

---

## 운영체제 과제 #3

: 멀티스레드를 이용한 연립방정식 구하는 프로그램 구현

---

과 목	운영체제 _02
전 공	컴퓨터공학과
학 번	20190850
이 름	이수진

### 목 차

I. 문제 기술.....	1
1. 문제 정의 및 해결	
II. 세부 구현사항.....	1
1. 구현 시 어려웠던 코드	
2. 병렬 설계	
III. 체크 포인트.....	4
1. 실행 결과	
2. 결과 분석	
3. 성능 평가	
IV. 결론.....	5

## I. 문제 기술

### 1. 문제 정의 및 해결

연립방정식의 해를 구하는 프로그램을 작성한다. 연립방정식의 풀이는 Gaussian elimination과 back substitution을 이용하고, Multi-thread 방식을 사용하여 성능을 높인다.

이 문제는 반복문을 이용하여 모듈마다 각각 Multi-thread를 생성하고, thread가 한번 종료된 뒤 다시 실행하는 방식으로 해결하였다. 이때 행렬과 두 벡터, 크기, thread의 개수를 포함하는 구조체를 생성하고 thread 함수에 전달할 인자로 사용했다.

## II. 세부 구현사항

### 1. 구현 시 어려웠던 코드

#### (1) thread의 인덱스 전달

multi-process에서는 함수 안에서 process의 인덱스를 인자로 전달된 pid를 이용해서 알아냈다. 그와 달리 multithread는 행렬과 두 벡터, 그들의 크기, thread의 개수를 포함하는 구조체만을 인자로 전달해야 했다. 이는 전역 변수의 사용으로 해결하였다. 그런데 전역 변수 t\_index 하나만으로는 인덱스가 올바르게 전달되지 않아 결과 벡터가 다르게 나오는 오류가 발생하여 함수 안에 지역 변수를 추가하여 t\_index + 1 한 값을 넣어주었다.

```
int t_index = 0;

void *GaussianElimination(void *arg)
{
    int pi = t_index++;
    ...
}
```

#### (2) thread 종료 대기

Gaussian Elimination 함수를 실행 후 multithread를 소멸하고, 다시 생성하여 Back Substitution 함수를 실행하고자 하였다. 처음엔 pthread\_exit()를 사용해 종료하고자 하였으나 앞 함수만 실행 후 프로그램이 종료되는 에러가 발생하였다. 이는 pthread\_join()을 사용하여 해결하였다.

```
for (int j=0; j<p; j++)
    pthread_join(threads1[1], NULL);
...

for (int q; q<p; q++)
    pthread_join(threads2[1], NULL);
...
```

## 2. 병렬 설계 알고리즘

반복문을 이용하여 모듈마다 각각 Multi-thread를 생성하고, thread가 한 번 종료된 뒤 다시 실행하는 방식으로 설계하였다.

### (1) Gaussian elimination

과제 설명에 있는 알고리즘을 사용하였다. Gaussian elimination은 연립방정식 수식을 upper triangle 형태의 방정식으로 변경하는 과정이다.

pi는 자식 thread index로, 함수가 실행될 때마다 시작 부분에서 +1 해주어야 한다. for-i를 이용하여 각 thread 인덱스에 따라 start와 end를 설정한 뒤 아래쪽 행을 0으로 만드는데, 이는 순서가 중요하지 않기 때문에 병렬로 수행한다. for-i, for-j로 A, b를 적절히 대입한 후에는 pthread\_barrier\_wait() 함수를 수행한다.

```
function *Gaussianelimination(arg)
arg <- 전달된 구조체 포인터 변수
data <- arg의 정보를 입력받는 구조체 포인터 변수
p <- data가 가리키는 구조체 안의 thread 개수
t_index <- 자식 thread index를 가리키는 전역 변수
pi <- 자식 thread index를 가리키는 지역 변수
start, end <- thread pi가 담당할 시작 열과 마지막 열의 index
m <- 병렬화해야 하는 블록의 개수
A, b <- data가 가리키는 구조체 안의 행렬과 벡터
n <- 행렬의 가로 크기 또는 벡터의 크기

    data <- arg
    pi <- t_index++

    for l <- 0 to n-1 do
        m <- n-l-1
        start <- (pi*m)/p+1+l
        end <- ((pi+1)*m)/p+l;
        for i <- start to end+1 do
            for j <- 1+l to n do
                 $A[i*n+j] <- A[i*n+j] - (A[i*n+l] / A[l*n+l]) * A[l*n+j]$ 
            end for
             $b[i] <- b[i] - (A[i*n+l] / A[l*n+l]) * b[l]$ 
        end for
        pthread_barrier_wait(&barrier)
    end for

End Gaussianelimination()
```

## (2) Back substitution

과제 설명에 있는 알고리즘을 사용하였다. Back Substitution은 후진 대입법으로 맨 밑의 있는 행부터  $x$ 의 값을 구해 오는 과정이다.

