3.1. Atomic bit operations & spin lock

Multithread 프로그램에서 가장 큰 문제가 동시에 실행하는 여러개의 thread간의 동기화입니다. UP의 경우 기본적으로 어느 시점에 control을 다른 thread로 넘길 지를 알 수 있으므로 프로세서의 별다른 지원없이도 동기화가 가능하지만 MP에선 동시에 작동하는 여러개의 프로세서들을 동기화시키려면 특수한 기능을 제공해야합니다. 기본적으론 메모리의 어떤 값을 읽어서 조건을 확인하고 그 값에 어떤 변화를 주는 작업을 atomic하게 수행할 수 있어야합니다. 보통 test and set bit이나 exchange등을 atomic하게 수행하는 기능을 제공하게 되는데 ix86에선 두가지 모두 지원됩니다. (ix86에선 instruction에 LOCK prefix를 붙이면 대부분의 instruction들을 atomic하게 수행할 수 있습니다.)

이 절에선 atomic test and set bit을 이용해 동기화의 기본적인 구성요소인 spin lock을 만들어보겠습니다. Pthread를 사용해 몇개의 thread들 사이에서 spin을 이용한 동기화를 해볼텐데, 각각의 thread들이 독립된 processor에서 작동하는 것이 아니라 time slice되어 작동하기 때문에 spin을 사용하는 것은 좋은 생각은 아닙니다. 예를 들기위한 것이라고 생각하시기 바랍니다.

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

static __inline__ int
test_and_set_bit(int nr, volatile void *addr)
{
        int oldbit;

        __asm__ __volatile__(
                "lock                   ₩n₩t"
                "btsl   %2, %1          ₩n₩t"
                "sbbl   %0, %0          "
                : "=r" (oldbit), "=m" (*(unsigned *)addr)
                : "Ir" (nr));
        return oldbit;
}

typedef unsigned spin_t;

static __inline__ void
spin_lock(spin_t *spin)
{
        if (test_and_set_bit(0, spin))
        while (*(__volatile__ spin_t *)spin) ;
}

static __inline__ void
spin_unlock(spin_t *spin)
{
        *(__volatile__ spin_t *)spin = 0;
}

spin_t spin = 0;

static __inline__ void
waste_time(void) {
        time_t ltime;

        ltime = time(NULL);
```

```
        while (time(NULL) == ltime) ;
}

#define pp(fmt, arg...) printf("thread %d : "fmt, thread_no , ##arg)

static void *
thread_func(void *_arg)
{
        int thread_no = (int)_arg;

        while (1) {
        pp("wasting time₩n");
        waste_time();
        pp("entering critical section₩n");
        pp("now in critical section, wasting time₩n");
        waste_time();
        pp("leaving critical section₩n");
        }

        return NULL;
}

static void *
thread_sfunc(void *_arg)
{
        int thread_no = (int)_arg;
        time_t ltime;

        while (1) {
                pp("wasting time₩n");
                waste_time();
                pp("entering critical section₩n");
                spin_lock(&spin);
                pp("now in critical section, wasting time₩n");
                waste_time();
                pp("leaving critical section₩n");
                spin_unlock(&spin);
        }

        return NULL;
}

int
main(void)
{
        pthread_t thread0, thread1, thread2, thread3;

        pthread_create(&thread0, NULL, thread_func, (void *)0);
        pthread_create(&thread1, NULL, thread_func, (void *)1);
        pthread_create(&thread2, NULL, thread_sfunc, (void *)2);
        pthread_create(&thread3, NULL, thread_sfunc, (void *)3);

        pthread_join(thread0, NULL);
        pthread_join(thread1, NULL);
        pthread_join(thread2, NULL);
        pthread_join(thread3, NULL);

        return 0;
}
```

4개의 thread가 시간보내기, critical section들어가기, 시간보내기, critical section나오기의 과정을 반복합니다. 0번과 1번 thread는 아무런 동기화도 하지 않고 2, 3은 spin을 이용해 critical section을 보호합니다.

우선 test_and_set_bit 함수를 살펴보도록 하겠습니다.

```
        lock; btsl %2, %1
```

```
lock; btsl %2, %1
```

Atomic하게 *addr의 nr번째 bit을 test and set합니다. Test의 결과는 carry flag에 기록이됩니다.

```
sbbl %0, %0
```

위의 btsl에서 CF기록된 test결과를 %0에 저장합니다. sbbl은 subtraction with carry로 위처럼 같은 두 수로 실행하면 carry flag과 같은 논리값이 operand에 저장됩니다.

Output, input 지정에선 nr의 constrait가 "Ir"인 점을 제외하면 별다른 점은 없습니다. nr이 한 word안에서의 bit offset이므로 immediate일 때 0~31사이로 제한을 둔 것이고, 레지스터 operand일 때는 컴파일타임에 확인할 수 없기 때문에 그냥 'r'을 constraint로 준 것입니다.

test_and_set_bit이 있으면 spin의 구현은 간단한데요. test_and_set_bit의 결과값이 0일 때까지 반복적으로 수행하면 됩니다. 위의 프로그램에서 기다리는 부분을 while로 처리한 것은 lock; btsl보다 보통의 memory read가 bus에 부담을 덜 주기 때문입니다. while의 조건에서 __volatile__로 캐스팅후 사용하는 건 gcc가 레지스터에 spin의 값을 읽은 후 그 값을 계속 test하는 것을 막기 위해서 입니다. Gcc에게 이 값은 현재 프로그램의 진행과 관계없이 바뀔 수 있다는 것을 알려주는 것입니다.

Unlock은 그냥 spin의 값을 0으로 하면 됩니다. 역시 memory에 직접쓰도록 하기위해 __volatile__로 캐스팅후 사용하는 것을 볼 수 있습니다. 그냥 spin을 __volatile__로 지정해주어도 됩니다.

Inline assembly는 단순하므로 컴파일된 assembly는 생략하고 실행 결과를 보겠습니다.

```
thread 0 : wasting time
thread 1 : wasting time
thread 2 : wasting time
thread 3 : wasting time
thread 0 : entering critical section
thread 0 : now in critical section, wasting time
thread 3 : entering critical section
thread 3 : now in critical section, wasting time
thread 2 : entering critical section
thread 1 : entering critical section
thread 1 : now in critical section, wasting time
thread 1 : leaving critical section
thread 1 : wasting time
thread 0 : leaving critical section
thread 0 : wasting time
thread 3 : leaving critical section
thread 3 : wasting time
thread 2 : now in critical section, wasting time
thread 2 : leaving critical section
thread 2 : wasting time
thread 1 : entering critical section
thread 1 : now in critical section, wasting time
thread 0 : entering critical section
thread 0 : now in critical section, wasting time
thread 3 : entering critical section
thread 3 : now in critical section, wasting time
thread 2 : entering critical section
thread 1 : leaving critical section
thread 1 : wasting time
thread 0 : leaving critical section
thread 0 : wasting time
thread 3 : leaving critical section
thread 3 : wasting time
thread 2 : now in critical section, wasting time
```

0과 1의 critical section은 겹치지만 2와 3은 겹치지 않음을 알 수 있습니다.

이전 Applications | 처음으로 | 위로 | 다음 Function call with stack switch