운영체제 Assignment2
2014121136 경제학과 구한모

<사전조사 보고서>

1. Process 와 Thread의 차이

process와 thread는 다음과 같은 차이점을 보인다.[i]

	process	Thread
PCB	서로 다른 Process는 서로 다른 PCB를 가짐	한 PCB 내부에 여러 개의 다른 thread가 존재할 수 있음
실행	한 OS 에서는 한 process만 실행 가능함. Multi process는 일을 교대로 처리(alternate)하는 것임.	한 OS가 여러 thread를 동시에 처리할 수 있음.
주소영역	프로세스는 각자의 주소영역이 별도로 존재한다.	한 프로세스 내의 여러 스레드들은 주소영역을 공유한다.
문맥교환 속도	스레드에 비해 상대적으로 느린 문맥 교환(context switch)	프로세스에 비해 상대적으로 빠른 문맥 교환(context switch)
태스크 진행 중 문제 발생시	하나의 프로세스에서 생겨난 문제는 부모나 자식 프로세스 등, 기타 다른 프로세스에 대개 영향을 끼치 않는다.	하나의 스레드에서 생겨난 문제는 그 스레드를 포함하는 프로세스 전체에 영향을 끼치며, 다른 스레드에도 영향을 미칠 수 있다.
공통점	프로세스와 스레드는 각자가 각자의 논리적인 흐름을 따로 가진다는 점과, 각각이 개별적으로 스케줄링 된다는 공통점을 가진다.	

2. Process와 Thread의 리눅스에서의 구조 및 구현 등의 차이

	process	Thread
생성방식	fork() 시스템콜을 이용하여 자신의 PCB를 복사하지만 자신의 pid와는 다른 pid의 PCB를 만드는 형식으로 생성	clone() 시스템콜을 이용하여 자신의 프로세스의 pid를 공유하는 프로세스를 만드는 형식으로 생성
구조	결국 각자가 하나의 PCB 이미지라는 것은 공통점이고 리눅스는 process와 thread를 따로 구별하지 않고 task라는 이름으로 부른다.	
process 간, thread 간 데이터 공유	IPC 방법을 통해 데이터를 공유하게 된다. 즉, 커널모드를 통하여 데이터를 공유한다.	Flag를 이용하여 parent thread와 child thread의 데이터 공유 정도를 설정할 수 있게 한다.

3. 멀티프로세싱과 멀티스레딩

A. 시간적인 차이 : 멀티프로세싱은 다수의 프로세스를 교대로 돌리는 방식이기에 한 프로세스가 끝나고 다른 프로세스로 넘어갈 때 mode-switching이 발생하게 되어 추가적인 시간이 걸린다. 또한 새

프로세스를 생성할 때에도 새로운 PCB를 fork함수를 통해 만들어야 하기에 상대적으로 오래걸린다. 반면, 멀티스레딩은 한 프로세스 내의 여러 스레드가 한번에 동시에 작동되기 때문에 mode-switching이 발생하지 않아 상대적으로 더 짧은 시간이 걸리고 생성시에도

B. 공유 문제 : 멀티스레딩의 경우 한 프로세스 내의 여러 스레드가 동시에 돌아가는 것인 만큼 코드와 데이터 그리고 커널 컨텍스트를 공유하게 된다. 반면, 멀티프로세싱의 경우 여러 PCB의 프로세스가 돌아가기에 특정 데이터를 공유하기 위해서는 커널모드를 통해 IPC의 여러 방법을 이용해야 한다.

C. 병렬CPU 사용 가능 여부 : 멀티프로세싱의 경우 여러 프로세스를 한 CPU가 교대로 처리하는 개념이기에 병렬 CPU가 있어도 동시에 여러 task를 처리할 수 없었으나 멀티스레딩의 경우 한 프로세스 내의 여러 스레드가 동시에 작동되는 방법이기 때문에 병렬 CPU를 효과적으로 사용할 수 있다.

4. 프로세스간 통신(IPC)

A. IPC의 필요성 : 프로세스간 데이터 등의 공유와 동기화를 위해 IPC라는 개념은 필수적이다. 또한 두 프로세스가 동시에 한 데이터에 접근할 때에 충돌(race condition)이 일어나지 않도록 하여 원하는 방식으로 데이터를 관리하는 방법이 필요하다. -이를 위하여 3가지의 기준을 만족하는 방법들이 필요한데 바로 1) Mutual Exclusion, 2)Progress, 3) Bounded Waiting 이다. 즉, 1) 한 번에 한 프로세스만 데이터에 접근하며 2) 데이터가 필요한 프로세스가 있다면 데이터는 항상 접근되어야 되며, 3) 어떤 프로세스라도 데이터에 접근하기 위해 기다리는 시간은 유한해야 한다는 점이다.

