11장 스레드 동기화

11311011 김민경

1. 스레드 동기화는 어떤 상황에서 필요한 기술인가?
멀티 스레드 응용프로그램을 작성할 때 꼭 필요한 경우는 어떤 경우일까?
간단한 사례로 설명하라.

스레드 동기화는 멀티 스레드를 사용하는 환경에서 특정 자원에 대한 공유가 이루어지는 경우, 공유 자원을 독점적/배타적으로 처리하여 작업을 완료 후 대기중인 다른 스레드에게 알리는 경우에 필요하다. 예를 들어 계좌를 다루는 프로그램에서 잔액은 공유 자원이 되며 잔액을 이용하여 행해지는 작업은 이체, 출금 등 여러형태가 있다. 동기화는 하나의 스레드가 작업을 실행하면 다른 스레드의 접근을 막는 개념이다. 그런데 계좌 프로그램에서 동기화가 사용되지 않는다면 문제가 발생할 수 있다. 만약 이체 작업을 진행하고 있는 스레드가 이체 결과를 통해 잔액을 조정하는 작업을 행하고 있을 때는 동시에 출금 작업이 진행되어서는 안된다. 이체 작업이 완료되지 않은 상태에서 출금 작업이 개입하여 잔액을 조정하게 되면 계좌에는 원치 않는 결과가 발생하기 때문이다.

2. 상호배제, race condition, 임계 구역을 간단히 설명하고 이들 3개의 단어를 연결하여 말이 되는 문장 하나를 작성해보라.

1) 상호배제 : 하나의 프로세스가 임계구역을 점유하고 있는 경우 다른 프로세스가 임계구역에서 실행될 수 없도록 배제하여 오직 한 스레드만 배타적, 독점적으로 사용되도록 관리하는 것.

2) 경쟁 상황 : 동기화가 적용되지 않는 공유자원에 두 개 이상의 프로세스/스레드들이 접근하려는 경쟁적 상황.

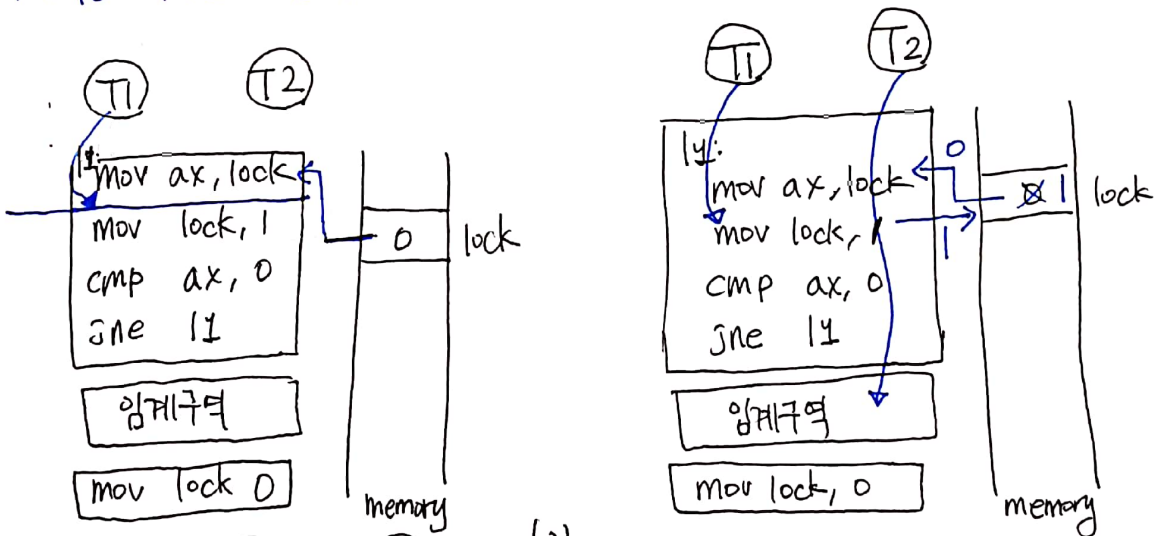
3) 임계구역 : 다수의 프로세스들이 공유하는 공간/코드

멀티 스레드 응용 프로그램에서 각 프로세스/스레드들은 임계구역을 가지고 있으며, 임계 구역에서는 race condition이 발생하여 의도치 않은 결과를 초래할 수 있기에 race condition에 의한 악영향을 예방하고자 한번에 하나의 스레드만 배타적, 독점적으로 관리되도록 상호 배제 조건이 지켜져야 한다.

