드 ksoftirqd가 스케줄링 될 때까지 지연)

인터럽트 처리의 지연 – softirq(4)

- 소프트 인터럽트 처리 함수
  - raise_softirq : 소프트 인터럽트 요청 생성
  - raise_softirq_irqoff : 소프트 인터럽트 요청 생성. 인터럽트 금지 상태에서 호출 가능
  - do_softirq : 대기 중인 소프트 인터럽트 요청에 대한 소프트 인터럽트 핸들러를 실행
  - wakeup_softirqd : 커널 스레드 ksoftirqd를 실행하고, 소프트 인터럽트 핸들러의 실행 의뢰

인터럽트 처리의 지연 – tasklet(1)

- 소프트 인터럽트의 확장
- 임의의 함수를 소프트 인터럽트 핸들러로 등록하여 실행 가능
- **tasklet**은 동일한 핸들러 여럿이 서로 다른 프로세서에서 동시에 실행되지 않는다는 제약을 가진 **softirq**

인터럽트 처리의 지연 – tasklet(1)

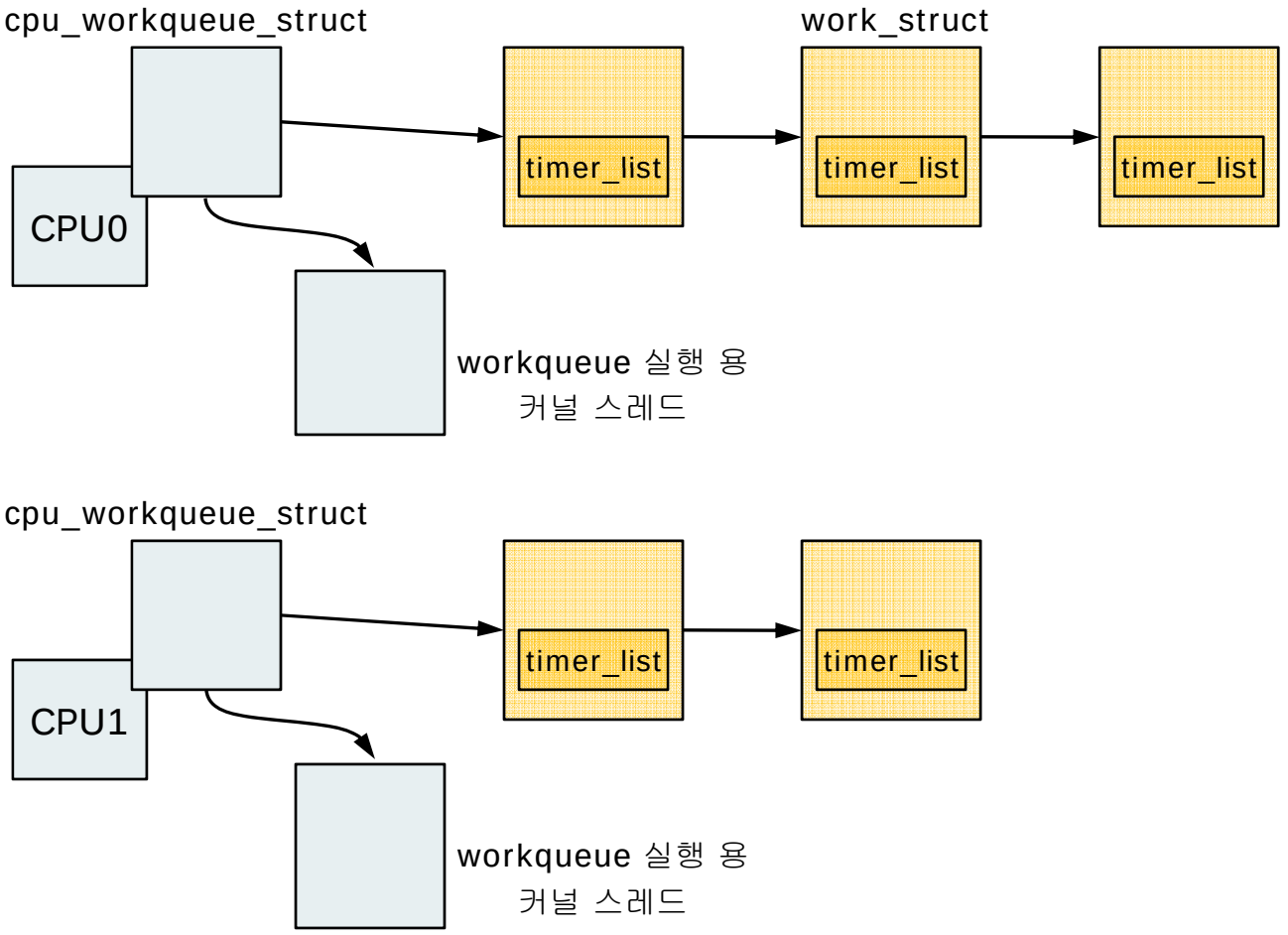
- **tasklet** 조작 함수
  - tasklet_schedule : tasklet 등록(TASKLET_SOFTIRQ)
  - tasklet_hi_schedule : tasklet 등록(HI_SOFTIRQ)
  - tasklet_action : tasklet 실행(TASKLET_SOFTIRQ)
  - tasklet_hi_action : tasklet 실행(HI_SOFTIRQ)
  - tasklet_kill : tasklet 등록 해제
  - tasklet_disable : tasklet 실행 금지
  - tasklet_disable_nosync : tasklet 실행 금지. 지정한 tasklet이 실행 중이었을 경우라도 종료를 기다리지 않음
  - tasklet_enable, tasklet_hi_enable : 지정한 tasklet 실행 허가

