OS 과제 2

2015147574 백진우

1. 사전 조사 보고서

1) 프로세스와 스레드

a) 정의

프로세스 : 컴퓨터에서 연속적으로 실행되고 있는 컴퓨터 프로그램

스레드 : 프로세스 내에서 실행되는 여러 흐름의 단위

b) 차이점

Process	- Code, Data, Heap, Stack 영역으로 이루어짐 - 각각의 Memory space를 차지한다.
Thread	- 프로세스 안에서 동작하여 Code, Data, Heap 영역을 공유하고 별도의 Stack만 가지고 있다. - Context switching시 Stack영역만 Switching하면 되므로 프로세스 스위칭보다 빠르다. - 쓰레드 간 자원 공유가 가능하여 편리하지만 자원 동기화의 문제가 있다.

프로세스는 스레드를 담는 컨테이너 개념이며 스레드는 하나의 포로세스 안에 담기지만, 프로세스는 여러 개의 스레드를 가질 수 있음.

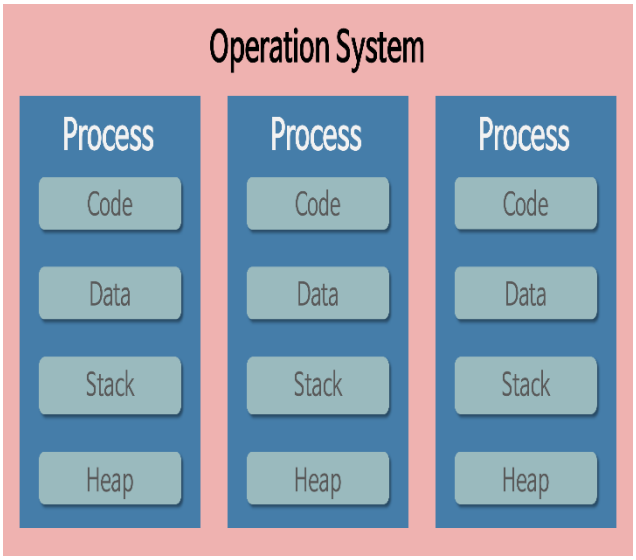
C) 리눅스에서의 구조 및 구현

리눅스에서의 Process는 task_struct라는 거대한 자료구조로서, image, context(program context, kernel context)로 구성되어져 있다. 이러한 구조를 PCB라고도 한다. PCB는 특정한 프로세스를 관리할 필요가 있는 정보를 포함하는 운영 체제 커널의 자료 구조이다. 오른쪽 그림은 PCB가 가지고 있는 정보를 나타낸 것이다. Parent process가 fork()를 호출하면 부모 프로세스를 그대로 복사한 자식 프로세스가 생성되고 이때 PCB도 같이 생성된다. 이러한 복사과정은 프로세스 fork() 할 때 마다 heavy weight copy가 발생해 성능 저하가 된다.

리눅스의 task_struct를 살펴보면 몇몇 struct들은 pointer 형식으로 선언되어 있다. Task를 구성하는 몇몇 기본정보들만 process가 처리하고, 나머지는 point

형식은 부모와 공유해서 사용하는 것을 light weight copy라고 한다. 리눅스에서는 스레드와 프로세스를 따로 구분하지 않는다. Child와 parent task가 포인터로 연결되어 있는 것이 스레드이다. Task 생성은 for()명령어 대신에 Clone()이라는 시스템 콜을 사용하여 Thread를 생성할 수 있다.

● **프로세스**



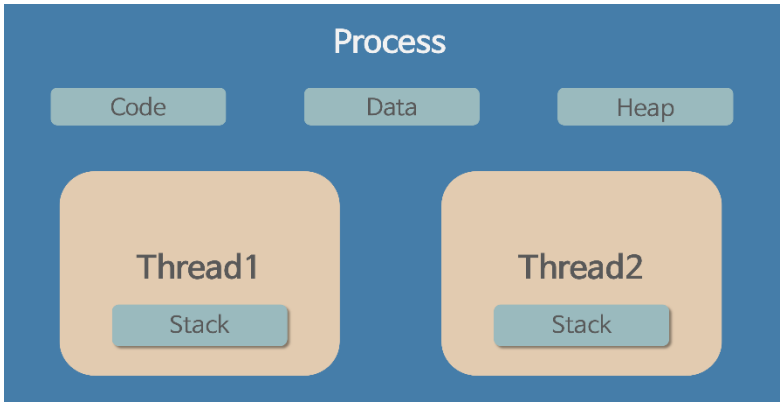
1. 프로세스는 각각 독립된 메모리 영역을 할당 받는다.

2. 기본적으로 프로세스당 최소 1개의 스레드를 가지고 있다.

3. 각 프로세스는 별도의 주소 공간에서 실행되며, 한 프로세스는 다른 프로세스의 변수나 자료구조에 접근 할 수 없다.

4. 한 프로세스가 다른 프로세스의 자언에 접근 하려면 프로세스 간의 통신을 사용해야 한다.

● **스레드**



1. 스레드는 프로세스 내에서 각각 Stack만 따로 할당받고 Code, Data, Heap 영역은 공유한다.

2. 스레드는 한 프로세스 내에서 동작되는 여러 실행의 흐름으로, 프로세스 내의 주소 공간이나 자원들을 같은 프로세스 내에 스레드끼리 공유하면서 실행된다.

3. 같은 프로세스 안에 있는 여러 스레드들은 같은 힙 공간을 공유한다. 반면에 프로세스는 다른 프로세스의 메모리에 직접 접근할 수 없다.

4. 각각의 스레드는 별도의 레지스터와 스택을 갖고 있지만, 힙 메모리는 서로 읽고 쓸 수 있다.