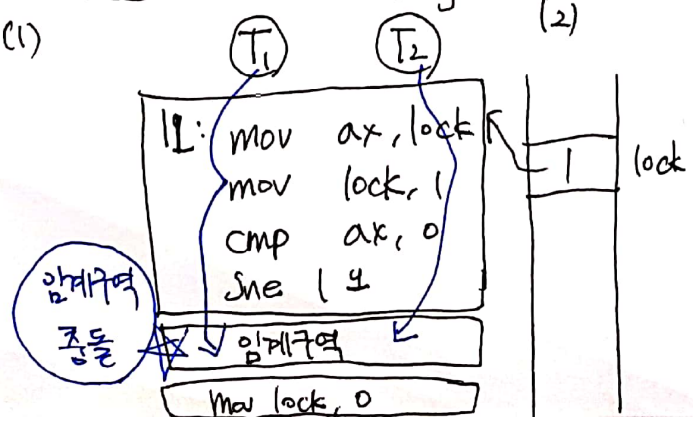
3. 상호배제를 위한 하드웨어 솔루션으로 인터럽트를 어떤 식으로 이용할 수 있는가? 인터럽트를 이용한 방법은 완벽한 해결책이 되지 못한다. 그 이유는 무엇인가?

입출력 장치나 타이머가 인터럽트를 걸 수 있도록 허용해놓고, 임계구역을 실행 중일 때 인터럽트는 무시하고 임계구역이 끝난 다음에 처리하도록 하면 된다.
인터럽트 플래그를 이용하여 cli, sti 와 같은 명령어를 사용하면서 인터럽트 요청을 일시적으로 무시하도록 한다. cli 명령어는 CPU 에 인터럽트 플래그를 0으로 리셋시켜 인터럽트가 발생해도 CPU가 인터럽트 서비스 루틴으로 점프하지 않고 현재 작업을 계속 수행하게 한다. sti 명령어는 CPU의 인터럽트 플래그를 1로 설정하여 인터럽트가 발생하면 하던 일을 중단하고 인터럽트 서비스루틴을 실행하게 한다. 하지만, 이와 같은 방식은 다중 CPU 시스템에서는 활용할 수 없다.
한 스레드가 인터럽트 cli 와 같은 요청을 하여 인터럽트 서비스를 금지하였다고 하더라도 다른 코어의 인터럽트 서비스까지 금지시킬 수는 없기 때문이다.

4. 이 코드들이 임계 구역으로 진입할 때 상호배제가 잘 이루어지지 않는다. 구체적인 이유를 그림과 함께 설명하라.



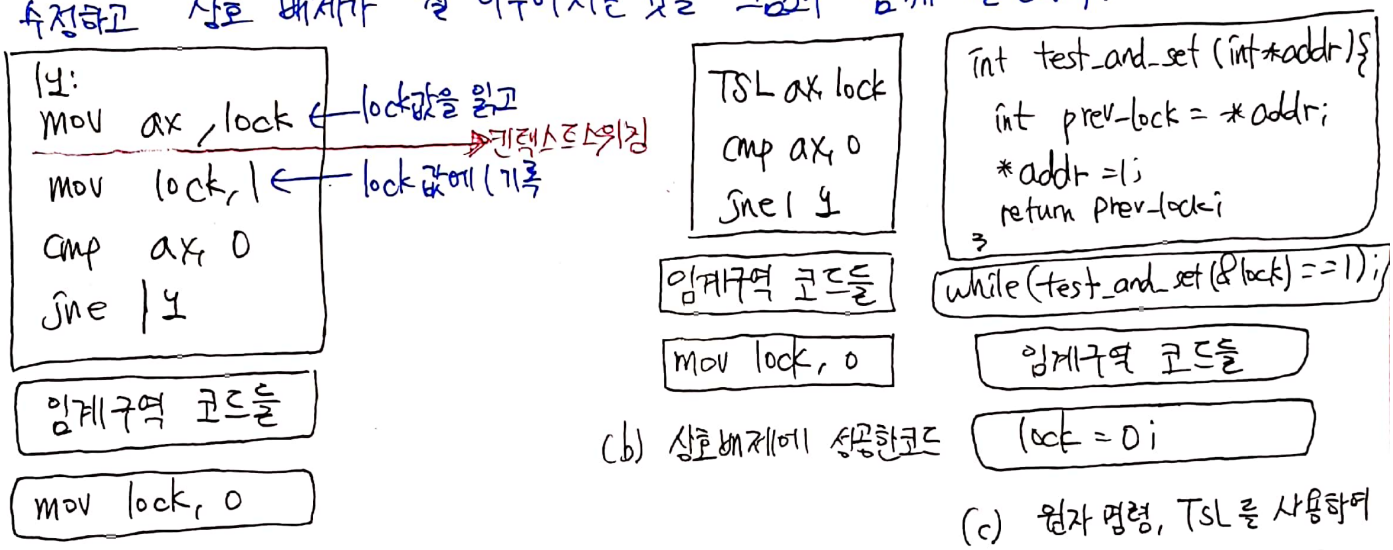
(1)



