Spring 2019 Operating Systems
Lab 07: Pthread

0. 과제 수행 전 유의사항
프로그래밍은 자기 손으로 작성해야 한다. 아이디어를 서로 공유하고 도움을 받는 것은 좋으나, 다른 사람의 코드를 그대로 가져다 쓰는 것은 자신에게도 손해이고 다른 사람에게 피해를 끼치는 결과를 낳게되므로 허용하지 않는다. 과제 체크 시 두 사람의 코드가 유사하면, 두 사람 모두에게 과제 점수를 부여할 수 없으니 유의바란다.

1. 소개
리눅스 쓰레드는 다중 쓰레드 프로그래밍을 위한 커널 수준의 쓰레드를 제공하는 리눅스 라이브러리다. 쓰레드들은 clone( ) 시스템콜에 의해 만들어지고 모든 스케줄링은 커널에 의해서 이루어진다. 쓰레드는 프로그램을 통한 제어의 순차적인 흐름이다. 따라서, 다중 쓰레드 프로그래밍은 여러 제어 쓰레드가 한 프로그램에서 동시에 수행하는 병렬 프로그래밍의 한 형태이다. 쓰레드는 세미(semi) 프로세스 혹은 light-weighted 프로세스라고 불리우며, 비슷한 일을 하는 fork( )에 비해서 빠른 생성 능력과 적은 메모리를 사용하는 장점이 있다.

모든 쓰레드는 같은 메모리 공간을 (파일 디스크립터와 같은 일부 시스템 자원들) 공유하는 유닉스 스타일의 다중 프로세스와는 다르다. 대신에 자신만의 고유 메모리상에서 동작한다. 그래서 한 프로세스의 두 쓰레드 사이의 문맥 교환(context switch)은 두 프로세스 사이의 문맥 교환보다 훨씬 수월하다.

쓰레드를 사용하는 두 가지 주요한 이유
- 어떤 프로그램들은 하나의 제어흐름 보다는 서로 통신하는 여러 쓰레드로 작성될 때 에만 최고의 성능을 낼 수 있다. (서버 프로그램)
- 다중 프로세서 시스템에서, 쓰레드들은 여러 프로세서상에서 병렬적으로 수행될 수 있다. 이는 한 프로그램이 다른 프로세서에서 작업을 분배할 수 있게 해준다.

쓰레드는 자신만의 스택 메모리 영역을 가지고 코드를 실행한다. 하지만, 실제 프로세스와는 달리 쓰레드는 다른 형제 쓰레드들과 메모리를 공유한다. 이렇듯 쓰레드는 전역 메모리를 공유하게 되므로 fork 보다 더 적은 메모리를 소비하게 된다.
fork 에 비해서 더 빠른 수행을 보이는 이유는 fork 는 기본적으로 모든 메모리의 모든 기술자(파일 기술자)를 copy-on-write 방식으로 자식에게 복사하는데 반해 쓰레드는 많은 부분을 공유하기 때문이다. 하지만, 단점은 모든 쓰레드가 같은 메모리 공간을 공유하게 되므로, 하나의 쓰레드가 잘못된 메모리 연산을 하게 되면, 모든 프로세스가 그 값에 의해 영향을 받게 된다는 점이다. 또 하나는 fork 가 프로세스의 생성 방식에 있어 OS의 보호를 받지만, 쓰레드는 그렇지 않기 때문에 한 개의 쓰레드에 문제가 발생하면, 전체 쓰레드에 영향을 줄 가능성이 커지며 디버깅이 어려워진다.

1.1. 원자성(atomicity)과 휘발성(volatility)
쓰레드에 의해 공유되는 메모리 영역은 접근하는 데에 더욱 주의가 필요하다. 병렬 프로그램은 일반적인 지역 메모리처럼 공유 메모리 객체를 접근할 수 없기 때문이다.
원자성(atomicity)은 어떤 객체에 대한 연산이 분리될 수 없는, 인터럽트 되지않는 과정으로 이루어져야 된다는 개념을 말한다. 공유 메모리 상의 데이터에 대한 연산은 원자적으로 이루어질 수 없다. 게다가 GCC 컴파일러는 종종 레지스터에 공유 변수들의 값을 버퍼링하는 최적화를 수행한다. 레지스터에 공유 메모리의 값을 버퍼링하는 GCC의 최적화를 막기위해 공유 메모리 상의 모든 객체는 volatile 속성으로 선언되어야한다. 한 word의 volatile 객체를 읽고 쓰는 것은 원자적이기 때문이다.