5. 한 스레드가 프로세스 자원을 변경하면, 다른 아웃 스레드도 그 변경 결과를 즉시 볼 수 있다

2) 멀티 프로세싱과 멀티 스레딩

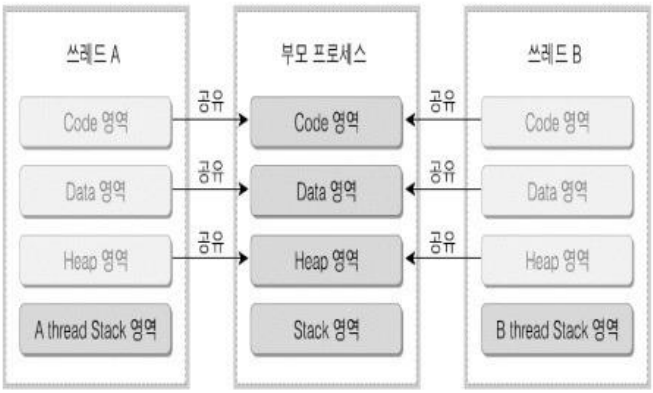
i) 멀티 프로세싱

a) 개념 및 구현방법

하나의 응용프로그램을 여러 개의 프로세스로 구성하여 각 프로세스가 하나의 작업(테스크)을 처리하도록 하는 것이다. 또한 멀티 프로세스 같은경우에 fork()을 통해 여러 개의 프로세스를 복사되며 메모리 공간을 공유하지 않아서 IPC(프로세스 간 통신)이 필요하다.

ii) 멀티 스레딩

a) 개념 및 구현방법



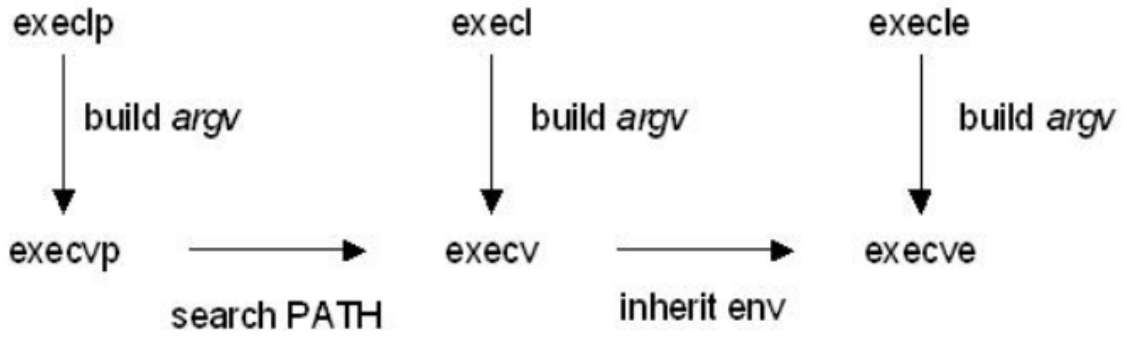
하나의 응용프로그램을 여러 개의 스레드로 구성하고 각 스레드로 하여금 하나의 작업을 처리하도록 하는 것이다. 위의 그림처럼 멀티 스레드는 스레드간에 코드, 데이터, 힙 영역을 공유한다. 따라서 스레드간의 통신이 프로세스간 통신보다 편하다.

3) exec 계열의 시스템 콜

#include<unistd.h>
int execl(const char *path, const char *arg0, ..., const char *argn, (char *)0); path에 지정한 경로명의 파일을 실행하며 arg0~argn을 인자로 전달한다. 관례적으로 arg0에는 실행 파일명을 지정한다. execl함수의 마지막 인자로는 인자의 끝을 의미하는 NULL 포인터((char*)0)를 지정해야 한다. path에 지정하는 경로명은 절대 경로나 상대 경로 모두 사용할 수 있다.
int execv(const char *path, char *const argv[]); path에 지정한 경로명에 있는 파일을 실행하며 argv를 인자로 전달한다. argv는 포인터 배열이다. 이 배열의 마지막에는 NULL 문자열을 저장해야 한다.
int execle(const char *path, const char *arg0, ..., const char *argn, (char *)0, char *const envp[]); path에 지정한 경로명의 파일을 실행하며 arg0~argn과 envp를 인자로 전달한다. envp에는 새로운 환경변수를 설정할 수 있다. arg0~argn을 포인터로 지정하므로, 마지막 값은 NULL 포인터로 지정해야 한다. Envp는 포인터 배열이다. 이 배열의 마지막에는 NULL 문자열을 저장해야 한다.
int execve(const char *path, char *const argv[], char *const envp[]); path에 지정한 경로명의 파일을 실행하며 argv, envp를 인자로 전달한다. argv와 envp는 포인터 배열이다. 이 배열의 마지막에는 NULL 문자열을 저장해야 한다.
int execlp(const char *file, const char *arg0, ..., const char *argn, (char *)0); file에 지정한 파일을 실행하며 arg0~argn만 인자로 전달한다. 파일은 이 함수를 호출한 프로세스의 검색 경로(환경 변수 PATH에 정의된 경로)에서 찾는다. arg0~argn은 포인터로 지정한다. execl 함수의 마지막 인자는 NULL 포인터로 지정해야 한다.
int execvp(const char *file, char *const argv[]); file에 지정한 파일을 실행하며 argv를 인자로 전달한다. argv는 포인터 배열이다. 이 배열의 마지막에는 NULL 문자열을 저장해야 한다.

그림. exec 계열 함수

exec계열 함수는 새 프로그램의 실행을 시작하는 데 사용됩니다. exec를 호출하는 프로세스는 새 프로그램(text, data, heap, stack)으로 완전히 대체되고 새 프로그램은 main ()에서 시작됩니다. 그리고 execve()는 커널내에서 유일한 system call이다. exec 계열의 함수를 사용하면 pid만 같고 안에 있는 내용이 다 바뀐 새로운 프로그램이 된다. 또한, 밑에 그림은 실제 수행 되는 것을 나타내는데, 모든 exec family는 결과적으로 execve를 call하게 됩니다.



4) IPC

IPC : 프로세스간 통신으로 프로세스들 사이에 서로 데이터를 주고 받는 행위 또는 그에 대한 방법이나 경로를 뜻한다.

1)PIPE : 익명의 PIPE를 통해서 동일한 PPID를 가진 프로세스들 간에 단방향 통신을 지원한다.

2)Named PIPE : 이름을 가진 PIPE를 통해서 프로세스들 간에 단방향 통신을 지원한다.

3)Message Queue : 구조체 기반으로 통신하고 메모리를 사용하는 PIPE이다.

4)Shared Memory : 시스템 상의 공유 메모리를 통해 통신한다.

5)Memory Map : 파일을 프로세스의 메모리에 일정 부분 맵핑 키셔 사용한다.

