arch/x86/kernel/syscall\_table\_32.S 에 syscall 정의

(hw 1-1) Display the memory map of the following program (ex1.cpp). What are the starting addresses of the code, data, heap, stack segment of this program and how many pages each segment occupies? What is the address of main function, the addresses of the global variables and local variables?

ex1.cpp

#include <stdio.h>

int x;

int y[10000];

int main(){

int k;

int \*pk;

pk=new int;

printf(“ex1. &main:%p &x:%p &y:%p &y[9999]:%p &k:%p &pk:%p pk:%p\n”,

main,&x,&y,&y[9999],&k,&pk,pk);

for(;;); // to see memory map of this process

return 0;

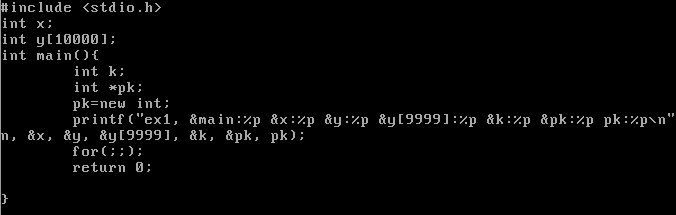
}

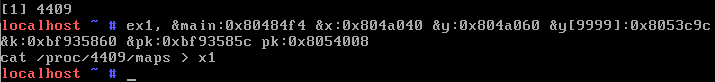
# g++ –o ex1 ex1.cpp

# ./ex1 &

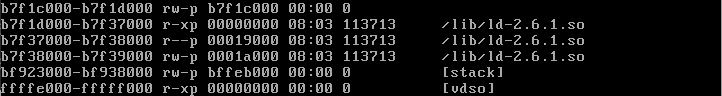
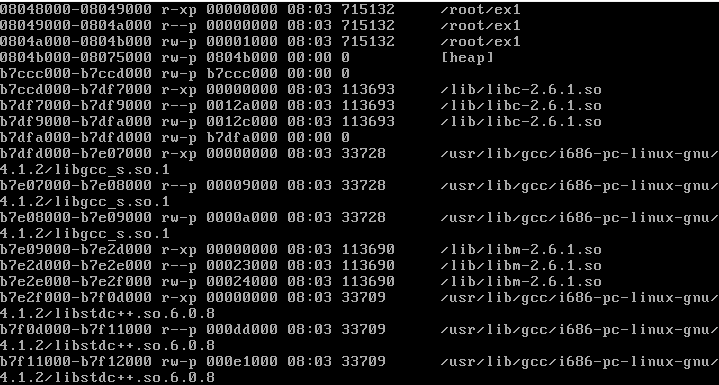
# cat /proc/(pid of ex1)/maps > x1

# vi x1





여기에서 y 배열은 크기가 크기 때문에 heap 영역까지 침범하게 된다. &pk에 대해서 출력한건 pk의 주소를 출력한거고, 그냥 pk를 출력한건 pk 안에 들어있는 int를 출력한 것이다.



starting address of code : 8048000, 1 page

starting address of data : 804a000, 1 page

starting address of heap : 804b000, 42 pages

starting address of stack : bf923000, 21 pages

address of main : 0x80484f4

addresses of global variables (x, y) : 0x804a040, 0x804a060

address of local variables (k, pk) : 0xbf935860, 0xbf93585c

(hw 1-2) Write another simple program, ex2.cpp (see below), and run ex2, ex1 at the same time. Confirm they have the same address for main function. How can they run at the same location at the same time?

#include <stdio.h>

int x1;

int main(){

int \*pk1;

pk1 = new int;

printf(“ex2. &main:%p &x1:%p\n”, main,&x1);

for(;;); // to see memory map of this process

return 0;

}

# g++ -o ex2 ex2.cpp

#./ex2 &

...........

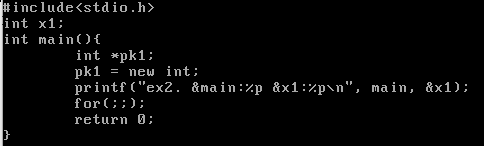
#./ex1 &

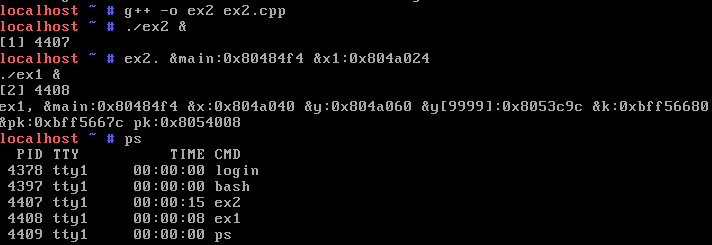
...........