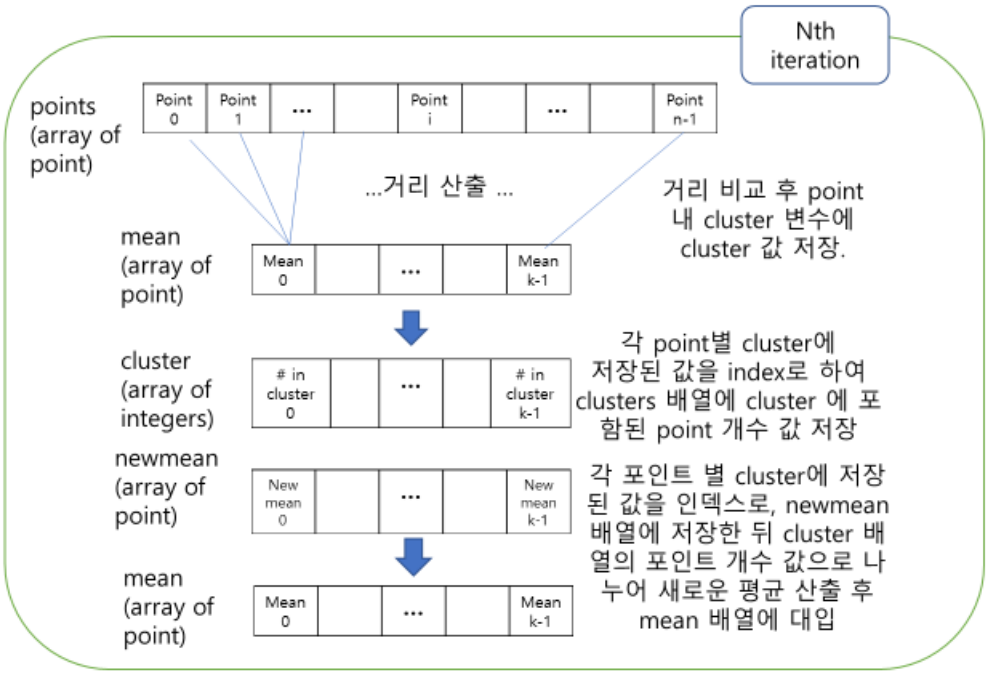
B. Basic IPC (Pipe, FIFO): 파이프는 한 프로세스가 다른 프로세스로 데이터를 보내기 위해 사용하는 방식이다. 이는 일방적이기 때문에 두 프로세스의 소통을 위해서는 서로 다른 방향의 파이프를 만들어 프로세스간 데이터를 공유하게 된다. 그리고 이를 FIFO 혹은 Named Pipe라 부른다.

C. System V IPC (Message Passing, shared memory, semaphores) : UNIX System V에서 구현된 이 IPC방법들은 단일 host 내에서 프로세스간 통신을 가능하게 하는 방법들이다. 메시지의 경우 프로세스가 메시지를 주고 받을 수 있는 함수를 제공하며, 공유 메모리의 경우 여러 프로세스들이 데이터를 공유할 수 있는 특정 메모리 공간을 지정하여 데이터를 공유하게 된다. 세마포의 경우 보통 프로세스 동기화를 위해 사용되며, 프로세스의 진행에 따라 공유 데이터에 대한 접근 순서를 스케줄링하는 역할을 하게 된다.

<실습 과제 수행 보고서>

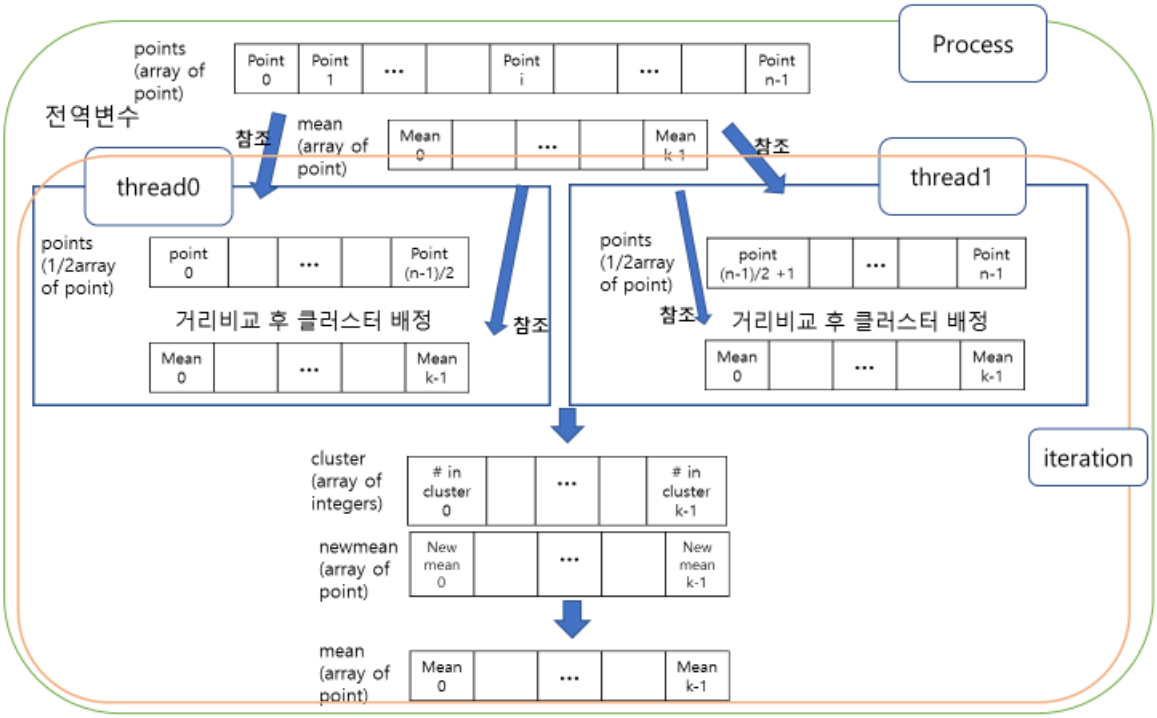
1. 작성한 프로그램의 동작 과정과 구현 방법(도식화 자료를 이용하여 설명)