$pi$ 는 자식 thread index로, 함수가 실행될 때마다 시작 부분에서 +1 해주어야 한다. for-i에서 벡터  $c$ ,  $start$ 와  $end$ 를 계산하고, for-j로  $A$ ,  $b$ 를 적절히 대입한다. 그 후 `pthread_barrier_wait()` 함수를 수행한다.

```
function *BackSubstitution(arg)
arg <- 전달된 구조체 포인터 변수
data <- arg의 정보를 입력받는 구조체 포인터 변수
p <- data가 가리키는 구조체 안의 thread 개수
t_index <- 자식 thread index를 가리키는 전역 변수
pi <- 자식 thread index를 가리키는 지역 변수
start, end <- thread pi가 담당할 시작 열과 마지막 열의 index
A, b <- data가 가리키는 구조체 안의 행렬과 벡터
c <- data가 가리키는 구조체 안의 결과 벡터
n <- 행렬의 가로 크기 또는 벡터의 크기

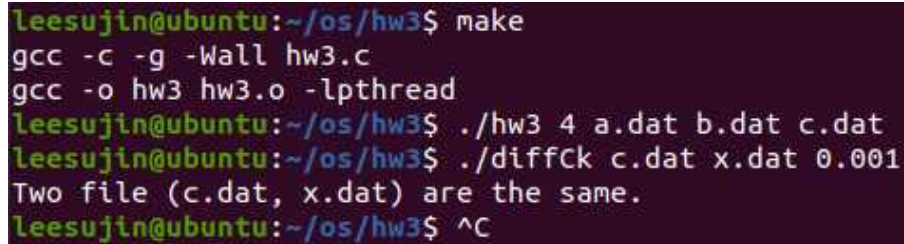
    data <- arg
    pi <- t_index++

    for i <- n-1 to -1 do
        c[i] <- b[i] / a[i*n+i]
        start <- (pi*i)/p
        end <- ((pi+1)*i)/p
        for j <- start to end do
            b[j] <- b[j] - (x[i]*a[j*n+i])
            A[j*n+i] <- 0
        end for
        pthread_barrier_wait(&barrier)
    end for

End BackSubstitution()
```

### III. 체크 포인트

#### 1. 실행 결과



```
leesujin@ubuntu:~/os/hw3$ make
gcc -c -g -Wall hw3.c
gcc -o hw3 hw3.o -lpthread
leesujin@ubuntu:~/os/hw3$ ./hw3 4 a.dat b.dat c.dat
leesujin@ubuntu:~/os/hw3$ ./diffCk c.dat x.dat 0.001
Two file (c.dat, x.dat) are the same.
leesujin@ubuntu:~/os/hw3$ ^C
```

[그림 1] 실행 결과 화면

#### 2. 결과 분석

(1) p개의 thread를 생성할 때는 p개의 process를 생성할 때와 비교해서 주의해야 할 점이 생기지 않는가?

thread 종료 시에 부모 thread까지 종료되지 않도록 주의해야 한다.

(2) Thread들 간에 공유해야 하는 자료구조들은 어떤 것들이며 공유를 위해 어떤 식으로 어떤 위치에 선언되어야 하는가?

행렬 및 두 벡터가 공유되어야 한다. 이는 구조체를 만들어 안에 변수로 선언한다. 그리고 main() 함수에서 구조체 변수를 선언한 뒤 read()로 읽어 들인 행렬 및 두 벡터를 구조체에 대입시킨다.

(3) 어떤 방식으로 병렬 설계를 하였고, 그 이유는 무엇인가?

모듈마다 각각 multithread를 생성하는 방식으로 설계하였다. 아직 thread에 대한 이해가 아직 완벽하지 않았기 때문에 번거로운 방식보다는 쉽게 이해되는 것으로 선택했다.

(4) pthread\_barrier는 어떻게 초기화하고, 어떤 스레드들이 ...wait()를 호출하면 되는가?

초기화는 다음과 같이 객체 barrier와 thread 개수 p를 전달한다.

```
pthread_barrier_init(&barrier, NULL, p);
```

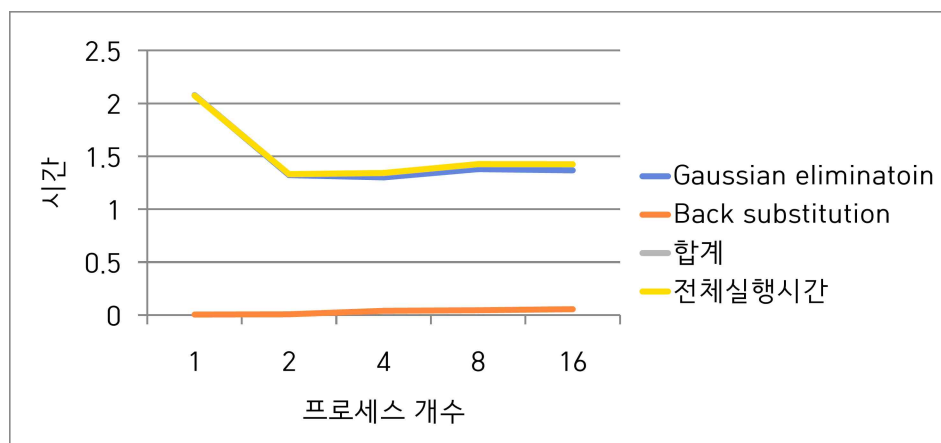
Barrier에 참가한 thread 중 함수 안에서 먼저 도착한 thread가 wait()를 호출한다.

### 3. 성능 평가

(1) 생성된 thread의 개수가 1, 2, 4, 8, 16개 일 때의 시간을 구해 표 1을 채우고 전체 실행시간의 변화를 그래프로 표시하시오.

[표 1] 실행 시간 표

프로세스 개수	계산 시간			전체실행시간 (I/O 포함)
	Gaussian elimination	Back substitution	합계	
1	2.076028	0.004763	2.080791	2.072299
2	1.322353	0.007768	1.330120	1.332187
4	1.301058	0.040152	1.341210	1.343224
8	1.379182	0.045617	1.424799	1.427153
16	1.367576	0.055530	1.423106	1.425882



[그림 3] 실행 시간 그래프

## IV. 결론

Multithread는 Multi-process처럼 단일 thread에서 오래 걸리는 시간 문제를 해결한다. Multi-process와 실행 시간은 비슷하나, 구현 측면에서는 Multithread가 비교적 간단함을 알 수 있다.