# ps

.............. ex2

.............. ex1





ex1과 ex2를 동시에 실행해봤을 때, 두 프로세스의 main 주소는 0x80484f4로 똑같은걸 볼 수 있다. 이렇게 같은 메인 주소를 가지고 있는 두개의 프로세스가 동시에 실행될 수 있는 이유는 사실상 표시된 0x80484f4는 실제 주소가 아닌 process image에 들어있는 가상 주소이기 때문이다. 가상 주소는 같아도 실제 메모리에 들어있는 main의 주소는 ex2와 ex1이 다르기 때문에 실행에는 별다른 문제가 되지 않는다.

(hw 2) Show the memory map of the following program. Which pages does the program access during the run time? Show the page numbers that the program accesses in the order they are accessed. Indicate which pages are for code and which are for global data and which are for local data.

ex4.c

int A[5][1024];

int main(){

int i,j;

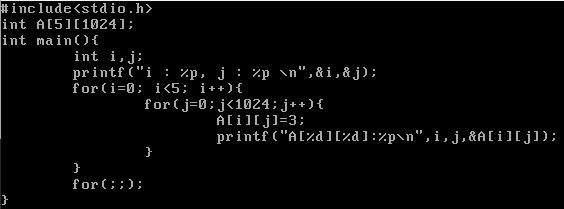
for(i=0;i<5;i++){

for(j=0;j<1024;j++)

A[i][j]=3;

}

ex4.c



어느 페이지에 접근하는지 보고 또 maps를 보기 위해서 중간중간 printf와 마지막에 for(;;);를 추가해주었다.





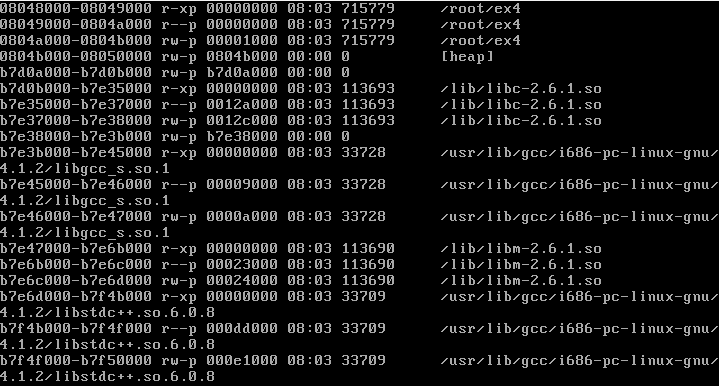


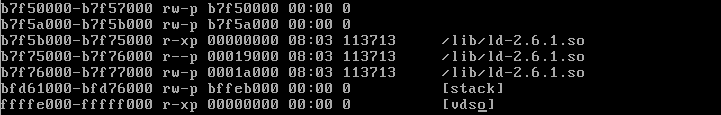












런타임 때 접근되는 페이지는 먼저 코드영역인 8048번이 접근되고, 다음 숙제에서 봤을 때 804a번도 미리 접근되는 것 같다. 그 다음에 i와 j가 코드안에 있기 때문에 스택영역인 bfd74번이 접근되고 배열 A에 접근하면서 804b, 804c, 804d, 804e, 804f번 페이지가 차례대로 접근되는 것 같다. 코드영역은 8048번에서 8049번까지 쓰고, global data는 804a부터 힙 영역인 8050번사이에 있게 된다. 마지막으로 local data는 스택 영역인 bfd61번부터 bfd76번 사이에 있게 된다.

hw 3) How many page faults will the program in hw 2) generate? Explain your reasoning. Remember in the beginning the system has no page of the current process in the memory.

