시스템프로그래밍(가) 과제4

소프트웨어학부 20192800 권대현

1. 개발 환경
   - 운영체제: Windows 11 Home
   - 하위 시스템: GNU/Linux 5.15.146.1-microsoft-standard-WSL2 x86_64
   - 리눅스 버전: Ubuntu 22.04.2 LTS
2. 소스코드 설명
   - 20192800-mut.c

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

#define THREADS 4
#define N 3000

int primes[N];
int pflag[N];
int total = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;  //mutex 생성 및 초기화

int is_prime(int v)
{
    int i;
    int bound = floor(sqrt ((double)v)) + 1;
    for (i = 2; i < bound; i++) {
        /* No need to check against known composites */
        if (!pflag[i])
            continue;
        if (v % i == 0) {
        pthread_mutex_lock(&mutex); //lock
            pflag[v] = 0;
        pthread_mutex_unlock(&mutex);   //unlock
            return 0;
        }
    }
    return (v > 1);
}

void *work(void *arg)
{
    int start;
    int end;
    int i;
```

```
    start = (N/THREADS) * (*(int *)arg);
    end = start + N/THREADS;
    for (i = start; i < end; i++) {
        if (is_prime(i)) {
            pthread_mutex_lock(&mutex);  //mutex 로 lock 수행
            primes[total] = i;  //Critical Section
            total++;
            pthread_mutex_unlock(&mutex);  //mutex lock 해제
        }
    }
    return NULL;
}

int main(int argn, char **argv)
{
    int i;
    pthread_t tids[THREADS];
    int thread_ids[THREADS];

    for (i = 0; i < N; i++) {
        pflag[i] = 1;
    }

    for (i = 0; i < THREADS; i++) {
        thread_ids[i] = i;  //thread_id 배열에 저장
        pthread_create(&tids[i], NULL, work, (void *)&thread_ids[i]);
    }

    for (i = 0; i < THREADS; i++) {
        pthread_join(tids[i], NULL);    //join 으로 스레드 종료까지 기다림
    }

    printf("Number of prime numbers between 2 and %d: %d\n",
           N, total);
    for (i = 0; i < total; i++) {
        printf("%d\n", primes[i]);
    }

    pthread_mutex_destroy(&mutex);  //스레드 자원 반환

    return 0;
}
```

- 기존 prime.c에서 발생한 race condition을 mutex를 활용하여 해결한 코드이다.
- 먼저 전역적으로 phtread_mutex_t 타입의 mutex를 선언한다.
  - 선언함과 동시에 PTHREAD_MUTEX_INTIALIZER로 초기화를 해준다. 초기화를 해줬기 때문에 따로 pthread_mutex_init을 수행할 필요 없다.

- Race condition 문제가 발생하는 구간은 is_prime()에서의 pflag[v] = 0;과 work()에서의 prime[total] = i; total++; 구간이다.
  - pflag[v] = 0; 전후로 lock을 걸고 unlock을 해줌으로써 race condition을 해결했다.
  - prime[total] = i; total++; 코드 전후로 lock을 걸고 unlock을 해줌으로써 race condition을 해결했다.
- main() 함수도 다소 수정이 가해졌다.
  - 먼저 생성한 스레드의 id를 저장할 배열인 thread_ids[THREADS]를 선언했다.
  - for문을 통해 THREADS 개수만큼 스레드를 생성할 때, thread_ids[i] = i; 로 해당 for문 인덱스(thread_id)를 스레드 아이디 배열에 저장했고, 이를 pthread_create할 때 인자로 넣어줌으로써, race condition을 방지했다.
  - 이후 for문을 통해 THREADS 개수만큼 pthread_join을 수행하여 모든 스레드가 종료될 때까지 main 함수의 흐름을 정지시켰다.
  - 모든 print 작업이 끝나면 pthread_mutex_destory를 통해 mutex를 메모리에서 해제시켰다.

- 20192800-sem.c

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

#define THREADS 4
#define N 3000

int primes[N];
int pflag[N];
int total = 0;
sem_t sem;  //semaphore 선언

int is_prime(int v)
{
    int i;
    int bound = floor(sqrt ((double)v)) + 1;
    for (i = 2; i < bound; i++) {
        /* No need to check against known composites */
        if (!pflag[i])
            continue;
        if (v % i == 0) {
        sem_wait(&sem); //P()로 wait
            pflag[v] = 0;
        sem_post(&sem); //V()로 post
```

```
            return 0;
        }
    }
    return (v > 1);
}

void *work(void *arg)
{
    int start;
    int end;
    int i;
    start = (N/THREADS) * (*(int *)arg);
    end = start + N/THREADS;
    for (i = start; i < end; i++) {
        if (is_prime(i)) {
        sem_wait(&sem); //P()
            primes[total] = i;  //Critical Section
            total++;
        sem_post(&sem); //V()
        }
    }
    return NULL;
}

int main(int argn, char **argv)
{
    int i;
    pthread_t tids[THREADS];
    int thread_ids[THREADS];

    sem_init(&sem, 0, 1);   //value 를 1 로 설정하여 Binary semaphore 로 초기화

    for (i = 0; i < N; i++) {
        pflag[i] = 1;
    }

    for (i = 0; i < THREADS; i++) {
        thread_ids[i] = i;  //thread_id 배열에 저장
        pthread_create(&tids[i], NULL, work, (void *)&thread_ids[i]);
    }

    for (i = 0; i < THREADS; i++) {
    pthread_join(tids[i], NULL);    //스레드 종료 기다림
    }

    printf("Number of prime numbers between 2 and %d: %d\n",
           N, total);
    for (i = 0; i < total; i++) {
```

```
        printf("%d\n", primes[i]);
    }

    sem_destroy(&sem);  //세마포어 자원 반환

    return 0;
}
```