인터럽트 처리의 지연 – workqueue(1)

- 주로 프로세스 컨텍스트의 처리 지연에 사용
- 복수의 요청을 **workqueue**에 등록하고 나중에 처리
- **2.4** 커널에서는 인터럽트 컨텍스트와 프로세스 컨텍스트 처리 모두에 **taskqueue** 사용
- **2.6** 커널은 인터럽트 컨텍스트의 처리를 위해 **tasklet**을 사용하고 프로세스 컨텍스트의 처리에는 **workqueue** 사용
- 각종 디바이스 드라이버 처리의 지연 실행, 블록 **I/O** 요청의 지연 실행, 비동기 **I/O** 처리 등에서 사용

인터럽트 처리의 지연 – workqueue(2)

workqueue 구조



인터럽트 처리의 지연 – workqueue(3)

- **workqueue 조작 함수**
  - create_work : workqueue 생성
  - destroy_workqueue : workqueue 삭제
  - queue_work : 지정한 workqueue에 등록. 이 workqueue의 처리를 담당하는 커널 스레드 호출
  - run_workqueue : workqueue에 등록된 작업의 처리를 위해 커널 스레드가 호출
  - flush_workqueue : workqueue에 등록되어 있는 작업의 처리 완료 대기
  - queue_delayed_work : 지정한 workqueue의 처리를 지연시켜 등록
  - cancel_delayed_work : queue_delayed_work 수행 취소

인터럽트 처리의 지연 – workqueue(4)

- **keventd_wq**
  - 범용 workqueue
  - 커널이 부팅될 때 자동으로 준비됨

- **범용 workqueue 조작 함수**
  - schedule_work : 범용 workqueue에 대한 queue_work 함수
  - flush_scheduled_work : 범용 workqueue에 대한 flush_workqueue 함수
  - schedule_delayed_work : 범용 workqueue에 대한 queue_delayed_work 함수
  - schedule_delayed_work_on : schedule_delayed_work 함수에 실행할 CPU 지정