5) 동기화 (synchronization)

여러 프로세스의 접근 가능한 공유된 데이터에 여러 프로세들이 동시에 접근하여 데이터 연산이 잘못하게 될 경우가 있다. 이러한 가능성을 배제하기 위해서 critical section을 만들고 프로세스는 이 영역에 들어가기 위해서는 허가가 필요하다. 이러한 이유는 crtical section 안에 오직 하나의 프로세스만 이 존재하여 경쟁문제를 없애기 위해서이다.

6) 참조 문헌

프로세스와 스레드 https://gmlwjd9405.github.io/2018/09/14/process-vs-thread.html

https://includestdio.tistory.com/6

리눅스에서의 프로세스와 스레드 https://talkingaboutme.tistory.com/499

PCB https://jwprogramming.tistory.com/16, http://www.ques10.com/p/24811/short-note-on-process-control-block/

Execvp https://bbolmin.tistory.com/35, https://channelofchaos.tistory.com/55

멀티 프로세스 https://www.joinc.co.kr/w/Site/system_programing/Book_LSP/ch05_Process

IPC https://doitnow-man.tistory.com/110

동기화 https://l2men.tistory.com/16

chrono https://jacking.tistory.com/988

c언어로 스레드 구현 https://bitsoul.tistory.com/157

공유메모리 http://blog.naver.com/PostView.nhn?blogId=akj61300&logNo=80126318285

Time관련 사이트 https://modoocode.com/122, https://www.joinc.co.kr/w/man/2/gettimeofday

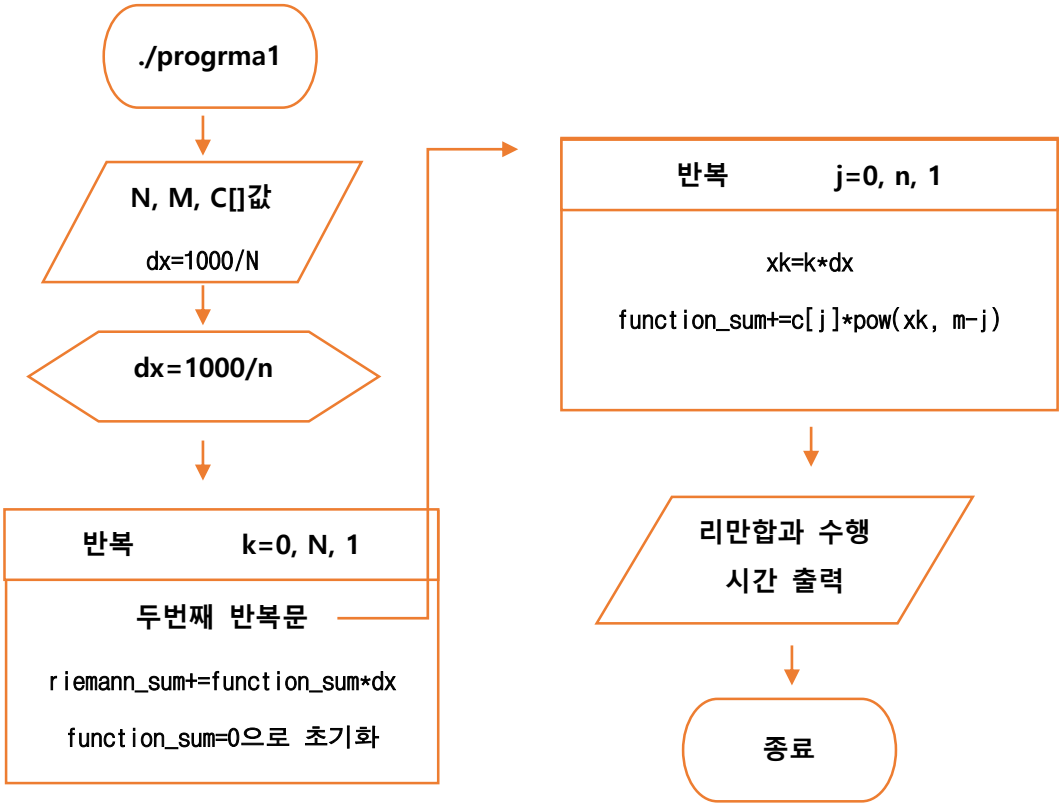
https://hashcode.co.kr/questions/404/c%EC%97%90%EC%84%9C%EB%8A%94-%ED%98%84%EC%9E%AC-%EB%82%A0%EC%A7%9C%EB%9E%91-%EC%8B%9C%EA%B0%84%EC%9D%84-%EA%B5%AC%ED%95%98%EB%A0%A4%EB%A9%B4-%EC%96%B4%EB%96%BB%EA%B2%8C-%ED%95%98%EB%82%98%EC%9A%94