- 기존 prime.c에서 발생한 race condition을 semaphore를 활용하여 해결한 코드이다.
- 먼저 전역적으로 sem_t 타입의 sem을 선언한다.
  - 따로 초기화하지 않았기 때문에 main() 초반에 sem_init을 통해 초기화해줄 것이다.
- Race condition 문제가 발생하는 구간은 is_prime()에서의 pflag[v] = 0;과 work()에서의 prime[total] = i; total++; 구간이다.
  - pflag[v] = 0; 이전에 sem_wait()으로 상호 배제를 보장하고, 메모리 접근 과정이 끝나면 sem_post()로 작업이 끝났음을 알려서 다른 스레드가 접근하도록 race condition을 해결했다.
  - prime[total] = i; total++; 역시 동일하게 진행했다.
- main() 함수의 수정 사항이다.
  - 먼저 생성한 스레드의 id를 저장할 배열인 thread_ids[THREADS]를 선언했다.
  - for문을 통해 THREADS 개수만큼 스레드를 생성할 때, thread_ids[i] = i; 로 해당 for문 인덱스(thread_id)를 스레드 아이디 배열에 저장했고, 이를 pthread_create할 때 인자로 넣어줌으로써, race condition을 방지했다.
  - sem_init()을 통해 sem에 대해 초기화 작업을 수행했다.
    - 두 번째 인자는 pshared로, 프로세스 간 공유할 것을 정하는 변수다. 0을 넘겨줌으로써 단일 프로세스 내에서만 사용되도록 설정했다.
    - 세 번째 인자는 semaphore의 초기 값을 정하는 변수다. 1로 설정하여 동시 접근 가능한 스레드를 1개로 제한하는 Binary Semaphore로 초기화했다.
  - 이후 for문을 통해 THREADS 개수만큼 pthread_join을 수행하여 모든 스레드가 종료될 때까지 main 함수의 흐름을 정지시켰다.
  - 모든 print 작업이 끝나면 sem_destory()를 통해 sem을 메모리에서 해제시켰다.