Concurrency

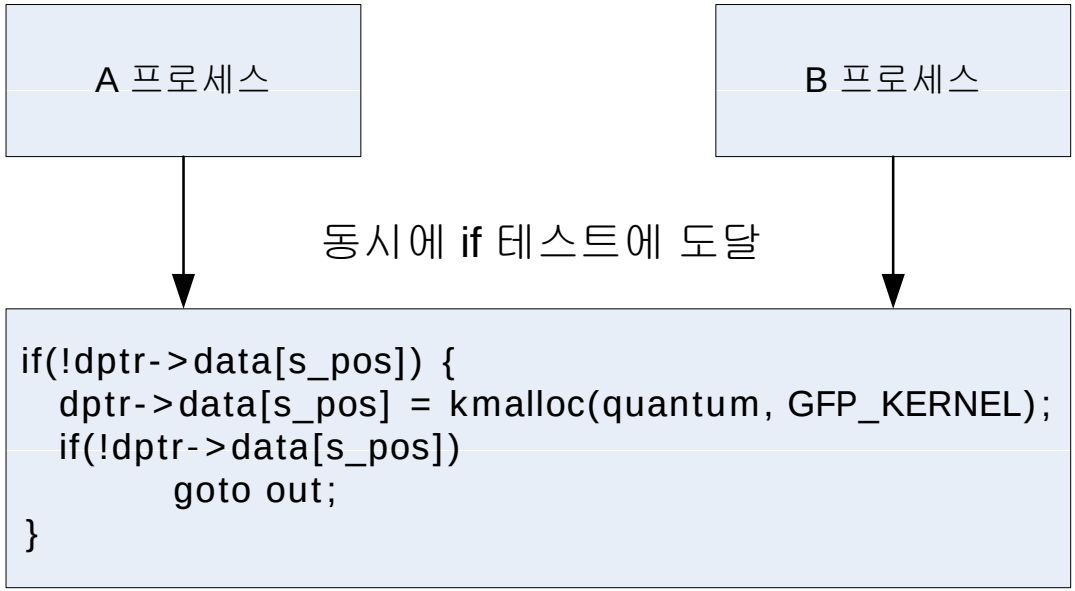
Operating Systems Lab.

Soongsil Univ.

2008.10

왜 동시성 문제가 발생하는가?

- 이유
  - 시스템이 여러 일을 한꺼번에 수행하려 할 때 발생
  - 두 개의 프로세스가 같은 디바이스 내의 write()함수를 수행하는 도중 메모리가 할당 되었는지를 확인하는 코드



  - 어떤 문제가 발생 하겠는가?

동시성과 동시성 관리

- 동시성을 일으키는 원인들
  - SMP 시스템
  - 선점형 커널 코드
  - 디바이스 인터럽트의 비동기식 이벤트
  - 워크 큐, 태스크릿, 타이머와 같은 메커니즘 사용
- 동시성 회피 방법
  - 동시 접근을 제공하는 공유 자원을 가능한 한 회피
    - 전연 변수의 회피
  - 자원에 대한 접근 권한을 직접 관리
    - 잠금, 상호 배제 (세마포어, 뮤텍스, 스핀락)
      - 공유 자원을 한 번에 한 스레드만 조작
  - 자원에 대한 참조 카운터를 처리
- 공유가 필요한 경우
  - 하드웨어 자원, 소프트웨어 자원, 포인터를 다른 부분으로 넘기는 경우

락킹 기법의 종류

- 대기**(sleep)**를 허용하는 **lock**
  - semaphore, MUTEX
  - Reader Writer SEMaphore
  - Completion
- 대기를 허용하지 않는 **lock**
  - spinlocks
  - seqlocks
- 기타 락킹**(locking)** 기법
  - RCU (Read-Copy-Update)
  - atomic variable
  - bit operations

세마포어와 뮤텍스 (1)

- 세마포어
  - <sem/semaphore.h>를 포함
  - struct semaphore 구조체를 사용
  - 세마포어를 직접 생성한 후 초기화
    - void sema_init(struct semaphore *sem, int val);
    - val : 세마포어에 할당할 초기 값
  - *P* 연산 - 세마포어 획득 (세마포어 값을 감소)
    - void down(struct semaphore *sem);
      - signal(인터럽트)에 의해 방해 받지 않음
    - int down_interruptible(struct semaphore *sem);
      - signal (인터럽트)수신 시 빠져 나오면서 에러를 리턴
      - 에러 값을 확인하는 코드를 작성해 주어야 함
    - int down_trylock(struct semaphore *sem);
      - 세마포어을 얻을 수 없는 경우 즉시 0이 아닌 값을 리턴

세마포어와 뮤텍스 (2)

- V연산 – 세마포어 반환 (세마포어 값을 증가)
  - void up(struct semaphore *sem);
    - 메모리 할당 실패, 오류 등이 발생할 경우 필히 세마포어를 풀어주어야 함

사용 예

```
if (down_interruptible(&sem))
        return –ERESTARTSYS;

임계 영역 코드;

up(&sem);

나머지 코드;
```

세마포어와 뮤텍스 (2)