804a040부터 A[0][0]가 쓰게 되고, 그 다음 804a080은 A[0][1], 이렇게 해서 804a번 페이지의 마지막인 804affc는 A[0][1007]이 쓰게 된다. A[0][1008]부터 다음 페이지인 804b번으로 넘어가게되고, 이 때 page fault가 일어나게 된다. 그렇기 때문에 A[0][0], A[0][1008], A[1][1008], A[2][1008], A[3][1008], A[4][1008] 에서 page fault가 일어나서 총 6번의 page fault가 일어난다고 추측해볼 수 있다.

hw 4) Confirm your answer in hw 3) by defining a new system call, sys\_show\_pfcnt(), in mm/mmap.c, which displays the number of page faults generated so far.

extern int pfcnt;

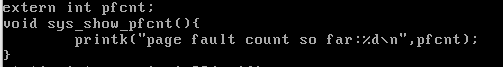
void sys\_show\_pfcnt(){

printk("page fault count so far:%d\n", pfcnt);

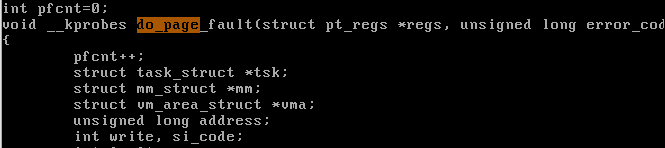
}

The "pfcnt" should be increased by one whenever there is a page fault. Remember a page fault will raise INT 14 which will be processed by page\_fault() in arch/x86/kernel/entry\_32.S, which in turn calls do\_page\_fault() in arch/x86/mm/fault.c. Define "int pfcnt=0" in this file and increase it inside do\_page\_fault(). Now you call this system call before and after the double loop in hw 2) and see the difference.

arch/x86/mm/mmap.c

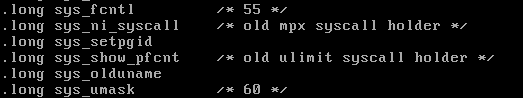


arch/x86/mm/fault.c



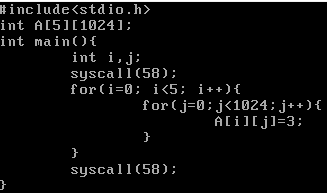
do\_page\_fault 함수가 불릴 때마다 pfcnt가 1 증가하도록 만들어줬다.

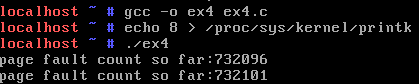
arch/x86/kernel/syscall\_table\_32.S



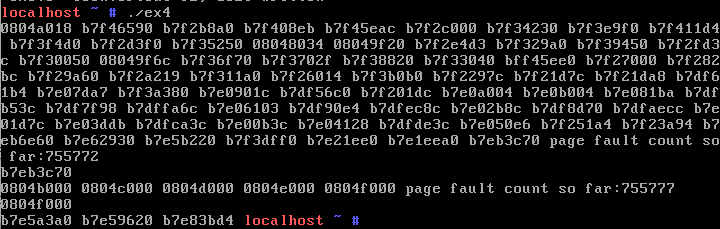
syscall table에서 58번에 sys\_show\_pfcnt를 정의해줬다.

ex4.c





hw3에서는 6번 일어날 것이라고 예측했지만, 실제로 보면 5번 일어났다는걸 알 수 있다.



커널을 수정해서 들여다본 결과, 맨 처음에 804a018이 찍혔기 때문에 804a번 페이지에서는 코드에서 syscall(58)이 불려지기 전에 이미 page fault가 발생했단걸 알 수 있었다. 따라서 실제로 확인해봤을 때는 804b, 804c, 804d, 804e, 804f번 페이지로 총 5번의 page fault가 일어났단걸 알 수 있다.

hw4-1) You can display the exact address where page fault has happened. Make ex3.c and insert following code (in italic) in arch/mm/fault.c:do\_page\_fault(). When you run ex3, the kernel will display the page fault addresses generated by ex3. Explain the result.

ex3.c:

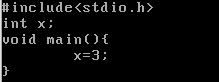
int x;

void main(){

x=3;

}

ex3.c



In kernel arch/mm/fault.c:

void do\_page\_fault(...........){

...........