오에스 강의노트

2. 프로그램밍 수행 결과 보고서

1) 작성한 프로그램의 동작 과정과 구현 방법

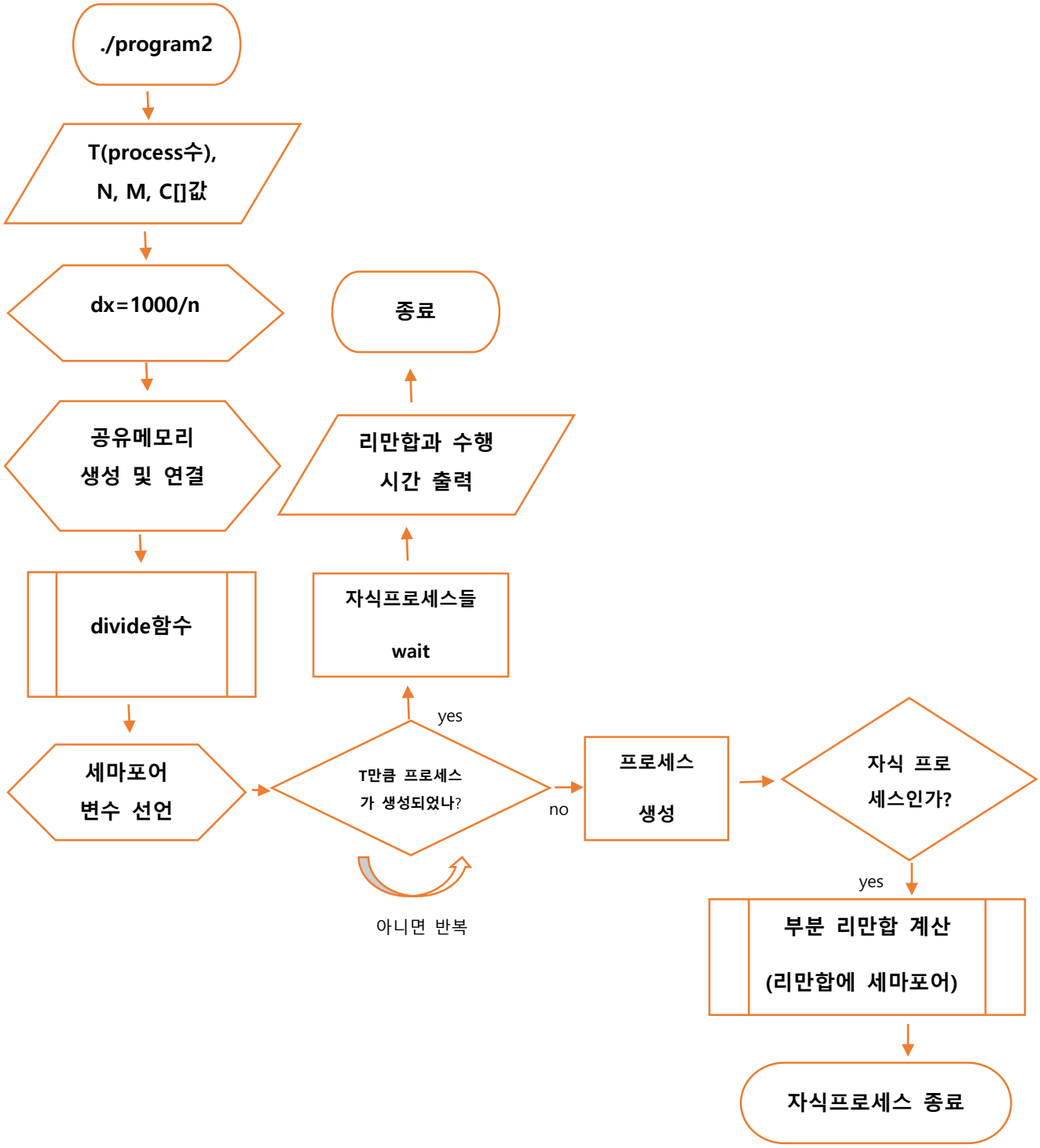
a) program1(no parallel)



리만합을 구하기 위한 기본적인 알고리즘이다. N, M, C[]값은 argv방식으로 입력을 받은 다음 델타x(dx)값은 1000에서 0까지로 주어졌으니 1000/N을 해서 구한다. 이제 이러한 리만합을 구하는 함수를 '**Riemann**'이라고 정의할 것이다.

```
gettimeofday(&start, NULL); //check start_time
dx=(long double)1000/n; //delta x=1000/n
for(int k=0; k<n; k++)
{
	for(int j = 0; j < m+1; j++) // m반복
	{
		xk=(long double)k*dx; // xk값
		function_sum+=(long double)arg[j+2]*pow(xk, m-j); //arg[j+2]는 cm을 의미
	}
	riemann_sum+=(long double)function_sum*dx; // 리만합에 사각형 하나값 저장
	function_sum=0; //function_sum 초기화
}
gettimeofday(&end, NULL); //check end_time
total_time_ms=1000*((long)end.tv_sec-(long)start.tv_sec)+((long)end.tv_usec-(long)start.tv_usec)/1000; // 시간을 ms단위로 total_time_ms에 저장
```

b) program2(Multi Process)



```
int divide[total_ps]; //각 프로세스당 몇번째 사각형까지 했는지 저장하는 배열
int d_r=n%total_ps; //나머지
int d_q=n/total_ps; //몫
divide[0]=0; //첫번째값 0으로 초기화
if(d_r==0) { //나머지가 0이면 균등하게 분배
        for(int i=1; i<total_ps+1; i++)
        {
                divide[i]=d_q;
        }
}
else { //나머지가 있으면 그 나머지만큼은 사각형을 몫+1만큼 설정하고 그 뒤에는 몫만큼 설정
        for(int i=1; i<d_r+1; i++)
        {
                divide[i]=d_q+1;
        }
        for(int i=d_r+1; i<total_ps+1; i++)
        {
                divide[i]=d_q;
        }
}
for(int i=1; i<total_ps+1; i++) { // 지금 내가 몇번째 사각형이 있는지 계산
        divide[i]=divide[i]+divide[i-1];
}
```

divide 함수 코드

위의 divide 함수 코드로 먼자 각 프로세스가 수행하는 사각형의 개수를 계산할수 있게 해준다. divide[현재프로세스+1]-divide[현재 프로세스]을 계산하면 현재프로세스가 수행해야 할 사각형의 개수가 나온다.