- 뮤텍스
  - #include <asm/semaphore.h> 포함
  - struct semaphore 구조체 이용
  - 뮤텍스 선언과 함께 초기화 매크로
    - DECLARE_MUTEX(name);
      - 뮤텍스가 풀린 상태를 초기화
    - DECLARE_MUTEX_LOCKED(name);
      - 뮤텍스가 잠긴 상태로 초기화
  - 동적인 뮤텍스 초기화 함수
    - void init_MUTEX(struct semaphore *sem);
    - void init_MUTEX_LOCKED(struct semaphore *sem);
  - 뮤텍스 획득과 반환은 세마포어와 동일한 함수 사용

읽기/쓰기 세마포어 (1)

- 세마포어 사용 유형
  - 보호된 자료 구조체를 읽기만 하는 작업
    - 여러 스레드가 동시에 접근 해도 괜찮음
  - 구조체를 변경해야만 하는 작업
- **rwsem(reader/writer semaphore)** 세마포어
  - <linux/rwsem.h>를 포함
  - struct rw_semaphore 구조체 이용
  - 성능의 최적화를 고려한 특수한 세마포어
  - 초기화 함수
    - void init_rwsem(struct rw_semaphore *sem);
  - 읽기 전용 접근 권한을 구현 하는 함수
    - void down_read(struct rw_semaphore *sem);
    - int down_read_trylock(struct rw_semaphore *sem);
    - void up_read(struct rw_semaphore *sem);

읽기/쓰기 세마포어 (2)

- 쓰기 전용 접근 권한을 구현 하는 함수
  - void down_write(struct rw_semaphore *sem);
  - int down_write_trylock(struct rw_semaphore *sem);
  - void up_write(struct rw_semaphore *sem);
  - void downgrade_write(struct rw_semaphore *sem);
    - 쓰기 락을 걸어 수정 후 한 동안은 읽기 전용 권한만 필요한 경우에 downgrade_write 함수를 사용하여 다른 읽기 스레드에게 읽기를 허용
- 우선 순위는 쓰기 스레드에 있음
  - 쓰기 스레드가 임계 구역에 접근하려는 순간, 읽기 스레드는 모든 쓰기 스레드가 작업을 끝낼 때까지 기다림
- rwsem은 쓰기 접근이 드물게 발생하고 쓰기 스레드가 짧은 기간 동안에만 접근 권한이 필요한 경우에 적합

Completion (1)

Completion

- <linux/completion.h> 포함
- 리눅스 2.4.7에서 도입되었으며 세마포어와 동작이 같음
- 다른 프로세서에서 동시에 up()과 down()하는 경우
- struct completion 구조체 사용

문제점

- 세마포어의 문제점은 up, down이 빈번하게 발생할 경우 그 다음 up 연산이 일어나기 전에 세마포어 자료구조가 기다리지 않고 제거 될 수 있다.
  - 이러한 우려로 인해 completion 매커니즘을 추가함

Completion (2)

- **Completion**
  - 세마포어의 up()은 complete()로 대체
  - 세마포어의 down()은 wait_for_completion()로 대체
  - 초기화 매크로 함수
    - DECLARE_COMPLETION(my_completion);
  - 동적으로 초기화 하는 함수
    - init_completion(&my_completion);
  - 완료를 기다리는 함수
    - void wait_for_completion(struct completion *c);
    - 시그널(인터럽트)이 불가능한 대기를 수행, 결과적으로 죽일 수 없는 프로세스가 된다.
  - 완료 이벤트를 보내는 함수
    - void complete(struct completion *c);
      - 대기 중인 스레드 하나만을 깨움
    - void complete_all(struct completion *c);
      - 대기 중인 모든 스레드를 깨움

Completion Example

```
...
DECLARE_COMPLETION(comp);

ssize_t xxx_read(struct file *filp, char __user *buf, size_t count, loff_t *pos)
{
   printk(KERN_DEBUG "process %i (%s) going to sleep\n",
      current->pid, current->comm);
   wait_for_completion(&comp);
   printk(KERN_DEBUG "awoken %i (%s) \n", current->pid, current->comm);
   return 0;
}

ssize_t xxx_write(sturct file *filp, const char __user *buf, size_t count, loff_t *pos)
{
   printk(KERN_DEBUG "process %i (%s) awakening the readers...\n",
      current->pid, current->comm);
   complete(&comp);
   return count;
}
...
```

스핀락(spinlocks) (1)