/\* get the address \*/

address = read\_cr2();

*if (strcmp(tsk->comm, "ex3")==0){*

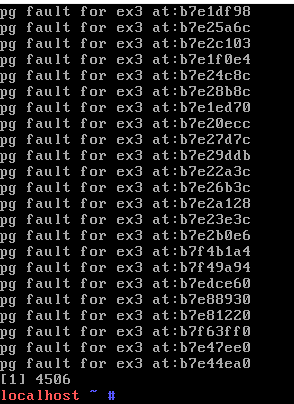
*printk("pg fault for ex3 at:%p\n", address);*

*}*

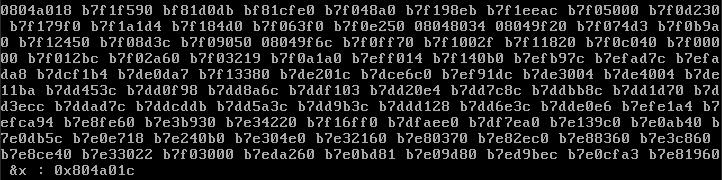
*..............*

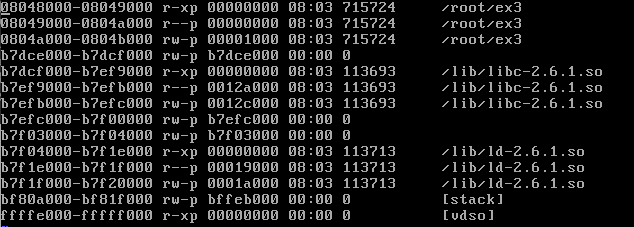
arch/x86/mm/fault.c/do\_page\_fault()

**



원래는 위와 같이 나오지만, 조금 더 자세히 보기 위해서 앞의 “pg fault for ex3 at:”를 빼고 ex3.c 끝에 for(;;);를 추가해서 실행결과와 map을 들여다보았다.





여러 번 시도를 해본 결과, 코드에 메인함수까지 없으면 위와 같은 주소들이 안찍히고 컴파일러는 ex3이 실행파일이란 인식을 아예 안한다. 메인 함수만 있어도 저 주소들이 그대로 찍히기 때문에 위의 주소들은 메인에 접근하면서 만들어진 것이라고 추측할 수 있다. 실행결과를 보면 페이지들 중에서도 주로 b7로 시작하는 페이지에 많이 접근하는데, 이는 메인이 만들어지면서 필요한 라이브러리를 메모리에 다 넣기 시작하기 때문인 것 같다.

hw4-2) Repeat hw4) with modified hw2) code as below. Why is pfcnt increased?

int A[5][1024];

int main(){

int i,j;

for(i=0;i<5;i++){

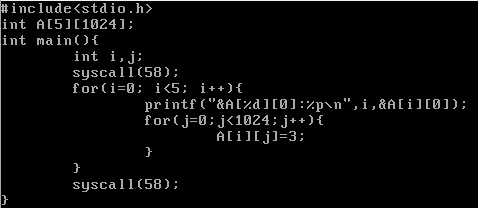
printf("&A[%d][0]:%p\n", i, &A[i][0]); // add this line

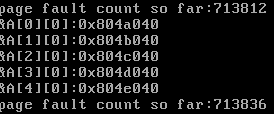
for(j=0;j<1024;j++)

A[i][j]=3;

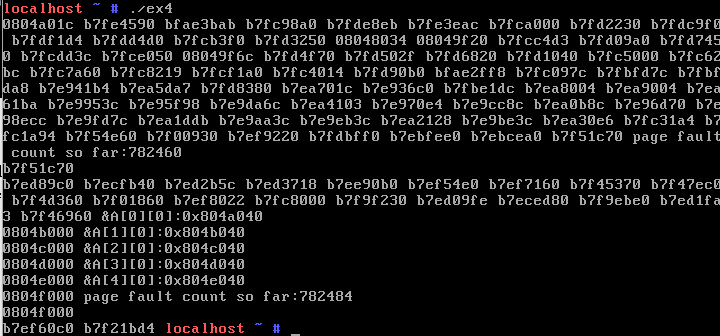
}

}





원래는 5번이였지만 지금은 총 24번의 page fault 일어났단걸 볼 수 있다.



자세히 보면 위 그림에서 b7ed89c0부터 b7f46960까지 과제 4-1번보다 19번의 page fault가 더 일어난걸 알 수 있다. 이는 printf가 호출되면서 필요한 라이브러리의 페이지를 메모리에 집어넣으면서 발생한 것으로 보여진다.

hw 5) Make a system call that prints vma information of the current process, and write a user program that displays the VMA list with it. Confirm that this result matches to those in /proc/xxxx/maps.