A. Noparellel – 즉 병렬 프로그래밍을 이용하지 않은 K-means clustering 방법의 도식



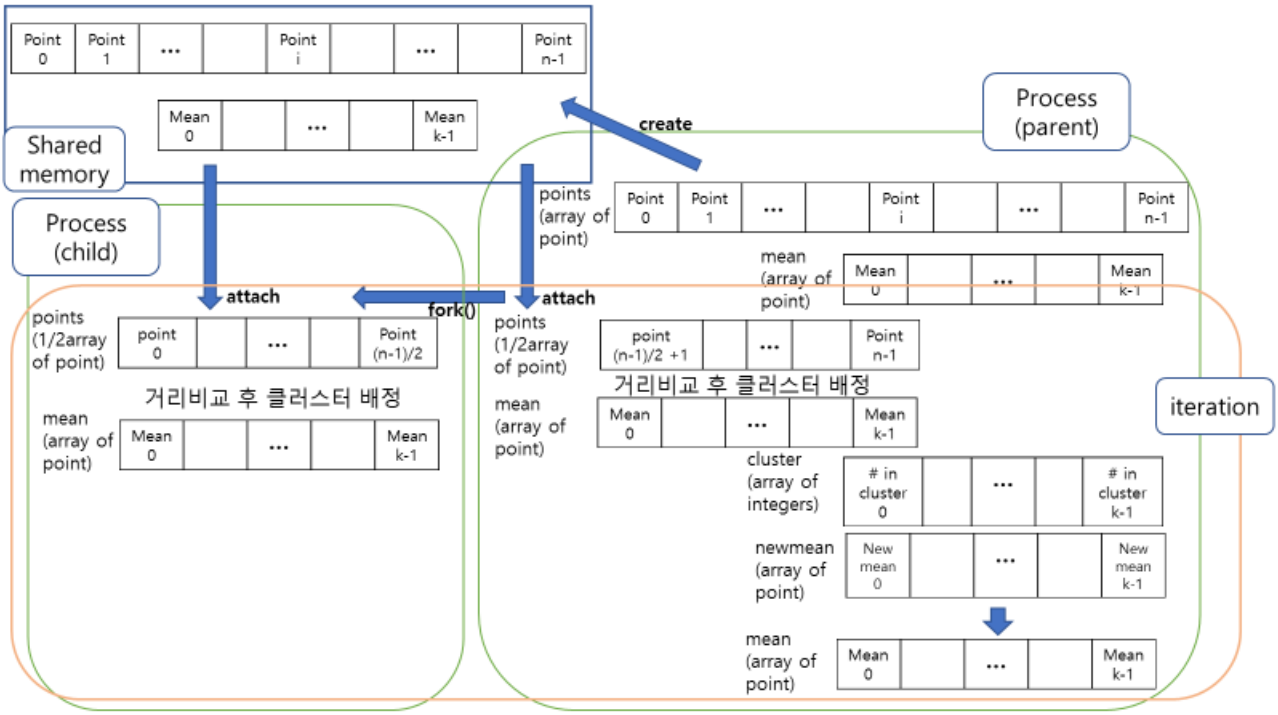
즉 위와 같은 방법으로 한 testcase의 한 iteration을 구성하였다. 밑에서 설명할 병렬프로그래밍은 전체적인 컨셉은 똑같으나, 위에서의 points 배열을 둘로 나누어 두 프로세스 혹은 두 스레드가 처리하도록 구현하려 한다. 따라서 도식에서 설명은 생략하고 그림으로만 나타내고자 한다.

B. Thread – 멀티 스레딩을 이용한 K-means clustering의 도식



한 프로세스내에서 두 개의 thread를 만든 뒤, 데이터 병렬로써 병렬 프로그래밍을 사용하였다. points배열을 반으로 나누어 한 스레드에서 배열의 앞 부분을 평균과 거리비교를 하여 클러스터 배정을 하였고, 나머지 반은 다른 스레드에서 클러스터 배정을 하였다. 도식에서는 표시되어있지 않지만, pthread_crate함수를 이용하여 thread를 생성하였으며, 동기화 목적을 위하여 두 스레드 모두 진행이 끝나고 나서 새로운 평균값을 계산할 수 있도록 process 내부에서 pthread_join함수를 사용하였다.[ii] 이 후 과정은 noparellel과 비슷하며 이러한 과정을 iteration input 값에 맞게 반복하도록 설계하였다.