개 요

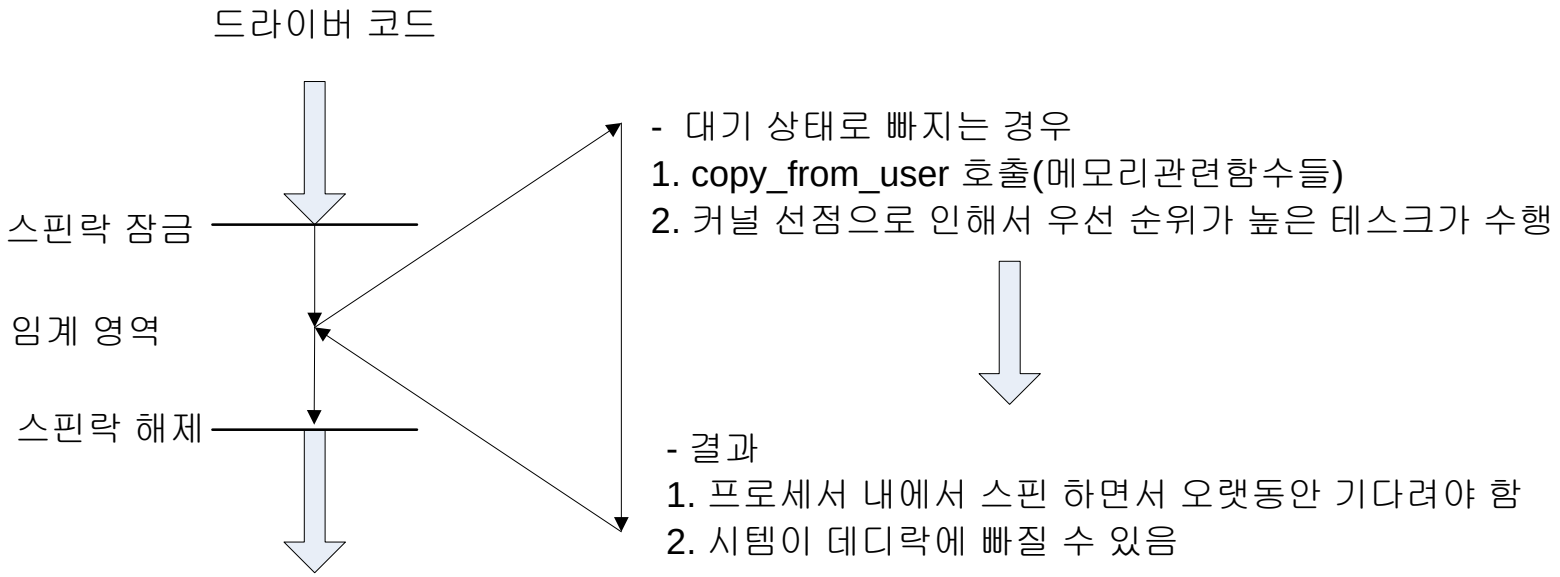
- 커널이 제공하는 상호 배제 도구
- 세마포어 보다는 사실상 스핀락 메커니즘을 이용
- 세마포어와는 다르게 블록 상태로 빠져서는 안 되는 코드에서 사용 가능
- 스핀락 내부 정수에서 한 비트를 이용해 구현
- 누군가 락을 이미 가져가 버렸다면 코드는 작은 루프 상태로 들어가서 주기적으로 락을 확인, 이때의 루프를 스핀(spin)이라 함
- 검사와 설정은 원자적으로 이루어짐
- 스핀락은 다중 프로세서 시스템을 고려함
- 비선점형 단일 프로세서 시스템에서는 스핀락이 아무 작업도 수행하지 않도록 최적화 해야 함

스핀락(spinlocks) (2)

- 스핀락 **API**
  - #include <linux/spinlock.h>을 포함
  - spinlock_t 타임의 구조체 사용
  - API
    - spinlock_t my_lock = SPIN_LOCK_UNLOCKED;
      - 컴파일 시점에서 스핀락을 초기화
    - void spin_lock_init(spinlock_t *lock);
      - 실행 시점에서 초기화
    - void spin_lock(spinlock_t *lock);
      - 임계 구역에 접근하기 전에 스핀락 얻기
      - 스핀락을 기다리는 동안에는 인터럽트가 불가능
    - void spin_unlock(spinlock_t *lock);
      - 스핀락을 해제