- Use system call 31 which is not used currently.

- Modify the system call table so that system call 31 is redirected to sys\_get\_VMAlist.

- Provide sys\_get\_VMAlist() in mm/mmap.c. This function will display all vma's of the current process.

struct vm\_area\_struct \*temp=current->mm->mmap;

for(;;){

if (temp==NULL) break; // we are done

display temp->vm\_start, temp->vm\_end,

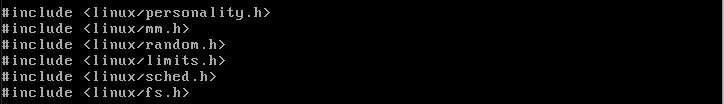
temp->vm\_file->f\_dentry->d\_name.name;

temp=temp->vm\_next;

}

- Write a user program (e.g. xx.c) that invokes system call 31

arch/x86/mm/mmap.c



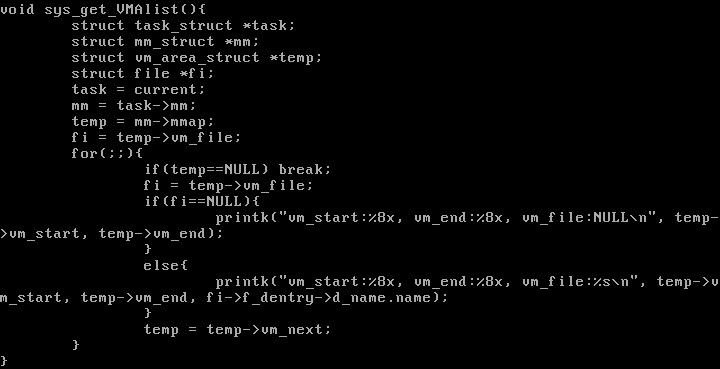
linux/fs.h헤더가 코드에 없으면 에러가 나기 때문에 이 부분을 코드에 추가해줬다.

struct vm\_area\_struct {

unsigned long vm\_start, vm\_end;

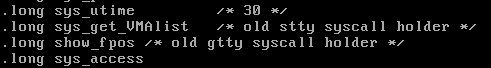
struct vm\_area\_struct \*vm\_next;

struct file \* vm\_file;

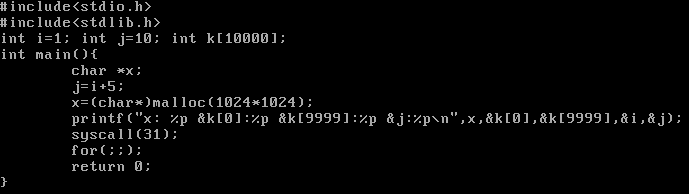


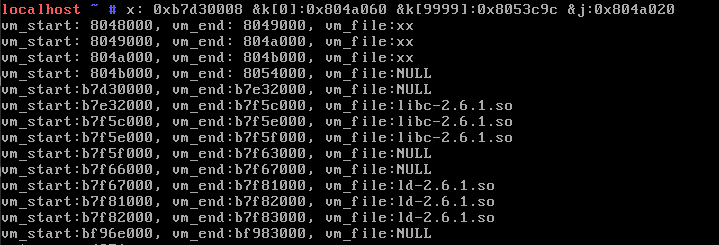
display 부분은 5번 문제의 코드를 그대로 쓰면 실행했을 때 에러가 나기 때문에 struct vm\_are\_struct \*temp=current->mm->mmap;를 몇단계로 나눠서 써줬다. segmantation fault가 일어나지 않도록 fi는 if문으로 적절히 처리해주었다.