(2)

위 코드들이 임계구역으로 진입할 때 상호배제는 잘 이루어지지 않는다. 예를 들어 T1 스레드와 T2 스레드가 임계구역에 진입할 때를 생각해보자. 만약 T1 이 mov ax, lock 코드를 실행하면 lock 값은 0 이 된다. 그런데 만약 T1 이 mov lock, 1 을 실행하기 전에 T2로 컨텍스트 스위칭이 일어나게 되면 T2 역시 똑같이 lock 값이 0 이된다. 즉, T1 과 T2 모두 lock 값이 0 이 되었기 때문에 mov lock, 1 을 실행시킬 경우에 T1 와 T2 모두 임계구역에 들어갈수 있게 되므로 상호 배제에 실패하게 된다.

5. 근본적인 문제점을 한 줄로 설명하고, 원자 명령을 이용하여 위의 코드를 수정하고 상호 배제가 잘 이루어지는 것을 그림과 함께 설명하라.



mov ax, lock 과 mov lock, 1 이 따로 분리되어 존재하기 때문에 문제 4번과 같이 상호 배제가 잘 이루어지지 않는다는 근본적인 문제점이 있다. 따라서 위 두 명령어 사이에 컨텍스트 스위칭이 일어나지 않도록 이 두 명령을 하나의 명령으로 만드는 원자 명령이 필요하다. 즉 TSL ax, lock 이라는 새로운 원자 명령어를 사용하여 잘못된 상호 배제를 막아야 한다.

6. 라이브러리와 시스템 호출로 멀티스레드의 동기화를 구현한 3가지 기법을 간단히 설명하라.

뮤텍스는 잠김/열림 중 한 상태를 가지는 락 변수를 이용하여 오직 한 스레드만이 자원을 배타적으로 사용하도록 하는 기법이다. 스레드가 락에 접근하게 되면, 스레드는 블록 상태로 대기 큐에 삽입 되어 잠들게 된다. 그리고 락을 해제하는 경우에 대기 큐에 있는 스레드 중 하나를 깨워 준비 상태로 만든다. 따라서 blocking lock 기법 혹은 sleep-waiting lock 기법이라고도 불린다.

스핀락은 lock 연산에서 락이 잠겨있을 때 블록되지 않고 락이 풀릴 때까지 락 변수를 검사하는 코드를 실행한다. 따라서 busy-waiting lock 이라고도 불린다. 스핀락은 임계 구역 코드가 짧아서 빨리 락이 풀리는 응용에는 매우 효과적이다. 하지만 단일 CPU 를 가진 운영체제에서는 비효율적인 방법이다.

세마포는 스레드가 동시에 사용할 수 있는 하나의 자원에 대해 스레드가 공유하도록 관리하는 동기화 프로그래밍 기법이며 프로그래밍 개념이다. 여기서 하나의 자원은 여러개의 인스턴스들을 포함한다. 대기큐는 busy-waiting 의 경우 사용하지 않고, sleep-wait 의 경우 사용한다. 그리고 counter 변수를 사용하여 사용 가능한 자원의 개수를 나타내어 가감시키는 역할을 한다. 이때 P/V 연산을 이용하여 자원 요청시, 자원 반환 시 실행되는 연산을 표현한다.