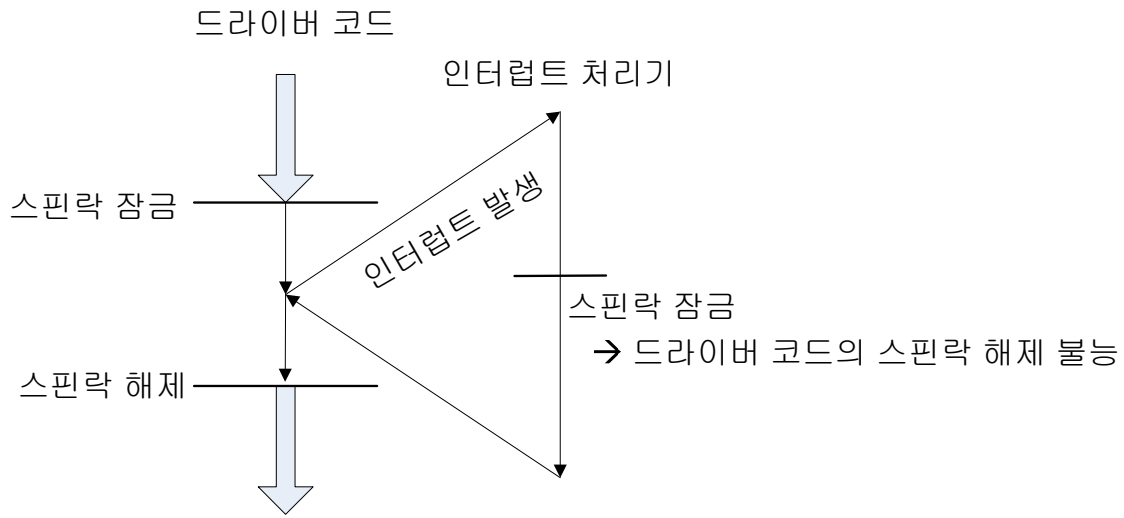
스핀락(spinlocks) (3)

- 스핀락으로 작업할 때 따라야 할 규칙
  - 스핀락을 쥐고 있는 동안에 수행할 코드가 원자적이어야 함
    - 코드가 블록 상태에 빠져서는 안됨(메모리 할당과 같은 블록 상태로 빠지는 함수를 호출 금지)
    - 커널 선점의 경우 스핀락 코드 내부에 선점 기능을 비활성화 하는 코드가 구현 되어 있음



스핀락(spinlocks) (4)

- 스핀락으로 작업할 때 따라야 할 규칙
  - 스핀락을 쥐고 있는 동안에 인터럽트를 비활성



  - 최대한 짧은 시간 동안 스핀락을 획득 해야함

스핀락(spinlocks) (4)

- 기타 스핀락 함수
  - 스핀락 잠금 함수
    - void **spin_lock_irqsave**(spinlock_t *lock, unsigned long flags);
      - 스핀락을 얻기 전에 인터럽트를 비활성화
      - 이전 인터럽트 상태는 flags에 저장
    - void **spin_lock_irq**(spinlock_t *lock);
      - 인터럽트를 비활성화시킬 일이 없을 경우 사용
    - void **spin_lock_bh**(spinlock_t *lock);
      - 하드웨어 인터럽트를 활성화 상태로 유지
      - 소프트웨어 인터럽트에서 락을 얻는 경우 사용
  - 스핀락 해제 함수 - 락을 얻는데 사용한 함수와 대응
    - void **spin_unlock_irqrestore**(spinlock_t *lock, unsigned long flags);
    - spin_lock_irqsave와 spin_unlock_irqrestore는 동일한 함수 내에서 호출 해야함
    - void **spin_unlock_irq**(spinlock_t *lock);
    - void **spin_unlock_bh**(spinlock_t *lock);

스핀락(spinlocks) (5)

- 비블록킹 스핀락 함수
  - int **spin_trylock**(spinlock_t *lock);
  - int **spin_trylock_bh**(spinlock_t *lock);

- 읽기/쓰기 스핀락 - 읽기/쓰기 세마포어와 흡사
  - rwlock_t 구조체 타입 사용

  - 정적 초기화 과정
    - rwlock_t my_rwlock = RW_LOCK_UNLOCKED;
  - 동적 초기화 과정
    - rwlock_init(&my_rwlock);

스핀락(spinlocks) (6)