C. Process – 멀티 프로세싱을 이용한 k-means clustering의 도식.



부모 프로세스에서 IPC 수단으로 공유메모리를 미리 만들고 입력 받은 점들의 값을 모두 넣어 놓은 후, iteration 돌 때 마다 child process를 생성하기 위해 fork문을 사용하여 새로운 프로세스를 만들었다. 그리고 부모와 자식 프로세스 각각에 공유메모리를 attach하여 접근 가능하도록 만들었다. 역시 데이터 병렬을 이용하여 점들의 앞 반 부분은 자식 프로세스가, 나머지 반 은 부모 프로세스가 평균과의 거리 비교를 하여 클러스터를 배정하였으며, 도식에는 표시되어 있지 않지만, 자식 프로세스가 끝나고 나서야 부모 프로세스에서 다시금 새로운 평균을 계산할 수 있도록 자식 프로세스가 끝났을 때에는 exit함수를, 부모 프로세스에서는 wait함수를 이용하여 동기화를 구현하였다.

2. 작성한 Makefile 에 대한 설명

A. Makefile 내용

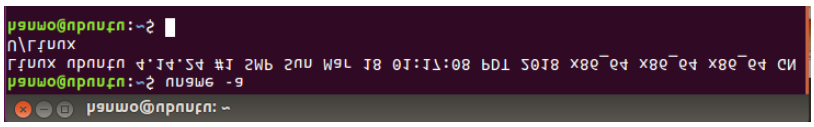
```
CXX      = g++
DEBUG    = -g
CXXFLAGS  = -Wall $(DEBUG) --std=c++11 -pthread
LDFLAGS = -lpthread

all: process noparellel thread

process: process.o
    g++ process.cpp -o process

process.o : process.cpp
    g++ process.cpp

noparellel: noparellel.o
    g++ noparellel.cpp -o noparellel

noparellel.o : noparellel.cpp
    g++ noparellel.cpp

thread: thread.o
    $(CXX) $(CXXFLAGS) thread.cpp -o thread

thread.o : thread.cpp
    g++ $(CXXFLAGS) thread.cpp

clean:
    rm -f thread process noparellel thread.o process.o noparellel.o

```

B. 설명 : 위의 4줄의 커맨드는 매크로로서, 자주 쓰이는 커맨드들을 대체할 수 있는 문구이다. cpp로 코딩하였으므로 g++라는 커맨드를 매크로로 설정하였다. 다만 -Wall -g --std=c++11 -pthread는 pthread를 빌드할 때 pthread 인식문제가 나오던 것의 해결방법을 찾다가 인터넷에서 발견한 코딩으로[iii] 시도해보아 문제는 해결하였으나 그 이유나 원리를 아직 찾지 못하였다. 그 밖에 all: 뒤에 명령어를 명시함으로써 make 커맨드 한번으로 process, noparellel, thread 라는 명령어를 수행하도록 만들었고 각각의 명령어는 컴파일 및 빌드까지 한번에 이어질 수 있도록 코딩을 구성하였다. 또한

clean을 통해 만들어진 바이너리파일과 오브젝트 파일들을 모두 없앨 수 있도록 구현하였다.

3. uname -a 실행 결과 화면과 개발 환경 명시(CPU, 메모리 등)

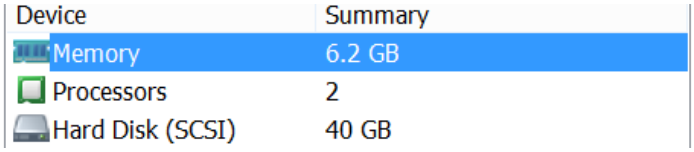
A. uname -a 실행결과 화면



B. 메모리, CPU 등 실제 하드웨어 개발 환경

제조업체:	Samsung Electronics
프로세서:	Intel(R) Core(TM) i5-4200U CPU @ 1.60GHz 2.30 GHz
설치된 메모리(RAM):	8.00GB
시스템 종류:	64비트 운영 체제, x64 기반 프로세서

C. VMware 설정 기반 개발 환경



4. 과제 수행 중 발생 하 애로사항 및 해결방법

A. 우선 c++ 파일 코드 작성 및 makefile 컴파일을 용이하게 할 수 있도록 eclipse oxygen을 설치하였다.

<process.cpp>관련 문제 해결 방법.

B. #include <unistd.h>를 이용하여도 eclipse 내에서 fork()가 인식되지 않는 문제

```
int shmid;
int *cal_num;
void *shared_memory = (void*)0;
shmid = shmget((key_t)1, sizeof(int)*k_num, IPC_CREAT | 0666);
pid_t pid =  fork();
```

minGW에서는 unistd.h에 fork함수가 포함되어 있지 않다는 답변에 착안하여 makefile 내의 flag 를 g 에서 g++로 바꾼 뒤 인식문제 해결[iv] / 정확히는 terminal에서 g++로 커맨드를 바꾸니 컴파일이 가능하게 되었다.

```
4
5 ifeq ($(BUILD_MODE),debug)
6     CFLAGS += -g++
```

