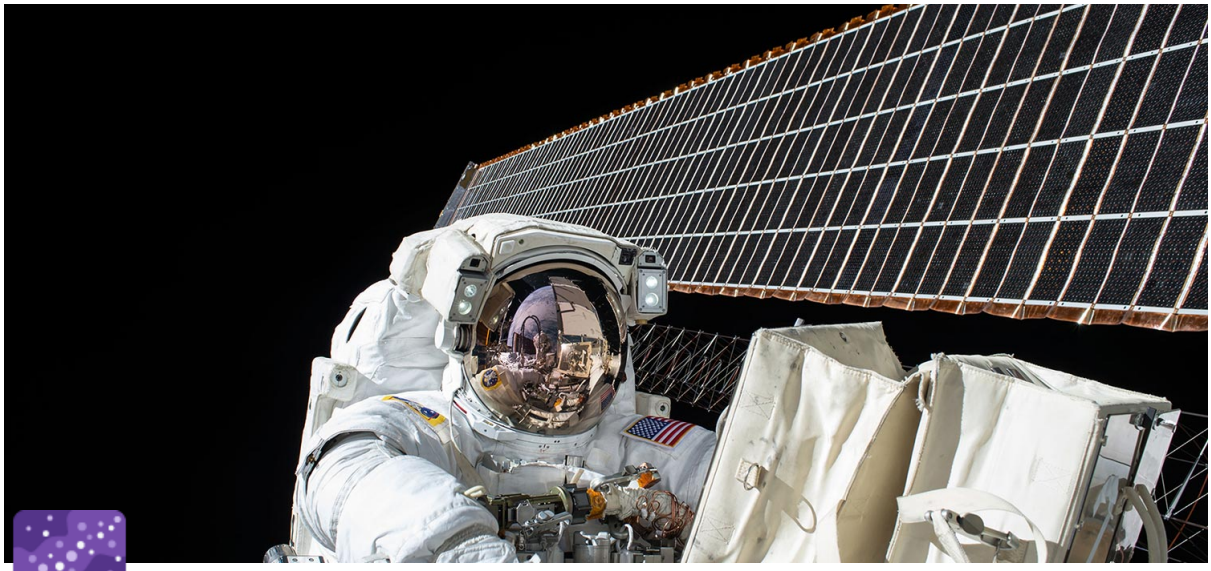


Finding Frequency Distribution

첫 번째 줄에는 값들의 수가 주어지고 (k)

두 번째 줄 부터 k번째 줄까지는

0~9999 중 임의의 양의 정수들이 주어진다.

Thread나 Process들을 통해서 주어진 간격의 크기(i)로

도수분포표를 만들어라.

간격의 크기란?

0 ~ 10

11 ~ 20

여기서 간격의 크기는 10이다.

eg) > ./ku_pfred n i input_file.txt

n: 사용해야 하는 쓰레드 / 프로세스들의 개수

i: 간격의 크기

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/7dddb6bd-aadb-427d-80a1-18c15046b53b/12_Assignment_2.pdf

대략적으로 제 코드는 이런 방식으로 작동이 됩니다.

1. main에서 파일 (dataset)을 연다.
2. 파일에서 숫자의 개수 (첫 줄)을 읽는다.
3. 주어진 쓰레드 / 프로세스 수와 숫자의 개수에 따라 읽어야 할 지점 (offset)을 쓰레드 / 프로세스마다 달리 설정해준다.
4. 각각의 쓰레드 / 프로세스들은 각자 주어진 offset에서 숫자의 개수 / 프로세스 수 혹은 쓰레드 수 만큼 읽는다.
    - 만약 자료의 개수가 10000개고 쓰레드 / 프로세스 수가 3개나 7개등 자료의 개수의 약수가 아니라면, 마지막 쓰레드 / 프로세스가 그 나머지만큼 더 자료를 처리한다.
5. 나중에 병렬적으로 읽은 정보들을 main에서 다 합친다.

물론 쓰레드와 프로세스는 다른 방식으로 자원을 공유하고 수정하기 때문에

살짝 차이가 나지만 두 방식 다 분할해서 동시에 나누고 합칩니다.

Multi-process program

Design & Implementation

POSIX Message Queue를 통해서 프로세서 간의 자원을 공유한다!

각 프로세스들은 4 바이트 (int) 짜리 메세지 하나 만을 가지는 메세지 큐를 가집니다.

(mq_maxmsg = 1, mq_msgsize = 4)

각 프로세스들마다 파일을 읽어야 하는 위치 offset이 주어지는데, 이는 프로세스의 개수와 자료의 개수에 따라 달라집니다.