- 읽기용 잠금 함수
  - void **read_lock**(rwlock_t *lock);
  - void **read_lock_irqsave**(rwlock_t *lock, unsigned long flags);
  - void **read_lock_irq**(rwlock_t *lock);
  - void **read_lock_bh**(rwlock_t *lock);

- 읽기용 해제 함수
  - void **read_unlock**(rwlock_t *lock);
  - void **read_unlock_irqrestore**(rwlock_t *lock, unsigned long flags);
  - void **read_unlock_irq**(rwlock_t *lock);
  - void **read_unlock_bh**(rwlock_t *lock);

스핀락(spinlocks) (7)

- 쓰기용 잠금 함수
  - void **write_lock**(rwlock_t *lock);
  - void **write_lock_irqsave**(rwlock_t *lock, unsigned long flags);
  - void **write_lock_irq**(rwlock_t *lock);
  - void **write_lock_bh**(rwlock_t *lock);
  - int **write_trylock**(rwlock_t *lock);
- 쓰기용 해제 함수
  - void **write_unlock**(rwlock_t *lock);
  - void **write_unlock_irqrestore**(rwlock_t *lock, unsigned long flags);
  - void **write_unlock_irq**(rwlock_t *lock);
  - void **write_unlock_bh**(rwlock_t *lock);

원자 변수 (1)

Atomic Variable

- 공유 자원이 간단한 정수 값인 경우 사용
  - 정수 값을 위해 잠금을 구현하면 배보다 배꼽이 더 큰 격임
- #include <asm/atomic.h> 포함
- atomic_t 구조체 타입 이용
- 실행 속도 빠름
  - 단일 기계어 명령어로 컴파일 하기 때문

관련 함수

- 정적 초기화
  - atomic_t v = ATOMIC_INIT(0);
- 동적 초기화
  - void atomic_set(atomic_t *v, int i);
  - 원자 변수 값을 정수 i로 설정

원자 변수 (2)

- int atomic_read(atomic_t *v);
  - v의 현재 값을 반환
- void atomic_add(int i, atomic_t *v);
  - v가 가리키는 원자 변수에 i를 더함
- void atomic_sub(int i, atomic_t *v);
  - *v에서 i를 뺌
- void atomic_inc(atomic_t *v);
  - 원자 변수를 1 증가
- void atomic_dec(atomic_t *v);
  - 원자 변수를 1 감소
- int atomic_inc_and_test(atomic_t *v);
- int atomic_dec_and_test(atomic_t *v);
- int atomic_sub_and_test(int i, atotmic_t *v);
  - 지정한 연산을 수행하고 결과를 테스트
    - 원자 변수가 0이면 참을 반환
    - 원자 변수가 0이 아니면 거짓을 반환

원자 변수 (3)

- int atomic_add_negative(int i, atomic_t *v);
  - v에 정수 변수 i를 더함
    - 결과가 음수면 반환 값은 참, 그렇지 않으면 거짓
- int atomic_add_return (int i, atomic_t *v);
- int atomic_sub_return(int i, atomic_t *v);
- int atomic_inc_return(atomic_t *v);
- int atomic_dec_return(atomic_t *v);
  - atomic_add와 동작이 같지만, 호출자에게 새로운 원자 변수 값을 반환 하는 것이 다름

비트 연산 (1)

- **bit operations**
  - 원자적으로 개별 비트를 조작해야 할 경우 사용
  - 원자 비트 연산은 매우 빠름
    - 인터럽트를 비활성화 시키지 않고 단일 기계어 명령으로 연산을 수행하기 때문
  - 아키텍처에 의존적
  - #include <asm/bitops.h>를 포함

- 관련 함수
  - void set_bit(nr, void *addr)
    - addr이 가리키는 자료 아이템에서 비트 nr을 1로 설정
  - void clear_bit(nr, void *addr)
    - addr이 가리키는 자료 아이템에서 비트 nr을 0으로 지움
  - void change_bit(nr, void *addr);
    - 비트를 뒤집기
  - test_bit(nr, void *addr);
    - 현재 비트 값을 반환

비트 연산 (2)

- int test_and_set_bit(nr, void *addr);
- int test_and_clear_bit(nr, void *addr);
- int test_and_change_bit(nr, void *addr);
  - 앞서 소개한 함수와 동일, 단 각각 이번 비트 값을 반환

seqlocks – sequence counter lock (1)