C. process.cpp 코드 작성 시, 프로세스 간 데이터 공유를 위한 공유데이터 설정

```
int shmid_mean, shmid_points, shmid_clusters ;
int *cal_num;
void *shared_memory = (void*)0;
shmid_mean = shmget((key_t)1, sizeof(int)*k_num, IPC_CREAT | 0666);
shmid_points = shmget((key_t)2, sizeof(point)*n_num, IPC_CREAT | 0666);
shmid_clusters = shmget((key_t)3, sizeof(list<point>)*k_num, IPC_CREAT | 0666);
```

```
23 #include <sys/ipc.h>
24 #include <sys/shm.h>
```

위의 헤더들을 포함하여 데이터 공유를 위한 함수를 끌어왔다. 프로세스 간에는 평균의 배열을 공유할 수 있는 메모리와 주어진 점들을 공유할 수 있는 메모리, 그리고 점들을 포함하는 클러스터를 공

유할 수 있는 메모리를 설정하였다.[v] 그리고 IPC를 데이터 병렬을 이용하여 작성할 것이기 때문에 따로 semaphore와 같은 접근 제어는 설정하지 않았다. 또한 root 가 아닌 모드로 시행할 때에도 메모리에 접근할 수 있도록 권한을 0666으로 설정하였다[vi]

D. IPC설정 시, point 객체의 포인터를 공유 메모리에 담았으나 메모리 공유가 원활히 이뤄지지 않던 문제.

shmget 함수와 shmat 함수를 이용하여 point객체의 배열인 points[]의 포인터를 공유메모리에 담았으나 points[0]값만 저장되고 나머지는 저장되지 않는 문제가 발생하였다. 첫 포인터만 공유메모리에 담았기 때문이었으므로 for문을 이용하여 공유메모리에 배열 전체를 저장할 수 있도록 코드를 수정하였다

```
int shmid_mean, shmid_points, shmid_clusters ;
shmid_mean = shmget((key_t)5, sizeof(point)*(k_num), IPC_CREAT | 0666);
shmid_points = shmget((key_t)6, sizeof(point)*n_num, IPC_CREAT | 0666);


//now attach those shared memories to the process
void* shm_mean = shmat(shmid_mean, (void*)0, 0);
void* shm_points = shmat(shmid_points, (void*)0, 0);


point* shared_mean;
point* shared_points;


shared_mean = (point*)shm_mean;
*shared_mean = *mean;
for(int m = 0; m<k_num; m++){
    shared_mean[m] = mean[m];
}
shared_points = (point*)shm_points;
*shared_points = *points;
for(int m = 0; m<n_num; m++){
    shared_points[m] = points[m];
}
```

E. Segmentation Fault 문제 – process.cpp를 처음으로 컴파일 및 실행하였을 때에는 작동은 되었으나, 그 이후로 segmentation fault라는 오류가 발생하여 컴파일 및 실행이 되지 않는 상황이 발생하였다. 온라인 검색 결과 이는 프로세스가 종료되더라도 공유메모리가 남아있는 경우 때문임을 발견하고[vii] 또한, shmget함수에서 key_t값을 바꿔보며 해결이 되는 것을 발견하여, 매 testcase가 끝날 때마다 shmctl함수를 이용하여 공유메모리를 정리, 제거하는 코드를 구현하였다

```
shmctl(shmid_mean, IPC_RMID,0);
shmctl(shmid_points,IPC_RMID,0);
for(int p=0; p<n_num; p++){
    cout<<points[p].cluster<<endl;
```

<thread 관련 애로사항 및 해결 방법>

F. thread.cpp 작성 할 때 pthread관련 함수들이 인식되지 않는 문제

pthread_create, pthread_join 등 관련 함수들을 이용하여 thread를 구현하는 방법은 인터넷을 참조하여 알아내었으나[viii] 이클립스에서는 이를 인식하지 못하고 있었다. 원인은 애초에 <pthread.h>를 include하지 않음에 있었고 해당 헤더를 포함함으로써 문제를 해결하였다.

G. thread.cpp 작성 시 pthread_create가 인식되지 않는 문제

```
hanmo@ubuntu:~/eclipse-workspace/hw2_2014121136$ g++ thread.cpp
/tmp/ccIYhbtk.o: In function `main':
thread.cpp:(.text+0x709): undefined reference to `pthread_create'
thread.cpp:(.text+0x72d): undefined reference to `pthread_create'
thread.cpp:(.text+0x746): undefined reference to `pthread_join'
thread.cpp:(.text+0x75f): undefined reference to `pthread_join'
collect2: error: ld returned 1 exit status
```

-lpthread 라는 커맨드를 추가해야 한다는 정보를 발견한 뒤 커맨드로 이용하여 컴파일.[ix]

```
hanmo@ubuntu:~/eclipse-workspace/hw2_2014121136$ g++ -o thread.cpp -lpthread
/usr/lib/gcc/x86_64-linux-gnu/5/../../../x86_64-linux-gnu/crt1.o: In function `_
start':
(.text+0x20): undefined reference to `main'
collect2: error: ld returned 1 exit status
```

그러나 여전히 문제 발생.

```
hanmo@ubuntu:~/eclipse-workspace/hw2_2014121136$ g++ thread.cpp -lpthread
hanmo@ubuntu:~/eclipse-workspace/hw2_2014121136$ ./a.out
```