1.2. 잠금(Lock)
결과값을 읽고 저장하는 것은 독립된 메모리 연산이다. ++i 는 항상 공유 메모리 상의 i 값을 1 만큼 증가시키지 않는다. 왜냐하면, 두 연산 사이에 다른 프로세서가 i 값에 접근할 수 있기 때문이다. 이처럼 한 쓰레드가 변수의 값을 바꾸는 동안, 다른 쓰레드가 그 변수에 대한 작업을 할 수 없게 보호하는 시스템콜이 필요하다. 이러한 과정을 lock 이라고 한다. lock 의 일반적인 과정은 다음과 같다.

- 어떤 쓰레드가 i 변수에 대해 lock을 건다.
- lock을 건 쓰레드가 잠긴 변수 i의 값을 수정한다.
- lock을 해제한다.

i 변수는 특정 쓰레드에 의해 lock 이 걸려있게 되므로 그 동안 i 값에 접근하는 다른 쓰레드들은 블럭이 된다. 이유는 한 변수에 대해서는 한 번에 하나의 lock 만을 허용하기 때문이다.

2. PThread (POSIX Thread)

POSIX 쓰레드는 흔히 Pthread 라고 불린다. 이는 POSIX 에서 표준으로 제안한 쓰레드 함수의 모음으로 쓰레드를 지원하기 위한 표준 C 라이브러리 세트를 제공한다. 리눅스 쓰레드가 제공하는 것은 프로토타입을 선언하는 /usr/include/pthread.h 헤더를 통해 이용 가능하다.

다중 쓰레드 프로그램의 작성은 기본적으로 두 단계의 과정이다.
- 공유 변수들에 lock을 걸고 쓰레드를 만들기 위한 pthread 함수들을 사용한다.
- 쓰레드 서브 함수에 넘겨야 할 모든 인자들을 포함하는 구조체를 만든다.

2.1. lock의 초기화

쓰레드 사용의 첫 번째 과정은 모든 lock 들을 초기화하는 것이다. POSIX lock 들은 pthread_mutex_t 타입의 변수로 선언된다.

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

pthread_mutex_init 함수는 mutex 인자가 가리키는 mutex 객체를 mutexattr 에 의해 명시된 mutex 속성에 따라 초기화를 한다. mutexattr이 NULL이면, 디폴트 속성이 사용된다.

2.2. 쓰레드의 생성

POSIX 는 각각의 쓰레드를 나타내기 위해 사용자가 pthread_t 타입의 변수를 선언하도록 하였다. 다음 호출로 쓰레드가 생성된다.

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

성공한다면, 새롭게 생성되는 쓰레드의 ID가 thread 인자가 지시한 영역에 저장되고 0을 반환한다. 에러가 발생하면 0이 아닌 값을 반환한다. 예를들어, func( ) 함수를 수행하는 쓰레드를 만들고 func( )에 arg 변수를 가리키는 포인터를 넘겨주기 위해서는 다음과 같이 한다.

```
pthread_t thread;
pthread_create(&thread, NULL, func, &arg);
```

func( ) 호출은 다음과 같은 프로토타입을 가져야한다.

```
void *func(void *arg);
```

2.3. 종료

마지막 단계로 func( ) 함수의 결과를 접근하기 전에 모든 쓰레드가 종료되어야 할 것이고, 프로세스는 그것을 기다려주어야 할 것이다.

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

th 가 가리키는 쓰레드가 종료할 때까지, 위의 pthread_join 함수를 호출한 쓰레드의 수행을 멈춘다. 만약, thread_return 이 NULL이 아니면, th 의 리턴값은 thread_return 이 가리키는 영역에 저장된다.