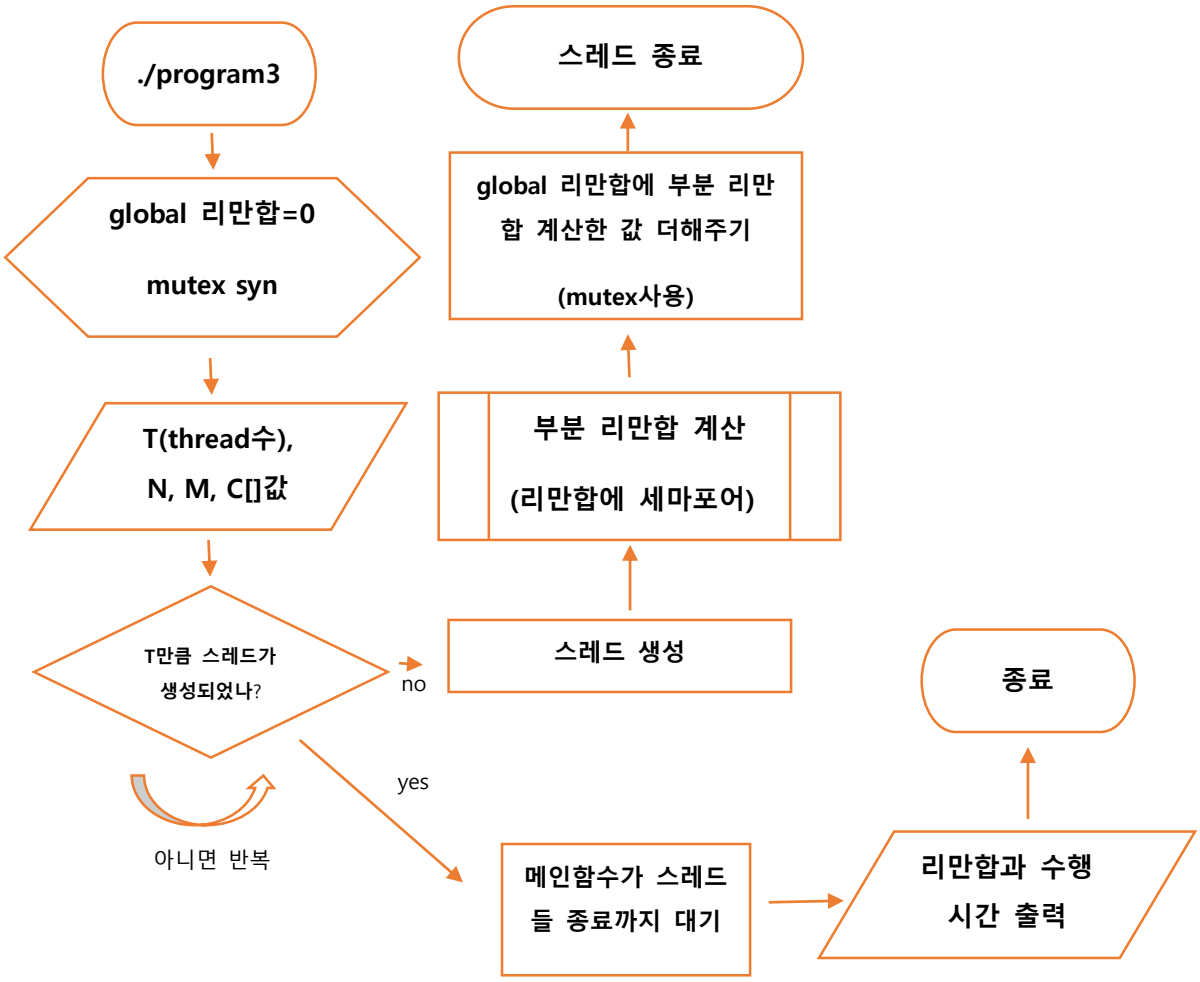
```
sem_t *syn; //세마포어 변수 선언
if((syn=sem_open("syn",O_CREAT,0777,1))==NULL) { //선언실패시 에러메시지 출력
        perror("Sem_open error");
        exit(1);
}
```

```
while(run_ps<total_ps) { //생성할 process개수만큼 반복
        pids[run_ps] = fork(); //create process
        if(pids[run_ps] < 0) { //fork fail
                perror("fork failed");
                exit(EXIT_FAILURE);
        }
        else if(pids[run_ps] == 0) { //child

                for(int k=divide[run_ps]; k<divide[run_ps+1]; k++) {
                        for(int j = 0; j < m+1; j++) {
                                xk=(long double)k*dx; // xk값
                                function_sum+=(long double)arg[j+3]*pow(xk, m-j); //arg[j+2]는 cm을 의미
                        }
                temp_sum+=(long double)function_sum*dx;
                function_sum=0;
                }
                sem_wait(syn); //세마포어 시작
                *riemann_sum+=temp_sum; // 공유메모리에다가 값저장
                sem_post(syn); //세마포어 종료
                sem_close(syn); //세마포어 없애기
                shmdt(riemann_sum); // 프로세스가 끝나면 공유메모리와 접속끊기
                _exit(EXIT_SUCCESS); //자식프로세스 종료
        }
        else { //parent
        }
        run_ps++; //현재 생성한 프로세스개수 체크
}
for(int i=0; i<total_ps; i++) {
        wait(&status); // 부모프로세스가 자식프로세스 갯수만큼 다끝날때까지 기다리기
}
```

위의 코드는 위의 순서도를 코드로 구현한 것이다. fork로 자식프로세스를 생성하고 fork한 값이 0이면 자식프로세스 0보다 큰 수 이면 부모 프로세스이다. wait을 한 이유는 자식프로세스보다 부모프로세스가 먼저 끝나면 자식프로세스에서 더한 리만부분합이 다 안 더해질 수 있기 때문에 wait함수를 썼다. 또한 race condition문제를 해결하기 위해 세마포어를 사용하였다.

c) program3(Multi Thread)