제대로된 커맨드로 컴파일 하여 해결

H. makefile 관련 문제 – pthread를 인식하지 못하는 문제 발생

```
make: *** [thread] Error 1
hanmo@ubuntu:~/eclipse-workspace/hw2_2014121136$ make
g++     thread.cpp   -o thread
/tmp/ccCv7NHo.o: In function `main':
thread.cpp:(.text+0x71a): undefined reference to `pthread_create'
thread.cpp:(.text+0x73b): undefined reference to `pthread_create'
thread.cpp:(.text+0x754): undefined reference to `pthread_join'
thread.cpp:(.text+0x76d): undefined reference to `pthread_join'
collect2: error: ld returned 1 exit status
<builtin>: recipe for target 'thread' failed
make: *** [thread] Error 1
```

처음에 make filed에서 thread 부분을

thread: thread.o

g++ -lpthread thread.cpp -o thread 의 형식으로 작성하였으나 위의 문제가 발생하였고 온라인 검색 결과 -Wall -g --std=c++11 -pthread 의 명령어를 이용하면 된다고 하여[x] 이하와 같이 작성.

```
CXX      = g++
DEBUG    = -g
CXXFLAGS  = -Wall $(DEBUG) --std=c++11 -pthread
LDFLAGS = -lpthread

all: process noparellel thread

process: process.o
    g++ process.cpp -o process

process.o : process.cpp
    g++ process.cpp

noparellel: noparellel.o
    g++ noparellel.cpp -o noparellel

noparellel.o : noparellel.cpp
    g++ noparellel.cpp

thread: thread.o
    $(CXX) $(CXXFLAGS) thread.cpp -o thread

thread.o : thread.cpp
    g++ $(CXXFLAGS) thread.cpp

clean:
    rm -f thread process noparellel thread.o process.o noparellel.o
```

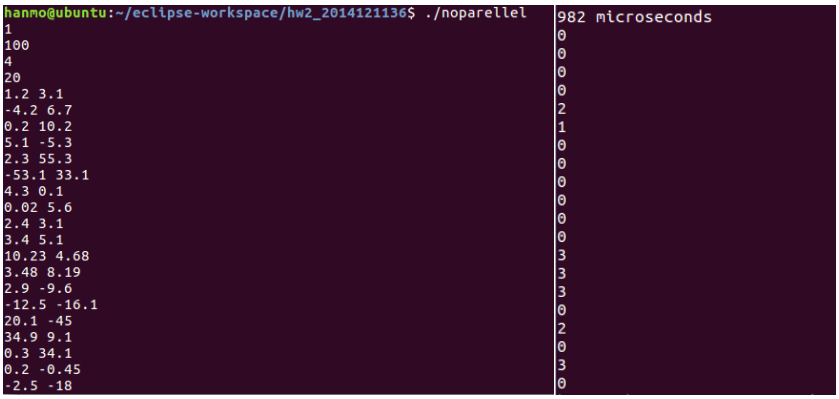
I. 시간 측정 오류문제 : 시간이 음수로 자주 나오는 경우가 있어 이를 해결하고자 하였음. timespec이라는 구조체를 활용하였는데, 이것이 자주 difference만을 반환하는 경우가 있어 이에 대한 해결책을 온라인에서 구하였다. 음수인 경우에 10억을 더하면 그 차이만큼을 반환할 수 있다고 한다.[xi]

```
    if(tse.tv_nsec-tss.tv_nsec>0){
            cout<<((long long)tse.tv_nsec-(long long)tss.tv_nsec)/(long long)1000<<" microseconds"<<endl;
        }else{
            cout<<(((long long)tse.tv_nsec-(long long)tss.tv_nsec)+1000000000)/(long long)1000<<" microseconds"<<
        }
```