- 20192800-cv.c

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

```c
#define THREADS 4
#define N 3000

int primes[N];
int pflag[N];
int total = 0;

int cs_access = 0;  //Critical Section 에 접근 중인 스레드 개수
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;     //condition value 생성 및 초기화
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;  //mutex 생성 및 초기화

int is_prime(int v)
{
    int i;
    int bound = floor(sqrt ((double)v)) + 1;
    for (i = 2; i < bound; i++) {
        /* No need to check against known composites */
        if (!pflag[i])
            continue;
        if (v % i == 0) {
        pthread_mutex_lock(&mutex); //lock
        while(cs_access > 0) {  //만약 Critical Section 에 하나 이상의 스레드가 접근해있다면
            pthread_cond_wait(&cond, &mutex);   //wait 을 통해 스레드 대기
        }
        cs_access++;    //CS 에 접근 중인 상태를 나타내는 변수
            pflag[v] = 0;
        cs_access--;    //CS 에 접근 해제
        pthread_cond_signal(&cond); //signal 을 통해 스레드 깨우기
        pthread_mutex_unlock(&mutex);   //unlock
            return 0;
        }
    }
    return (v > 1);
}

void *work(void *arg)
{
    int start;
    int end;
    int i;
    start = (N/THREADS) * (*(int *)arg);
    end = start + N/THREADS;
    for (i = start; i < end; i++) {
        if (is_prime(i)) {
            pthread_mutex_lock(&mutex);  //mutex 로 lock 수행
```

```
        while(cs_access > 0) {
            pthread_cond_wait(&cond, &mutex);
        }
        cs_access++;
            primes[total] = i;  //Critical Section
            total++;
        cs_access--;
        pthread_cond_signal(&cond);
            pthread_mutex_unlock(&mutex);  //mutex lock 해제
        }
    }
    return NULL;
}

int main(int argn, char **argv)
{
    int i;
    pthread_t tids[THREADS];
    int thread_ids[THREADS];

    for (i = 0; i < N; i++) {
        pflag[i] = 1;
    }

    for (i = 0; i < THREADS; i++) {
        thread_ids[i] = i;  //thread_id 배열에 저장
        pthread_create(&tids[i], NULL, work, (void *)&thread_ids[i]);
    }

    for (i = 0; i < THREADS; i++) {
        pthread_join(tids[i], NULL);
    }

    printf("Number of prime numbers between 2 and %d: %d\n",
           N, total);
    for (i = 0; i < total; i++) {
        printf("%d\n", primes[i]);
    }

    pthread_cond_destroy(&cond);    //조건 변수 자원 반환
    pthread_mutex_destroy(&mutex);  //스레드 자원 반환

    return 0;
}
```

- 기존 prime.c에서 발생한 race condition을 mutex와 condition variable를 활용하여 해결한 코드이다.
- 먼저 전역적으로 phtread_mutex_t 타입의 mutex를 선언한다.

        - 선언함과 동시에 PTHREAD_MUTEX_INTIALIZER로 초기화를 해준다. 초기화를 해줬기 때문에 따로 pthread_mutex_init을 수행할 필요 없다.
    - 전역적으로 pthread_cond_t 타입의 cond를 선언한다.
        - 선언함과 동시에 PTHREAD_COND_INITIALIZER로 초기화를 해준다. 초기화를 해줬기 때문에 따로 pthread_cond_init을 수행할 필요 없다.
    - 전역적으로 Critical Section에 접근 중인 스레드 개수를 나타내는 cs_access를 선언한다.
        - 스레드 내에서 Critical Section에 접근할 때마다 1을 증가시키고, 빠져나올 때 다시 1을 감소시킨다.
    - Race condition 문제가 발생하는 구간은 is_prime()에서의 pflag[v] = 0;과 work()에서의 prime[total] = i; total++; 구간이다.
        - pflag[v] = 0; 에 접근하기 전에, mutex를 통해 lock을 걸어준다. 이후 cs_access가 0이 될 때까지 wait()을 수행하는 while문을 만나게 된다. cs_access가 0이 되면 while문을 빠져나오고 CS 접근 작업을 수행한다. 수행하기 전에 cs_access를 1 증가시켜서 현재 1개의 스레드가 Critical Section에 접근했음을 알리고, 빠져나오면서 1 감소시킨다. 또한, signal을 보내서 대기 상태에 놓인 다른 스레드들을 깨우고, mutex를 통해 lock 상태를 해제하면서 모든 동작이 종료된다.
        - prime[total] = i; total++; 역시 위와 동일하게 진행된다.
    - main() 함수의 수정 사항이다.
        - 먼저 생성한 스레드의 id를 저장할 배열인 thread_ids[THREADS]를 선언했다.
        - for문을 통해 THREADS 개수만큼 스레드를 생성할 때, thread_ids[i] = i; 로 해당 for문 인덱스(thread_id)를 스레드 아이디 배열에 저장했고, 이를 pthread_create할 때 인자로 넣어줌으로써, race condition을 방지했다.
        - 이후 for문을 통해 THREADS 개수만큼 pthread_join을 수행하여 모든 스레드가 종료될 때까지 main 함수의 흐름을 정지시켰다.
        - 모든 print 작업이 끝나면 pthread_mutex_destory를 통해 mutex를 메모리에서 해제시켰다.
        - 마찬가지로 pthread_cond_destory를 통해 cond를 메모리에서 해제시켰다.