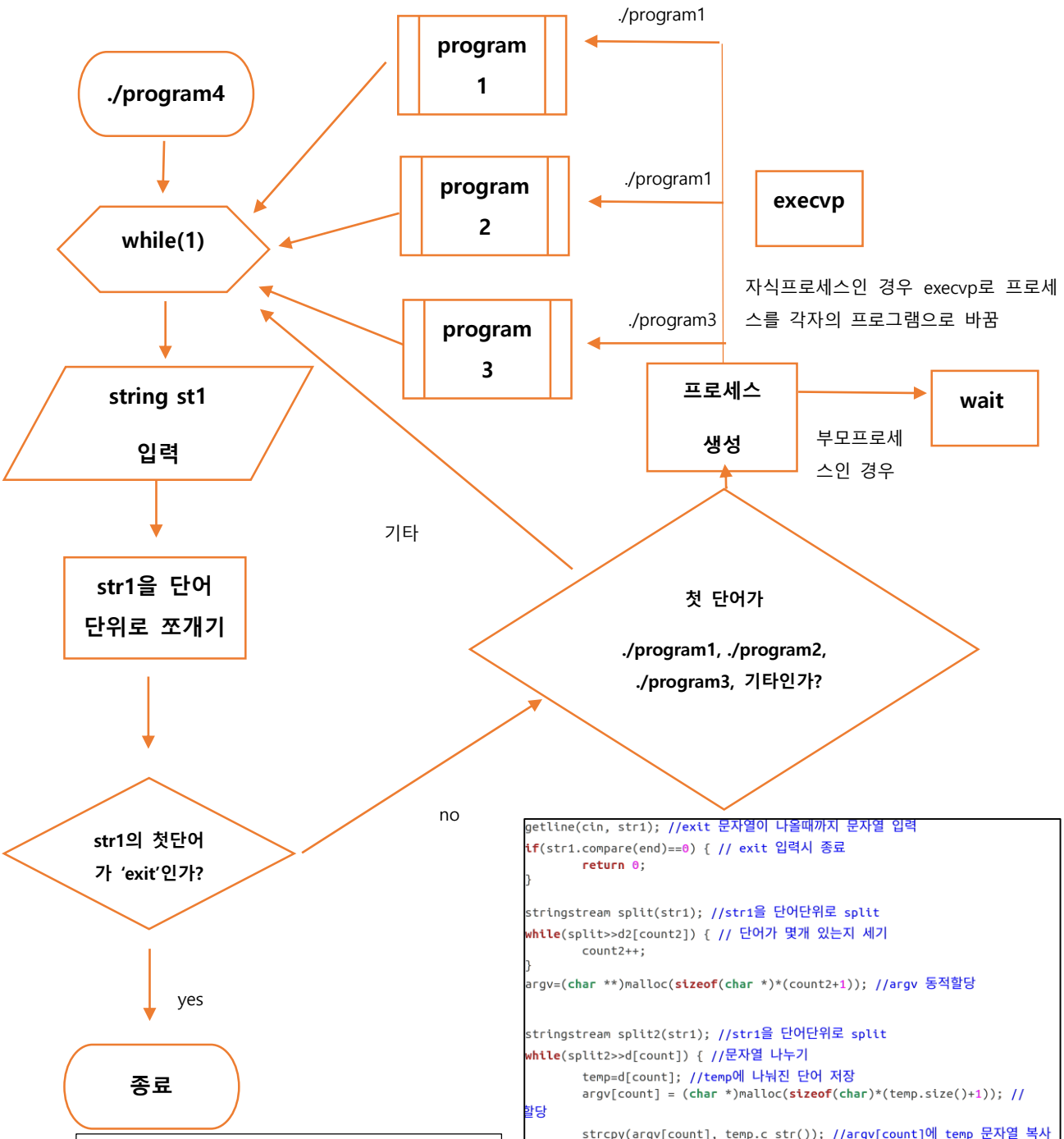



```
while(run_thread<total_thread) {
        thr_id=pthread_create(&thread_handle[run_thread], NULL, riemann, (void *)run_thread);
        if(thr_id != 0) { //thread_handle[run_thread]값이 0보다 작으면 생성실패
                perror("thread create failed");
                exit(EXIT_FAILURE);
        }
        run_thread++; // run_thread값을 스레드 생성시마다 증가
}

for(int i=0; i<total_thread; i++) { // 생성된 pthread가 종료되기를 기다림
        thr_id=pthread_join(thread_handle[i], (void **)&thread_result);
}
```

위의 코드는 pthread_create로 스레드 생성하는 함수와 pthread_join으로 메인함수가 스레드가 끝날때까지 기다린다. 또한 동기화 문제를 해결하기 위해서 mutex을 사용하였다.

d) program4(execvp)



split 함수로 string에서 단어를 구분하고 그 단어를 char형태의 배열에 저장한다. 그 다음에 입력할때마다 fork을 이용하여 프로세스를 생성하고 execvp 함수를 통해 새로 생긴 프로세스가 program1,2,3으로 바꾸어준다. exit을 입력시 프로그램이 종료된다.

```
getline(cin, str1); //exit 문자열이 나올때까지 문자열 입력
if(str1.compare(end)==0) { // exit 입력시 종료
        return 0;
}

stringstream split(str1); //str1을 단어단위로 split
while(split>>d2[count2]) { // 단어가 몇개 있는지 세기
        count2++;
}
argv=(char **)malloc(sizeof(char *)*(count2+1)); //argv 동적할당


stringstream split2(str1); //str1을 단어단위로 split
while(split2>>d[count]) { //문자열 나누기
        temp=d[count]; //temp에 나눠진 단어 저장
        argv[count] = (char *)malloc(sizeof(char)*(temp.size()+1)); //
할당
        strcpy(argv[count], temp.c_str()); //argv[count]에 temp 문자열 복사
        count++;
}
argv[count2] = NULL; //argv[] 끝에 값에 NULL값 저장

pid=fork(); //create process
if(pid<0) { //fork fail
        perror("fork failed");
        exit(EXIT_FAILURE);
}
else if(pid==0) { //child process
        execvp(argv[0], argv); //program1-3 실행
}
else { //parent process
        wait(&status); //wait child process
}
```

while문안에 있는 program4

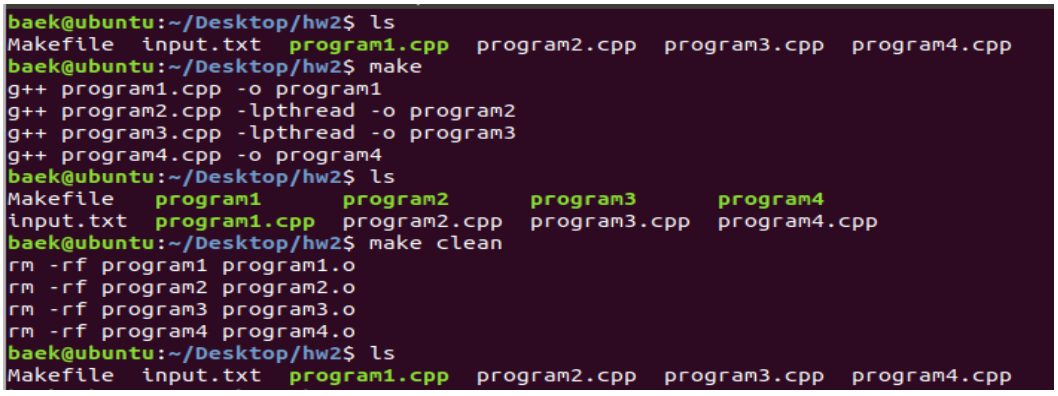
2) 작성한 Makefile에 대한 설명

```
all : program1.o program2.o program3.o program4.o

program1.o : program1.cpp
	g++ program1.cpp -o program1
program2.o : program2.cpp
	g++ program2.cpp -lpthread -o program2
program3.o : program3.cpp
	g++ program3.cpp -lpthread -o program3
program4.o : program4.cpp
	g++ program4.cpp -o program4

clean :
rm -rf program1 program1.o
rm -rf program2 program2.o
rm -rf program3 program3.o
rm -rf program4 program4.o
```

Default로 all로 설정하고 실행파일 program1-4까지 모두 생성한다. Program2에서는 세마포어 program3에서는 pthread함수 때문에 lpthread 옵션을 붙인다. -o 옵션은 프로그램을 실행 시 ./a.out이 아닌 ./program(1-4)로 실행하려고 한 것이다. 현재 c++언어로 코드를 짜서 g++ 옵션을 사용하였고 마지막으로 clean 명령어는 생성시켰던 실행파일들을 모두 지워주는 역할이다. 'rm'은 파일이나 디렉토리 삭제 옵션이고 -r 옵션은 하위 디렉터리를 포함하여 모든 내용을 삭제한다는 뜻이고 -f 옵션은 강제로 파일이나 디렉토리를 삭제하고, 삭제할 대상이 없을 경우 메시지를 출력하지 않는다는 옵션이다.