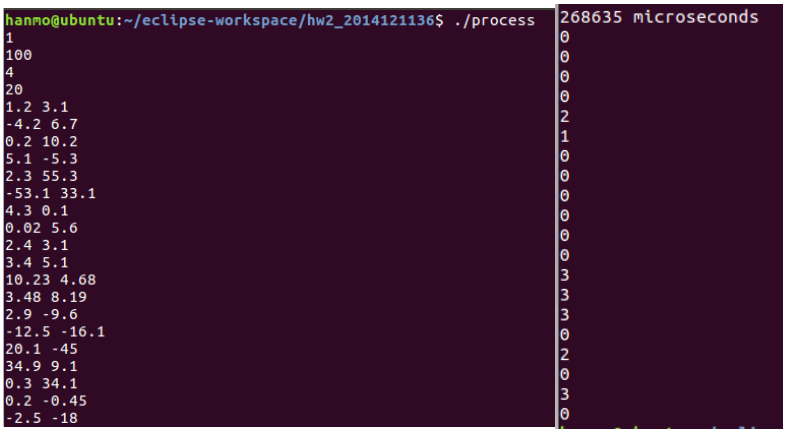
5. 결과 화면과 결과에 대한 분석 내용

A. 병렬 프로그램을 적용하지 않은 것, 멀티스레딩, 멀티 프로세싱의 차이를 전체 수행시간 관점에서 분석

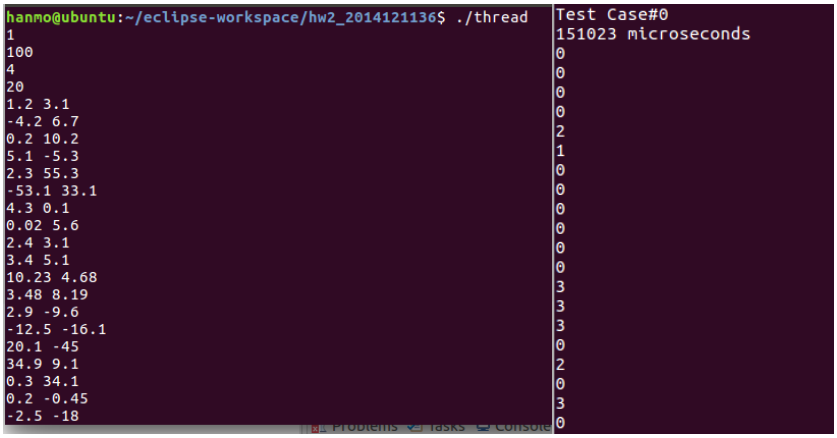
i. Noparellel – (100 iteration, k = 4, n= 20)



ii. Process – (위와 동일)

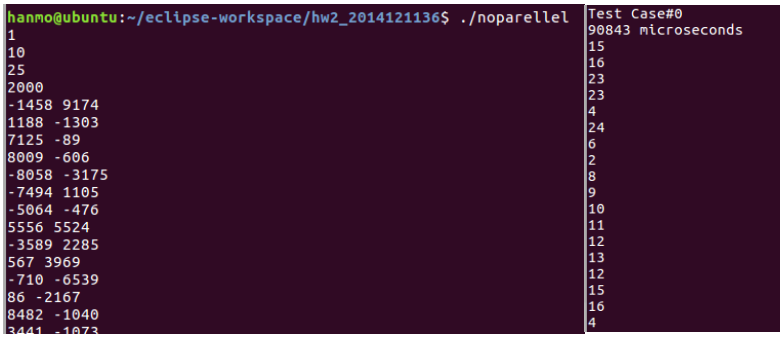


iii. Thread – (위와 동일)



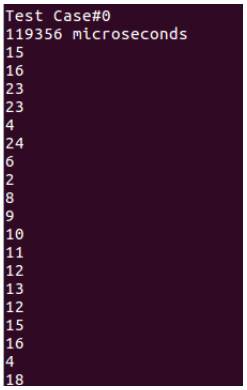
인풋의 크기가 작을 때는 병렬프로그래밍을 사용하지 않은 것이 멀티스레딩, 멀티프로세싱에 비해 압도적으로 빠르다. 스레드와 프로세스를 새로 생성하는 과정에서 걸리는 오버헤드가 병렬프로그래밍의 이점을 모두 상쇄시키기 때문으로 분석된다. 그 다음으로는 멀티스레딩이 멀티프로세싱보다 더 빠름을 알 수 있는데, 스레드를 새로 생성하는 것이 프로세스를 생성하는 것보다 훨씬 빠르기 때문을 알 수 있다.

iv. Noparellel(iteration = 10, k = 25, n = 2000)



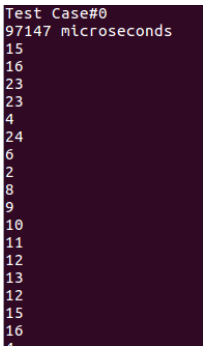
인풋의 크기를 2000으로 늘린 경우 반복횟수(iteration)가 줄어도 병렬 프로그래밍을 이용하지 않으면 시간이 약 100배 더 걸림을 알 수 있다.

v. Process(위와 같음)



인풋의 크기가 2000으로 늘어나면, 병렬 프로그래밍이 100배 더 시간이 걸린 것에 비하여 오히려 시간이 줄어든 것을 볼 수 있다. 이는 아마도 데이터 병렬 특성 상 인풋의 크기보다 반복 횟수에 더 많은 영향을 받으므로, 인풋 데이터는 크기가 커져도 병렬로 처리가 가능한 반면, 반복 횟수가 줄어듦으로써 오버헤드가 적게 걸려 큰 증가가 일어나지 않음을 생각해볼 수 있다.

vi. Thread(위와 같음)

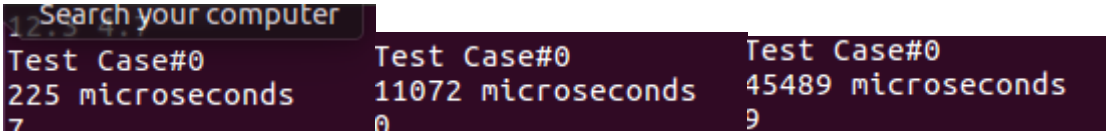


멀티스레딩의 경우 위와 같은 결론으로 분석이 가능하다. 거기에 더해서 프로세스와 마찬가지로 데이터를 반으로 처리함에도 불구하고 더 짧은 시간이 소요됨을 보았을 때, 프로세스에 비해 그 생성시간이 훨씬 짧음을 알 수 있다.