- Makefile

```
CC = gcc
CFLAGS = -lm -pthread

all: 20192800-mut 20192800-sem 20192800-cv

20192800-mut: 20192800-mut.c
```

```
    $(CC) -o 20192800-mut 20192800-mut.c $(CFLAGS)

20192800-sem: 20192800-sem.c
    $(CC) -o 20192800-sem 20192800-sem.c $(CFLAGS)

20192800-cv: 20192800-cv.c
    $(CC) -o 20192800-cv 20192800-cv.c $(CFLAGS)
```

- CC 변수에 gcc로 컴파일할 컴파일러를 정해준다.
- CFLAGS 변수에 컴파일 옵션을 정해준다.
  - -lm: math.h 라이브러리를 사용하기 위한 컴파일 옵션이다.
  - -pthread: 스레드 관련 라이브러리를 사용하기 위한 컴파일 옵션이다.
- all: 20192800-mut 20192800-sem 20192800-cv를 명시함으로써, 한번에 세 개의 컴파일을 수행하도록 했다.
- 이후론 각각의 컴파일을 CC -o (파일이름) (파일이름.c) CFLAGS 형태로 진행하도록 코드를 작성했다.

3. 문제점 및 해결 방법

- Race condition 근원지 확인 문제
  - 처음 prime.c를 컴파일하여 실행했을 때, total 개수가 실행마다 다르게 print되는 것을 보고 race condition이 발생하고 있음은 확인했다. 그러나, 정확히 어느 지점에서 발생하고 있는지 확인하기가 어려웠다.
    - 전역 변수인 primes[], total, pflag[]에 접근할 때의 코드가 Critical Section이라고 생각되었고, Critical Section에 다수의 스레드가 접근할 때 race condition이 발생하므로, 해당 코드를 보완하는 쪽으로 코드 개선을 이루는 목표를 세웠다.
- Mutex 구현 문제
  - Critical Section 전후로 pthread_mutex_lock()과 pthread_mutex_unlock()을 걸어줬음에도 불구하고, 매 실행마다 결과값이 달라지는 문제가 발생했다.
    - 먼저 함수마다 선언됐던 mutex를 전역 변수로 선언하는 것으로 수정했다.
    - 기존 스레드 생성 코드를 보면, 모두 &i를 참조하는 것을 확인할 수 있다. 이렇게 됐을 경우 참조하는 주소가 1개로 똑같기 때문에 race condition 문제가 발생할 수 있다. 따라서, THREADS 개수만큼 thread_id 주소를 저장할 배열을 선언하고 이를 넘겨주는 방식으로 해결했다.
    - 마지막으로 스레드 종료를 기다리는 pthread_join을 수행하여 모든 스레드가 종료해야 값을 출력할 수 있게끔 코드를 수정했다.
- Semaphore 구현 문제
  - 처음 semaphore를 구현할 때 counting semaphore로 실행 속도를 높이는 방안을 채택했다. 그러나, 여러 스레드를 접근하게 하니 상호 배제가 보장되지 않고 결과값이 매 실행마다 달라지는 문제가 발생했다.

        - 따라서, counting semaphore 방식을 버리고 binary semaphore을 채택하여 문제를 해결했다. 이를 위해 sem_init()에서 value값에 1을 넘겨주어 binary semaphore로 초기화 시켜줬다.
- Conditional Variable 구현 문제
    - Wait()과 Signal()의 사용 방법에 대해 고민이 생겼다.
        - Critical Section에 들어갔을 때 전역 변수로 선언한 인덱스의 값을 1 증가시켜서 현재 CS에 접근 중인 스레드가 있음을 선언하고, 다른 스레드들은 while문 내에서 인덱스 값이 0이 될 때까지 wait()을 수행하도록 했다. 만약 접근 중이던 스레드가 작업을 모두 수행하면 인덱스 값을 1 감소시키고 signal()을 통해 대기 중인 스레드들을 깨우는 방식으로 구현하여 문제를 해결했다.
- Makefile 작성 문제
    - 처음 gcc를 돌릴 때 math.h의 함수를 찾을 수 없다는 에러가 발생했다.
        - math.h 라이브러리를 import할 때는 pthread.h와 동일하게 컴파일 옵션을 따로 줘야 한다는 것을 인터넷 검색을 통해서 알았고, -lm 옵션을 줌으로써 컴파일에 성공했다.
    - 과제 명세에 따르면, 한번의 make만으로 3개의 컴파일이 이뤄져야 한다. 그러나, 내가 작성한 Makefile로는 단일 컴파일밖에 수행되지 않았다.
        - all: 코드를 작성하여 컴파일을 목표로 하는 파일들을 명시해야 한다는 것을 알았고, all: 20192800-mut 20192800-sem 20192800-cv를 작성하여 문제를 해결했다.