3) 개발 환경 명시

a) uname –a

```
baek@ubuntu:~$ uname -a
Linux ubuntu 4.19.27-2015147574 #1 SMP Thu Mar 14 11:01:30 PDT 2019 x86_64 x86_6
4 x86_64 GNU/Linux
```

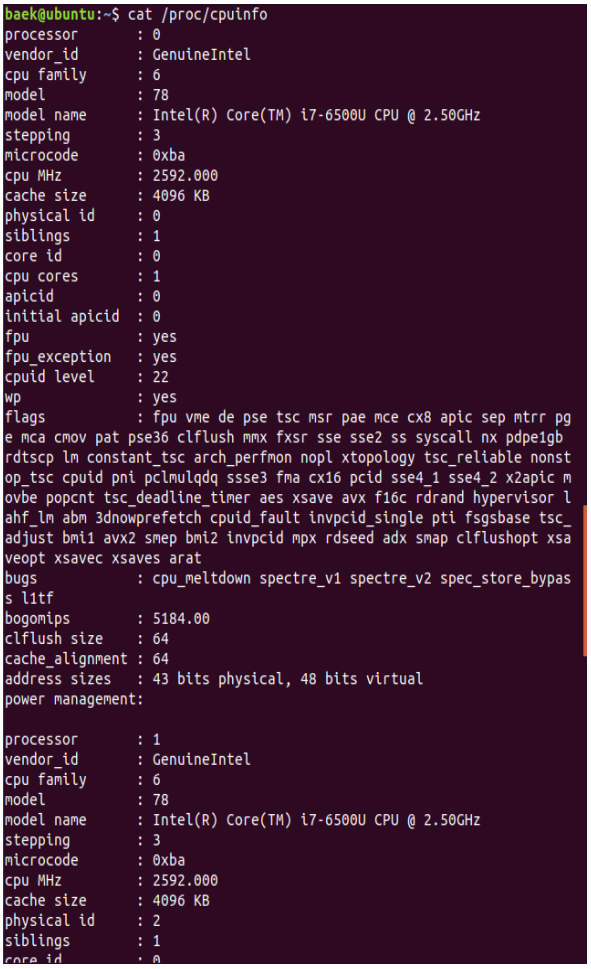
b) CPU, 메모리 정보

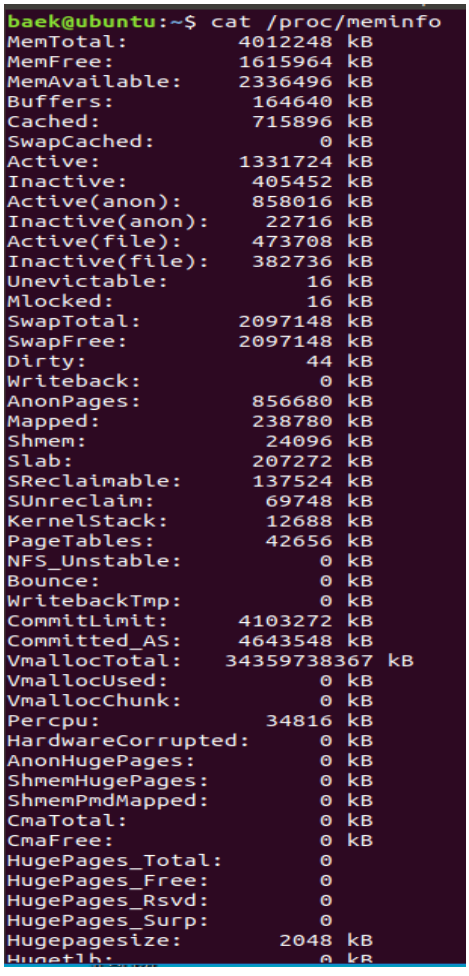
```
baek@ubuntu:~$ grep -c processor /proc/cpuinfo
4
```

cpu 코어 전체 개수: 4

```
baek@ubuntu:~$ cat /etc/issue
Ubuntu 18.04.2 LTS \n \l
```

OS : Ubuntu 18.04.2 LTS





왼쪽 그림이 cpu정보, 오른쪽 그림이 메모리정보

4) 과제 수행 중 발생한 애로사항 및 해결방안

a) Race Condition(program2, program3)

```
baek@ubuntu:~/Desktop/hw2$ ./program2 200 2000000 1 1 1
Riemann Sum: 500999.7500
Total Time: 122
baek@ubuntu:~/Desktop/hw2$ ./program2 200 2000000 1 1 1
Riemann Sum: 496082.2513
Total Time: 120
baek@ubuntu:~/Desktop/hw2$ ./program2 200 2000000 1 1 1
Riemann Sum: 500999.7500
Total Time: 123
```

```
sem_wait(syn);
*riemann_sum+=temp_sum; // 공유메모리에다가 값저장
sem_post(syn);
sem_close(syn);
```

위의 그림처럼 program2(program3도 마찬가지였다)을 여러 번 돌렸을 때 공유메모리에 동시에 접근하여서 값이 몇 개가 안 더해져서 값이 달라지는 경우가 생겼다. 프로그램 3은 동시에 전역변수에 접근하여 문제가 생겼다. program2은 위에 코드처럼 세마포어를 사용하여 동기화 시켜주어 race condition을 해결하였다

```
syn.lock(); //mutex lock
riemann_sum+=temp; // 글로벌 데이터에 저장
syn.unlock(); //mutex unlock
```

progrma3은 위에 그림처럼 mutex라는 동기화 기법을 사용하여 race condition 문제를 해결하였다.

b) program3에서 변수 사이즈

```
program3.cpp: In function ‘int main(int, char**)’:
program3.cpp:89:87: warning: cast to pointer from integer of different size [-W
int-to-pointer-cast]
 =pthread_create(&thread_handle[run_thread], NULL, riemann, (void *)run_thread)
;
```

처음에 run_thread 변수를 int형을 선언하고 넘겨주니 저런 warning 메시지가 떴다. run_thread을 unsigned long형태로 바꾸니 waring 메시지가 없어졌다.

