Linux Threads Programming

Matteo Dell'Omodarme matt@martine2.difi.unipi.it 지음

임종균 hermes44@secsm.org 옮김

1999년 12월 7일

목 차

제 1 절 약간의 이론 **1**

1.1 소개 . . . 1

1.2 쓰레드란 무엇인가? . . . 1

1.3 원자성(atomicity)과 휘발성(volatility) . . . 2

1.4 Lock (잠금) . . . 2

제 2 절 그리고 약간의 실제 **2**

2.1 pthread.h 헤더 . . . 2

2.2 lock의 초기화 . . . 3

2.3 쓰레드 생성하기 . . . 3

2.4 깨끗한 종료 . . . 3

2.5 쓰레드 루틴에 데이터 전달하기 . . . 4

2.6 병렬 코드의 예 . . . 4

제 1 절 약간의 이론

1.1 소개

LinuxThreads는 다중 쓰레드 프로그래밍을 위한 리눅스 라이브러리이다. LinuxThreads는 커널 수준의 쓰레드를 제공한다; 쓰레드들은 clone() 시스템콜(system call)에 의해 만들어지고 모든 스케줄링은 커널에서 이루어진다. Posix 1003.1c API를 구현하였고 커널 2.0.0이상의 커널과 적절한 C 라이브러리를 가지고 있는 어떠한 리눅스 시스템에 동작한다.

1.2 쓰레드란 무엇인가?

쓰레드는 프로그램을 통한 제어의 순차적인 흐름이다. 그래서 다중 쓰레드 프로그래밍은 여러 제어 쓰레드가 한 프로그램에서 동시에 수행하는 병렬 프로그래밍의 한 형태이다.

다중 쓰레드 프로그래밍은 모든 쓰레드가 같은 메모리 공간을 (그리고 파일 디스크립터와 같은 일부 시스템 자원들을) 공유하는 유닉스 스타일의 다중 프로세싱과는 다르다. 대신에 유닉스의 프로세스와 같이 자신만의 고유 메모리상에 동작한다. 그래서 한 프로세스의 두 쓰레드 사이의 문맥 교환(context switch)는 두 프로세스 사이의 문맥 교환보다 굉장히 수월하다.

쓰레드를 사용하는 두 가지 주요한 이유가 있다:

- 어떤 프로그램들은 하나의 제어 흐름 보다는 서로 통신하는 여러 쓰레드로 작성될 때만 최고의 성능을 낼 수 있다. (즉, 서버들)
- 다중 프로세서 시스템에서, 쓰레드들은 여러 프로세서상에서 병렬적으로 수행될 수 있다. 이는 한 프로그램이 다른 프로세서에 작업을 분배할 수 있게 해준다. 이런 프로그램은 한 번에 한 CPU만을 사용할 수 있는 단일 쓰레드 프로그램 보다 훨씬 더 빠르다.

1.3 원자성(atomicity)과 휘발성(volatility)

쓰레드에 의해 공유되는 메모리를 접근하는 데는 더 주의가 필요하다. 병렬 프로그램은 일반적인 지역 메모리처럼 공유 메모리 객체를 접근할 수 없기 때문이다.

원자성(atomicity)는 어떤 객체에 대한 연산은 분리될 수 없는, 인터럽트 되는 않는 과정으로 이루어져야 되다는 개념을 말한다. 공유 메모리상의 데이터에 대한 연산은 원자적으로 이루어질 수 없다. 게다가 GCC 컴파일러는 종종 레지스터에 공유 변수들의 값을 버퍼링하는 최적화를 수행할 것이다. 이렇게 메모리 연산을 피하는 것이라도 모든 프로세서가 공유 데이터의 값이 변경되는 것은 알 수 있어야만 한다.

레지스터에 공유 메모리의 값을 버퍼링하는 GCC의 최적화를 막기 위해 공유 메모리 상의 모든 객체는 volatile 속성의 타입으로 선언되어야 한다. 한 word의 volatile 객체를 읽고 쓸는 것은 원자적으로 이루어 지기 때문이다.

1.4 Lock (잠금)

결과값을 읽어오기 저장하는 것은 독립된 메모리 연산이다: ++i은 항상 공유 메모리 상의 i을 1만큼 증가시키지는 않는다. 두 연산 사이에 다른 프로세서가 i을 접근할 수 있기 때문이다. 그래서 두 프로세스가 둘 다 ++i을 수행한다면 2가 아닌 1만을 증가될 수도 있다.