7. 뮤텍스와 스핀락은 각각 어떤 상황에서 적합할지 간단히 설명하라.

자원에 대한 임계구역의 실행 시간에 따라 적합도가 다르다. 임계구역의 실행시간이 긴 경우 프로세스의 대기 시간이 길어지므로 대기하는 동안 컨텍스트 스위칭을 통한 다른 작업을 처리할 수 있는 뮤텍스가 효율적이다. 그래서 보통 사용자 모드, 사용자 응용프로그램 에서 많이 뮤텍스를 사용한다. 반면, 실행시간이 짧은 경우 뮤텍스의 컨텍스트 스위칭이 빈번하게 발생하여 시스템에 오버헤드가 발생할 수 있다. 스핀락은 대기 시간이 짧은 경우 프로세스가 CPU를 점유하고 있기 때문에 컨텍스트 스위칭이 발생하지 않고 락이 해제되는 순간 빠르게 작업을 효율적으로 처리할 수 있다.

커널 코드나 인터럽트 서비스 루틴과 같은 경우는 빠른 시간 내에 실행되어야 하므로 코드가 짧고, 실행 중에 블록되어 잠들 자도록 하면 안되기 때문에 스핀락을 주로 사용한다. 반면 단일 CPU를 가진 운영체제에서 스핀락은 비효율적이다. 의미없는 기다림으로 CPU를 계속해서 사용하기 때문에 CPU의 낭비가 심하다. 따라서 멀티 코어 CPU나 다중 CPU 에서 주로 사용된다.

8. N명만 동시에 사용할 수 있는 데이터베이스가 있다. 데이터베이스에 접근을 다루기 위해 뮤텍스, 스핀락, 세마포 중 어떤 것이 적합한가? 적합하지 않는것은 왜 적합하지 않는가?

세마포어의 방식이 문제에서 제시한 환경에 적합하다. n명의 사용자가 동시에 작업을 진행할 수 있는 환경이 되어야 하므로 데이터 베이스에 접근하는 n개의 프로세서에 대해 접근이 허용되어야 한다. 세마포어 방식은 공유 자원에 대한 접근을 n으로 지정할 수 있다. 그리고 만약 공유 자원에 접근한 사용자 수가 n 초과일 경우 대기 큐에서 대기하므로 스핀락과 같은 오버헤드를 방지할 수 있기 때문에 문제에서 요구하는 환경에 적합한 방법이다.
반면, 뮤텍스는 적절하지 않다. 뮤텍스는 하나의 락만 허용하기 때문에 한 명의 사용자가 데이터베이스를 점유하고 있는 상황일 때 n-1 명의 사용자들은 데이터베이스에 락이 걸려 접근할 수 없게 된다. 따라서 동시에 n명의 사용자 접근을 허용하기 위한 목적에 적절하지 않다.
또한, 스핀락 방식을 적용할 경우 한 명의 유저가 데이터베이스 영역에 진입시 데이터베이스는 락이 걸리게 된다. 나머지 n-1 명의 프로세스는 임계 영역에 접근하지 못한 busy-waiting 상태로 무한 loop를 진행한다. 뮤텍스와 마찬가지로 빌렬성을 보장하지 못하며, n-1명에 대한 busy-waiting 진행은 시스템에 과부하를 주어 데이터베이스 서버에 좋지 않은 영향을 끼칠 수 있기 때문에 스핀락은 적절한 방법은 아니다.

9. 우선순위 역전은 어떤 조건과 상황에서 발생하는지 서술하고 우선순위 역전이 일어나는 과정을 사례로 들어 보아라.