- **seqlocks (2.6)**
  - 락 없이 신속하게 공유 자원에 접근하는 방법
  - 보호 자원이 작고, 간단하고, 접근이 잦을 때 적당
  - 쓰기 접근이 드물지만 빨라야 할 때에 적당
  - 읽기 스레드가 자원에 바로 접근 가능하지만, 읽기 스레드 스스로가 쓰기 스레드와 충돌을 확인해야 하며 충돌이 발생하면 접근을 다시 한 번 시도해야 함
  - 포인터가 있는 자료 구조체를 보호하는 데는 적합하지 않음
  - #include <linux/seqlock.h> 포함
  - seqlock_t 구조체 타입 사용

- 관련 함수
  - 정적 초기화
    - seqlock_t lock1 = SEQLOCK_UNLOCKED;
  - 동적 초기화
    - seqlock_init(&lock1);

seqlocks (2)

- 읽기 스레드 코드 형태

```
unsigned int seq;
do {
        seq = read_seqbegin(&the_lock);
        /* 필요한 작업을 한다. */
    } while read_seqretry(&the lock, seq);
```

  - 읽기 접근은 임계 구역에 진입하기 전에 정수 순서 값을 얻음
  - 임계 구역을 빠져 나갈 때에는 순서 값을 현재 값과 비교
  - 값이 일치하지 않으면 읽기 접근을 다시 시도

- 인터럽트 핸들러에서 **seqlock**에 접근 할 경우의 함수
  - unsigned int **read_seqbegin_irqsave**(seqlock_t *lock, unsigned long flags);
  - int **read_seqretry_irqrestore**(seqlock_t *lock, unsigned int seq, unsigned long flags);

seqlocks (3)

- 쓰기 스레드에서 상호 배제 락 잠금/해제
  - void write_seqlock(seqlock_t *lock);
  - void write_sequnlock(seqlock_t *lock);

  - 스핀락으로 쓰기 접근 제어 → 스핀락의 모든 변형 함수 사용
  - 잠금
    - void write_seqlock_irqsave(seqlock_t *lock, unsigned long flags);
    - void write_seqlock_irq(seqlock_t *lock);
    - void write_seqlock_bh(seqlock_t *lock);
  - 풀기
    - void write_sequnlock_irqrestore(seqlock_t *lock, unsigned long flags);
    - void write_sequnlock_irq(seqlock_t *lock);
    - void write_sequnlock_bh(seqlock_t *lock);

RCU

- **RCU(Read-Copy-Update) (2.6)**
  - http://www.rdrop.com/users/paulmck/rclock/intro/rclock_intro.html 에서 whitepaper를 살펴볼 수 있다.
  - <linux/rcupdate.h> 포함
  - spinlock의 비효율성을 개선하기 위해 나옴
  - 스레드 모델에서 다중 reader, 다중 writer 모델을 염두함
  - 수정이 필요한 writer는 해당 데이터를 복사하고 변경한 뒤 reader 들이 읽고 있는 데이터가 정리되면 포인터만의 변경으로 비용을 낮추었다.
  - 포인터 변경은 atomic operation으로 진행된다.
  - RCU를 사용하는 실제 예제로 네트워크 라우팅 테이블에서 찾을 수 있다.

RCU Example

```
struct my_stuff *stuff;

rcu_read_lock(); // RCU를 사용하는 자료구조를 읽는다.
stuff = find_the_stuff(args...);
do_something_with(stuff);
rcu_read_unlock(); // RCU의 해제

...

synchronize_rcu(); // 모든 reader들이 작업을 끝마칠 때까지 기다린 후에 writer에 의해 변경된 내용을 업데이트 한다.
```

락킹 기법 사용시 고려해야 할 점

- 락킹 기법 사용시 고려해야 할 점
  - 한 함수가 락을 얻은 다음에 락을 얻으려는 다른 함수를 호출 하는 경우 데드락 상태가 발생할 수 있다.
  - 관련 함수는 락을 획득했다는 가정 하에 함수를 작성
  - 여러 락을 얻을 때에는 항상 같은 순서로 얻어야 함
  - 작은 단위의 락을 쓸 경우와 큰단위 락을 쓸 경우
  - 락이 필요없는 형태로 데이터구조를 만든다. **Ex)** 원형버퍼 등