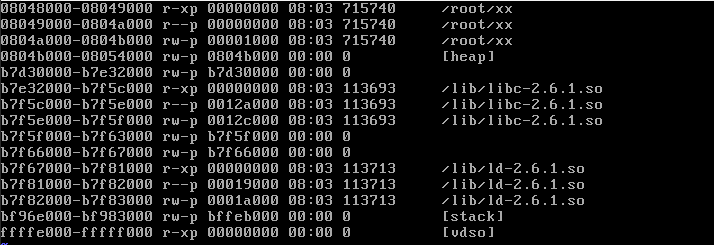
arch/x86/kernel/syscall\_table\_32.S



xx.c







실행결과를 maps와 비교해봤을 때, 실행파일에서 syscall(31)이 호출될 때 [vdso]는 출력이 안됐지만 나머지는 maps와 똑같이 나온걸 알 수 있었다.

hw 6) Count the number of page faults when you run following ex1 and ex2 by using sys\_show\_pfcnt(). Explain the results. Also compare the running time of each code (use gettimeofday() function) and explain why they differ. Run several times and compute the average.

double getUnixTime(){

struct timeval tv;

gettimeofday(&tv, (void \*)NULL); // get current time

return (tv.tv\_sec + tv.tv\_usec/1.0e6);// return it in seconds

}

……

double stime, etime, diff;

stime=getUnixTime(); // starting time

…… code …….

etime=getUnixTime(); // ending time

diff=etime-stime; // the difference

printf(“the elapsed time:%f\n”, diff);

ex1.c

#include <unistd.h>

#include <sys/time.h>

int A[8192][8192];

double getUnixTime(){

…………….

}

void main(){

int i,j;

call getUnixTime() and remember the stime…….

syscall(17); // display pfcnt

for(i=0;i<8192;i++){

for(j=0;j<8192;j++){

A[i][j]=3;

}

}

syscall(17); // display pfcnt again

call getUnixTime() and compute diff and print it…….

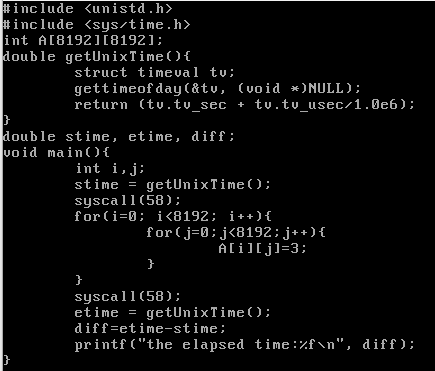
ex2.c

same as ex1.c except

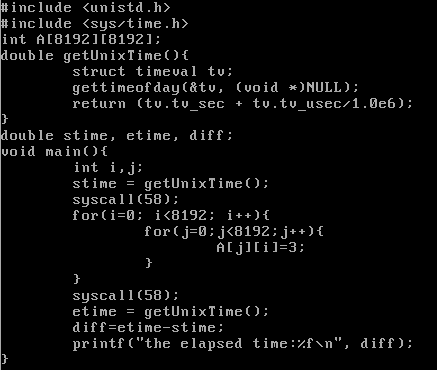
change A[i][j]=3; to A[j][i]=3;

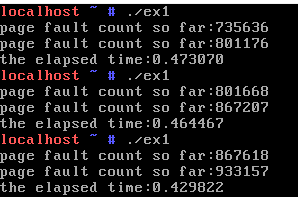
(If your vm dies, reduce the array size)

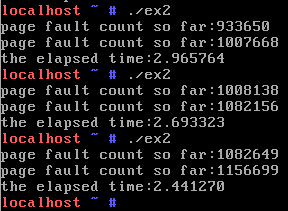
ex1.c



ex2.c







ex1에서는 row를 기준으로 차례대로 채우고 ex2에서는 column을 기준으로 차례대로 채우게 된다. row를 기준으로 채웠을 땐 예를들어서 A[0][0]이 들어왔다면 A[0][0]부터 A[0][8191] 까지는 다 메모리에 들어와있기 때문에 A[1][0]이 들어오기 전까지는 page fault가 안일어나게 된다. 이것을 row-wise라 하고, 만약에 ex2처럼 column-wise 방법을 쓰게 되면 A[0][0], A[1][0], A[2][0], … 순서대로 들어가게 되는데 이 때 A[0][0]과 A[1][0] 사이의 거리가 멀기 때문에 각각 다른 페이지에 들어가게 된다. 이것을 A[8191][8191]까지 넣으려면 page fault가 지속적으로 끊임없이 일어나게 된다. page fault가 일어나는 수를 계산해보면 ex1은 약 65539번 일어나고 ex2는 74018번 정도 일어나서 거의 만 페이지 차이가 난다. 이 page fault의 차이 때문에 ex1의 실행시간 평균은 0.455786초이지만 ex2의 실행시간 평균은 2.700119초로 약 6배정도 차이가 난다.