우선순위 역전은 스레드 동기화로 인해 우선순위가 높은 스레드가 순위가 낮은 스레드보다 늦게 스케줄링 되는 문제를 말한다. 여기서 우선순위 역전은 공유변수를 사용하는 스레드와 사용하지 않는 스레드가 혼재할 때 발생한다. 예를 들어서 T1, T2, T3 스레드가 있다고 가정해보자. 차례대로 T1이 가장 낮은 우선 순위, T2는 중간 우선순위, T3은 가장 높은 우선순위의 스레드이다. 만약 T1이 먼저 도착하여 실행되고 T1이 세마포의 P 연산을 통해 자원을 먼저 할당 받아서 공유변수를 사용하고 있다고 가정해보자. 그리고 T1이 실행되는 동안 T3이 도착하였다. T3은 T1 보다 우선순위가 높기 때문에 T1을 중단시키고 T3을 실행시키고 P 연산을 실행하였으나 T1이 소유하고 있으므로 자원을 대기하는 sleep 상태가 된다. 그런데 이때 T3 보다 우선순위가 낮은 T2가 도착한 경우 공유변수를 사용하지 않는 T2가 T3 보다 먼저 실행되는 우선순위 역전현상이 발생한다.

이와 같은 문제를 해결하기 위해서 우선순위 올림과 우선순위 상속 기법을 사용한다. 우선순위를 일시적으로 미리 정해진 우선순위보다 높여서 다른 스레드에 의해 선점되지 않고 공유자원에 대한 엑세스가 끝날 때 본래의 우선순위로 되돌리도록 한다. 혹은 우선순위를 상속시켜서 낮은 순위의 스레드의 우선순위를 요청 스레드보다 높게 변경하여 실행 중이던 공유자원을 계속 실행시키도록 하는 방법이 있다. 하지만 이 두 방법은 모두 여러가지 오버헤드와 구현에 따른 어려움을 가지고 있다.

10. 1에서 4000 까지 더할 때, 전역 변수 sum에 4개의 스레드가 동시에 실행하여 더하는 5장의 2번째 소스는 잘못된 코드이다. 이것을 뮤텍스를 이용하는 코드로 각각 완성하여 코드와 실행결과를 캡처하여 이미지로 손글씨와 함께 제출하라.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
void * worker (void *param);
pthread_mutex_t locki                    → <mutex.c>
int sum=0;
int main() {
    pthread_t tid[4];
    pthread_attr_t attr[4];
    int i;
    for (i=0; i<=3; i++) {
        pthread_attr_init (&attr[i]);
    }
    pthread_mutex_init (&lock, NULL);
    pthread_create (&tid[0], &attr[0], worker, "1");
    pthread_create (&tid[1], &attr[1], worker, "1001");
    pthread_create (&tid[2], &attr[2], worker, "2001");
    pthread_create (&tid[3], &attr[3], worker, "3001");
    printf ("Thread create\n");
    void * status = NULL;
    for (i=0; i<3; i++) { pthread_join (tid[i], &status); }
    printf ("Thread terminated\n");
    pthread_mutex_destroy (&lock);
    return 0;
}
void *worker (void *param) {
    int to = atoi(param);
    int i;
    printf ("start! sum = %d \n", sum);
    for (i=to; i<to+1000; i++) sum += i;
    pthread_mutex_unlock (&lock);
```

```
pthread_mutex_lock (&lock);
printf("End! Sum = %d \n", sum); }
```

<spin.c>

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void * worker (void * param);
pthread_spinlock_t lock;
int sum=0;
int main(){
    pthread_t tid[4];
    pthread_attr_t attr[4];
    int i;
    for (i=0; i<=3; i++){
        pthread_attr_init (&attr[i]);
    }
    pthread_spin_init (&lock, PTHREAD_PROCESS_PRIVATE);
    pthread_create (&tid[0], &attr[0], worker, "1");
    pthread_create (&tid[1], &attr[1], worker, "1001");
    pthread_create (&tid[2], &attr[2], worker, "2001");
    pthread_create (&tid[3], &attr[3], worker, "3001");
    printf("Thread create\n");
    void *status = NULL;
    for (i=0; i<=3; i++) { pthread_join (tid[i], &status); }
    printf("Thread terminated\n");
    pthread_spin_destroy (&lock);
    return 0;
}

void * worker(void *param){
    int to = atol(param);
    int i;
    printf("Start! Sum=%d\n", sum);
    pthread_spin_lock (&lock);
    for (i=to; i<to+1000; i++) sum+=i;
    pthread_spin_unlock (&lock);
    printf ("End! Sum=%d\n", sum);
}
```