2.4. 쓰레드 함수에 데이터 전달

호출한 함수의 데이터를 쓰레드 함수에 넘기는 방법은 두 가지가 있다.

- 전역변수
- 구조체

구조체를 택하는 것이 코드의 모듈성(modularity)을 보존하는 것에 가장 좋은 선택이다. 구조체는 세 가지 단계의 정보를 포함해야 한다. 첫 번째로 공유 변수들과 lock에 대한 정보, 두 번째는 함수에서 필요로 하는 모든 데이터에 관한 정보, 세 번째로 쓰레드를 구분해주는 ID와 쓰레드가 이용할 수 있는 CPU 수에 대한 정보.

넘겨지는 데이터는 모든 쓰레드들 사이에서 공유되는 것이어야 한다. 그러므로 필요한 변수들과 lock들의 포인터를 사용한다.

3. 과제 1

다음 코드를 수행해보면서 pthread 의 매커니즘을 이해한다.

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

#define NPROC 4

pthread_t threads[NPROC];
int done[NPROC];

void *thread_main(void *);

int main()
{
    int i, rc, status;

    printf("pid = %d\n", getpid());

    for (i = 0; i < NPROC; i++) {
        done[i] = 0;
        pthread_create(&threads[i], NULL, &thread_main, (void *)(long)i);
        printf("%d, %ld\n", i, threads[i]);
    }

    for (i = (NPROC-1); i >= 0; i--) {
        done[i] = 1;
        rc = pthread_join(threads[i], (void **)&status);
        if (rc == 0) {
            printf("completed join with thread %d status: %d\n", i, status);
        } else {
            printf("ERROR: return code from pthread_join() is %d, thread %d\n", rc, i);
            return -1;
        }
    }

    return 0;
}

void *thread_main(void *arg)
{
    int i;
    double result = 0.0;

    printf("thread: %d, %d\n", *((int *)&arg), getpid());

    while (!done[*((int *)&arg)]) {
        for (i = 0; i < 1000000; i++) {
            result = result + (double)random();
        }
        printf("thread: %d, result: %e\n", *((int *)&arg), result);
    }

    pthread_exit((void *)0);
}
```

컴파일

컴파일은 pthread 모듈을 로드해야 한다. 따라서, 다음 옵션을 gcc 컴파일 옵션에 붙여준다.

-lpthread

ex) gcc my_task.c -o my_task -lpthread

과제 1

과제 1의 코드는 완전한 듯 보이나, 여러 번 실행했을 때의 결과가 조금씩 다르다. 그 이유를 생각해보아라

4. 과제 2

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

#define L 128

typedef struct {
    double volatile *p_s;
    pthread_mutex_t *p_s_lock;
    int n;
    int nproc;
    double *x;
    double *y;
    int l;
} DATA;

void *SMP_scalprod(void *arg)
{
    register double localsum = 0.0;
    long i;
    DATA D = *(DATA *)arg;

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

    pthread_mutex_lock(D.p_s_lock);

    *(D.p_s) += localsum;

    pthread_mutex_unlock(D.p_s_lock);

    return NULL;
}

main(int argc, char *argv[])
{
    pthread_t *thread;
    void *retval;
    int i, cpu;
    DATA *A;
    volatile double s = 0.0;
    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] = 1;
    }

    pthread_mutex_init(&s_lock, NULL);

    for (i = 0; i < cpu; i++) {
        A[i].n = i;
        A[i].x = x;
        A[i].y = y;
        A[i].l = L;
        A[i].nproc = cpu;
        A[i].p_s = &s;
        A[i].p_s_lock = &s_lock;

        if (pthread_create(&thread[i], NULL, SMP_scalprod, &A[i])) {
            perror("pthread_create failed");
            exit(1);
        }
    }

    for (i = 0; i < cpu; i++) {
        if (pthread_join(thread[i], &retval)) {
            perror("pthread_join failed");
            exit(1);
        }
    }

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

과제 2
과제 2의 코드를 분석하시오. 어떤 동작을 하는 코드, 구조체를 사용하는 이유 등.

제출
과제 1, 2의 코드에 주석을 붙여서 제출한다. 분석의 경우 해당하는 코드의 아랫 부분에 작성한다.