B. 인풋 데이터의 크기를 변경해가며 세 프로그램의 성능 분석

같은 iteration, k에 대한 프로그램의 성능 분석(iteration = 10, k = 10, n=20, 200, 2000)

Noparellel



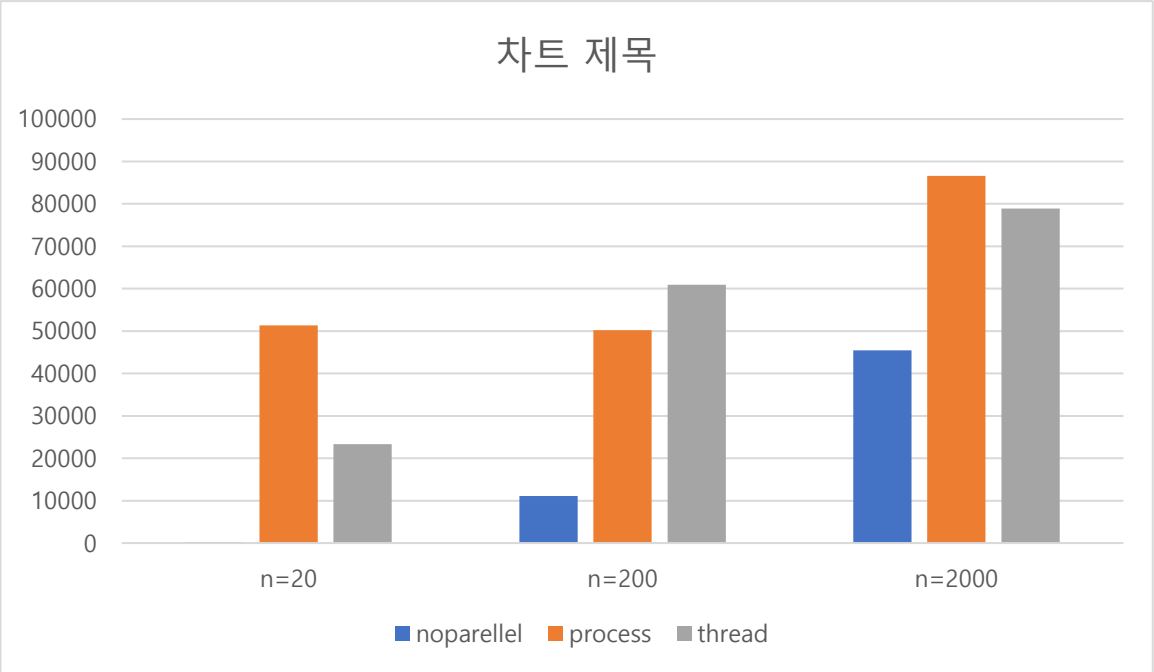
Process

```
Test Case#0
51320 microseconds
Test Case#0
50226 microseconds
Test Case#0
86601 microseconds
```

Thread

```
Test Case#0
23358 microseconds
Test Case#0
60940 microseconds
Test Case#0
78906 microseconds
```

위처럼



차트로 표현하면 위와 같은 형식의 그래프가 나타남을 알 수 있다. 200대 단위에서는 그 인풋 크기에 비해 오버헤드가 아직은 크지만, 인풋 크기가 증가함에 따라 처리 시간의 증가 추세는 noparellel에 비해 병렬 프로그래밍이 훨씬 나음을 볼 수 있다. (사실, process의 n=200의 값은 기타 여파로 인한 outlier라고 생각된다.)

C. 그 외 분석 결과

동료 학우에게 10만개짜리 데이터를 받아 돌려본 결과, (iteration = 10, k = 10,000 , n= 100,000) thread 는 2분 16초, process는 3분 그리고 noparellel은 14분 32초가 걸림을 알 수 있었다. 다만 인풋의 크기가 너무 커서 아웃풋이 너무 출력이 많이 되는 바람에 그 정확한 시간을 촬영할 수는 없었지만, 실제로 시간을 재며 측정한 결과, 인풋의 크기가 굉장히 커지면 오히려 병렬프로그래밍을 쓰지 않는 것이 더 비효율적임을 알 수 있었다.

---

[i] https://www.joinc.co.kr/w/Site/system_programing/Book_LSP/ch07_Thread

[ii] https://www.joinc.co.kr/w/Site/system_programing/Book_LSP/ch07_Thread

[iii] https://stackoverflow.com/questions/33689044/how-to-include-std-c11-and-lpthread-in-makefile?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa

[iv] https://stackoverflow.com/questions/9612315/why-does-my-compiler-not-accept-fork-despite-my-inclusion-of-

unistd-h?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa

[v] https://www.joinc.co.kr/w/Site/system_programing/IPC/SharedMemory

[vi] https://stackoverflow.com/questions/26120212/implementing-shared-memory-without-root-privilege?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa

[vii] https://www.joinc.co.kr/w/Site/system_programing/IPC/SharedMemory

[viii] https://www.joinc.co.kr/w/Site/system_programing/Book_LSP/ch07_Thread

[ix] https://stackoverflow.com/questions/17264984/undefined-reference-to-pthread-create

[x] https://stackoverflow.com/questions/33689044/how-to-include-std-c11-and-lpthread-in-makefile?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa

[xi] http://www.guyrutenberg.com/2007/09/22/profiling-code-using-clock_gettime/