그래서 한 쓰레드가 변수의 값을 바꾸는 동안은 다른 쓰레드가 그 변수에 대한 작업을 할 수 없게 하는 시스템 콜이 필요하다. 이는 아래 설명한 lock 방법으로 구현된다. 공유 변수의 값을 바꾸는 루틴을 수행하는 두 쓰레드가 있다고 가정을 하자. 그 루틴이 정확한 결과를 얻기 위해서는 다음과 같이 해야 한다.

- i 변수에 대해 lock을 건다.
- 잠긴 변수의 값을 수정한다.
- lock을 제거한다.

한 변수에 대한 lock이 걸릴 때 그 lock을 건 쓰레드만이 그 값을 바꿀 수 있다. 잠근 때문에 다른 쓰레드들은 블럭이 될 것이다. 한 변수에 대해서 는 한 번에 하나의 lock만이 허용되기 때문이다. 프첫번째 쓰레드가 lock을 제거할 때만 두번째 쓰레드가 lock을 걸 수 있다. 그 결과 공유 변수를 이용하는 것은 다른 프로세서들의 활동을 느리게 할지도 모른다. 하지만 일반적인 참조는 지역 캐시를 이용한다.

제 2 절 그리고 약간의 실제

2.1 pthread.h 헤더

LinuxThreads가 제공하는 것은 쓰레드 루틴들의 프로토타입을 선언하는 `/usr/include/pthread.h` 헤더를 통해서 이용 가능하다.

다중 쓰레드 프로그램의 작성은 기본적으로 두 단계의 과정이다:

- 공유 변수들에 lock을 걸고 쓰레드를 만들기 위한 pthread 루틴들을 사용한다.

- 쓰레드 서브 루틴에 넘겨야 할 모른 인자들을 포함하는 구조체를 만든다.

몇 가지 기본적인 pthread.h의 루틴들을 간단히 설명하면서 이 두 단계를 실펴보자.

2.2 lock의 초기화

해야만 하는 첫번째 행동들 중의 하나는 모든 lock들을 초기화하는 것이다. POSIX lock들은 pthread_mutex_t 타입의 변수로 선언된다; 각 lock을 초기화하기 위허 다음 루틴을 호출할 필요가 있다:

```
int pthread_mutex_init(pthread_mutex_t *mutex,
const pthread_mutexattr_t *mutexattr);
```

묶어서 보면:

```
#include <pthread.h>
...
        pthread_mutex_t lock;
        pthread_mutex_init(&lock, NULL);
...
```

pthread_mutex_init 함수는 mutex 인자가 가르키는 mutex 객체를 mutexattr에 의해 명시된 mutex 속성에 따라 초기화를 한다. mutexattr의 NULL이면, 디폴트 속성이 사용된다.

계속해서 이 초기화된 lock들을 어떻게 사용하는지 보겠다.

2.3 쓰레드 생성하기

POSIX는 각 쓰레드를 나타내기 위해 사용자가 pthread_t 타입의 변수를 선언하도록 한다. 다음 호출로 쓰레드가 생성된다:

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr, void
*(*start_routine)(void *), void *arg);
```

성공한다면 새로이 생성된 쓰레드의 id가 thread 인자가 지시한 영역에 저장이 되고 0인 리턴된다. 에러가 발생하면 0이 아닌 값이 리턴된다.

f() 루틴을 수행하는 쓰레드를 만들고 f()에 arg 변수를 가르키는 포인터를 넘기기 위해서는 다음과 같이 한다:

```
#include <pthread.h>
...
        pthread_t thread;
        pthread_create(&thread, NULL, f, &arg).
...
```

f() 루빈은 다음과 같은 프로토타입을 가져야 한다:

```
void *f(void *arg);
```

2.4 깨끗한 종료

마지막 단계로 f() 루틴의 결과를 접근하기 전에 만든 모든 쓰레드가 종료 할 때까지 기다려야 한다. 다음을 호출한다:

```
int pthread_join(pthread_t th, void **thread_return);
```

th 가 가르키는 쓰레드가 종료할 때까지 위의 함수를 호출한 쓰레드의 수행을 멈춘다. 만약 thread_return이 NULL이니면 th의 리턴값은 thread_return이 가리키는 영역에 저장된다.

2.5 쓰레드 루틴에 데이터 전달하기

호출한 루틴의 정보를 쓰레드 루틴에 넘기는 두 가지 방법이 있다:

- 전역 변수
- 구조체

두번째 것이 코드의 모듈성(modularity)을 보전하는 데 가장 좋은 선택이다. 구제체는 세 가지 단계의 정보를 포함해야 한다; 첫째로 공유 변수들과 lock들에 관한 정보, 두번째로 루틴에서 필요로 하는 모든 데이터에 대한 정보, 세번째로 쓰레드를 구분해주는 id와 쓰레드가 이용할 수 있는 CPU의 수에 대한 정보 (런타임에 이 정보를 제공하는 것이 더 쉽다). 구조체의 첫번째 요소을 살펴보자; 넘겨진 정보는 모든 쓰레드들 사이의 공유되는 것이어야 한다. 그래서 필요한 변수들과 lock들의 포인터를 사용 해야 한다. double 타입의 공유 변수 var와 그에 대한 lock을 넘기기 위해 구조체는 두 멤버 변수를 가져야만 한다:

```
double volatile *var;
pthread_mutex_t *var_lock;
```

volatile 속성의 사용 위치에 주목하라. 이는 포인터 자체가 아니라 var가 volatile임을 나타낸다.

2.6 병렬 코드의 예

쓰레드를 이용하여 쉽게 병렬화를 할 수 있는 프로그램의 예는 두 벡터의 스칼라코곱을 계산이다. 주석을 붙인 코드를 제시한다.

```
/* 컴파일 하려면 gcc  -D_REENTRANT -lpthread */

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

/* 알맞은 구조체 선언 */
typedef struct {
        double volatile *p_s;      /* 스칼라 곱의 공유 변수 */
        pthread_mutex_t *p_s_lock; /* 변수 s의 lock */
        int n;                     /* 쓰레드의 수 */
        int nproc;                 /* 이용할 수 있는 프로세서의 수 */
        double *x;                 /* 첫번째 벡터의 데이터 */
        double *y;                 /* 두번째 벡터의 데이터 */
        int l;                     /* 벡터의 길이 */
} DATA;
```

```
void *SMP_scalprod(void *arg)
{
        register double localsum;
        long i;
        DATA D = *(DATA *)arg;

        localsum = 0.0;

        /* 각 쓰레드는 i = D.n에서 부터 스칼라 곱을 시작한다.
           D.n = 1, 2, ...
           D.nproc 값을 갖는다. 정확히 D.nproc개의 쓰레드가 있기
           때문에 i의 증가 같은 D.nproc이다. */

        for(i = D.n; i < D.l; i += D.nproc)
        localsum += D.x[i]*D.y[i];

        /* s에 대한 lock을 건다 ... */
        pthread_mutex_lock(D.p_s_lock);

        /* ... s의 값을 바꾼다. ... */
        *(D.p_s) += localsum;

        /* ... 그리고 lock를 제거한다. */
        pthread_mutex_unlock(D.p_s_lock);

        return NULL;
}

#define L 9     /* 벡터의 차원 */

int main(int argc, char **argv)
{
        pthread_t *thread;
        void *retval;
        int cpu, i;
        DATA *A;
        volatile double s = 0; /* 공유 변수 */
        pthread_mutex_t s_lock;
        double x[L], y[L];

        if (argc != 2) {
                printf("usage: %s <number of CPU>\n", argv[0]);
                exit(1);
        }

        cpu = atoi(argv[1]);
        thread = (pthread_t *) calloc(cpu, sizeof(pthread_t));
        A = (DATA *) calloc(cpu, sizeof(DATA));


        for (i = 0; i < L; i++)
        x[i] = y[i] = i;

        /* lock 변수를 초기화한다. */
        pthread_mutex_init(&s_lock, NULL);
```

```
        for (i = 0; i < cpu; i++) {
                /* 구조체를 초기화한다. */
                A[i].n = i; /* 쓰레드의 수 */
                A[i].x = x;
                A[i].y = y;
                A[i].l = L;
                A[i].nproc = cpu; /* CPU의 수 */
                A[i].p_s = &s;
                A[i].p_s_lock = &s_lock;

                if (pthread_create(&thread[i], NULL, SMP_scalprod,
                    &A[i])) {
                        fprintf(stderr, "%s: cannot make thread\n",
                                argv[0]);
                        exit(1);
                }
        }

        for (i = 0; i < cpu; i++) {
                if (pthread_join(thread[i], &retval)) {
                        fprintf(stderr, "%s: cannot join thread\n",
                                argv[0]);
                        exit(1);
                }
        }

        printf("s = %f\n", s);
        exit(0);
}
```

Copyright © 1999, Matteo Dell'Omodarme

Published in Issue 48 of Linux Gazette, December 1999