5) 결과 화면과 결과에 대한 토의 내용

a) Process 와 Thread 생성 시간 차이

```
auto start=chrono::high_resolution_clock::now();
pids[run_ps] = fork();
auto end=chrono::high_resolution_clock::now()-start;
create_time=chrono::duration_cast<chrono::microseconds>(end).count();
create_avg+=create_time;
```

chrono을 이용하여 스레드와 프로세스 생성시간의 평균을 위와 같이 fork함수와 ptrhead_create함수를 하기 전과 하기 후의 차이을 구하였다. 나중에 create_avg는 총 프로세스 수 아니면 총 스레드 수만큼 나누어서 계산하였다. 프로세스와 스레드는 각각 5 개씩 생성하는 것을 5번씩 반복하여 계산하였다. 단위는 microsecond이다.

	Thread	Process
1번째	74	380
2번째	48	375
3번째	122	355
4번째	80	350
5번째	91	353
평균	83	362.6

. => 위의 표와 같이 프로세스 생성시간이 스레드 생성시간보다 많음을 알 수 있다. 스레드리는 프로세스 안에서 이미 공유하는 메모리가 있기 때문에 생성시간이 덜 걸린다.

b) 프로그램 수행결과 비교

i) n(리만합 사각형 개수)에 따른 수행시간 차이

프로세스와 스레드수는 4개로 고정한다. n뒤에 입력값은 1 1 1로 하였다. 모든 값은 5번 반복 하여 평균값을 구하였다. 시간은 ms단위이다.

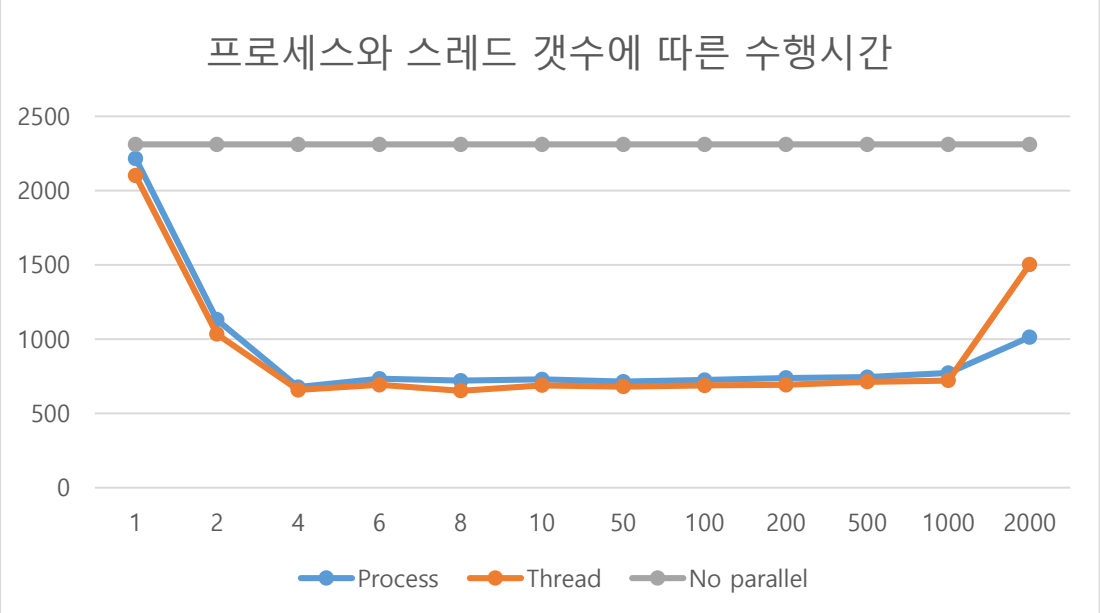
n	no parallel	Process(4개)	Thread(4개)
200000	38	18	10
2000000	239	84	73
20000000	2311	686	642
200000000	22226	6856	6467

⇨ 수행속도는 no parallel<Process<Thread이다. 내 논리코어수가 4개이니 프로세스와 스레드로 구현하는 것이 병렬로 구현안한것보다 빠르다. 또한 프로세스가 스레드보다 context switching이 많이 일어나기 때문에 속도가 스레드보다 느려진 것으로 생각된다..

ii) 프로세스와 스레드 개수에 따른 수행시간 차이

n을 20000000으로 고정하고 n뒤에 입력값은 1 1 1로 하였다. 모든 값은 5번 반복 하여 평균값을 구하였다 no parallel에서 걸린시간은 2311ms이다.

개수	Process	Thread
1	2216	2101
2	1131	1033
4	677	657
6	733	692
8	720	652
10	728	688
50	713	680
100	724	687
200	738	692
500	743	712
1000	771	721
2000	1013	1502



⇨ 4개전까지는 프로세스와 스레드 개수 차이에 따라서 급격하게 수행시간이 줄어든 반면에 내 cpu 논리코어가 4개라서 그 이후에는 시간이 비슷하다. 오히려 프로세스와 스레드 수가 증가함에따라서 생성시간과 context switching양이 많아져서 시간이 증가함을 볼 수있다. 2000개일때는 스레드가 급격하게 수행시간이 많아짐을 볼 수 있다. 스레드보다 훨씬 앞서서 말이다. 이러한 흥미로운 결과는 아마 스레드가 일정 개수이상 생성시 오버헤드가 더 많이 발생하여 이러한 결과가 보여준 것 같다.

iii) m에 따른 수행시간 차이

process와 thread수를 4개로, n을 2000000으로, c값들을 다 1로 고정하고 m값을 변화시킨다.

M값	No parallel	Process	Thread
1	239	84	73
5	1894	556	542
10	5532	1559	1545

⇨ mr값에 따라서 시간이 증가함을 볼 수 있다. 계산과정이 복잡해지면서 for문을 반복하는 횟수가 많아져서 수행시간이 오래걸린다. 또한 이것 역시 프로그램 수행속도는

no parallel<process<thread이다. 이유는 첫번째 표 분석과 똑같다.