0자료가 1000개이고 프로세스가 4개이면, 각각의 프로세스들은

10000 / 4개의 숫자들을 읽어야 합니다.

(만약 프로세스가 3, 7과 같은 10000의 약수가 아니라면 한 프로세스가 10000 % 3, 10000 % 7 씩

남은 만큼 더 읽어옵니다)

프로세스 0은 1번째 숫자 (a1) ~ 2500번째 숫자 (a2500),

프로세스 1은 2501번째 숫자 (a2501) ~ 5000번째 숫자 (a5000),

프로세스 2는 5001번째 숫자 (a5001) ~ 7500번째 숫자 (a7500),

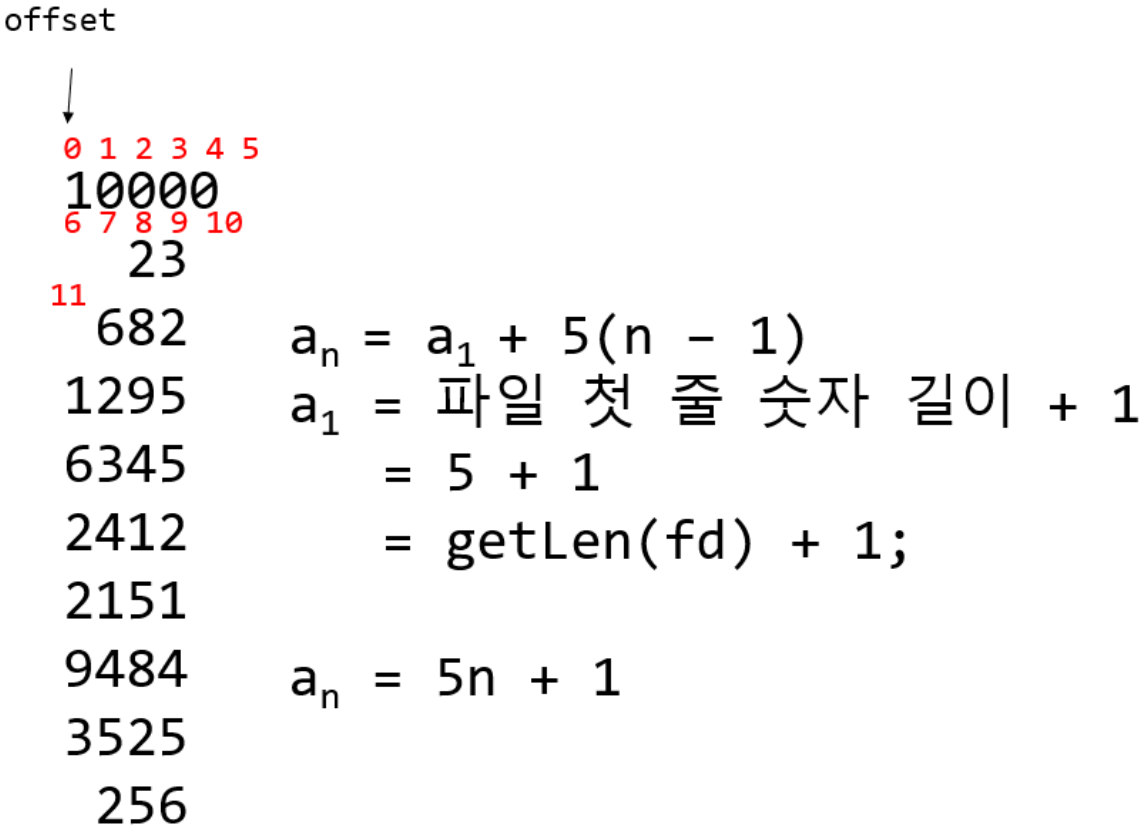
프로세스 3은 7501번째 숫자 (a7501) ~ 10000번째 숫자 (a10000) 까지 읽습니다.

pread 함수를 사용함에 따라, 몇 바이트씩 어디서 얼만큼씩 읽어야 하는지 명시해주어야 합니다.

기본적으로 숫자는 0부터 9999까지이니, 숫자는 4바이트씩 읽어주면 됩니다.

그렇다면 숫자를 어디서 부터 읽어야 하는지, 그 위치 - offset이 문제입니다.

offset은 이렇게 계산이 됩니다.



파일에는 10000개의 숫자들이 있는데, 특정 숫자들이 파일의 몇 번째에 있는지는

이 수열을 통해 알 수 있습니다.

파일의 n번째 숫자의 시작 위치는 공차가 5인 등차수열 an을 따릅니다.

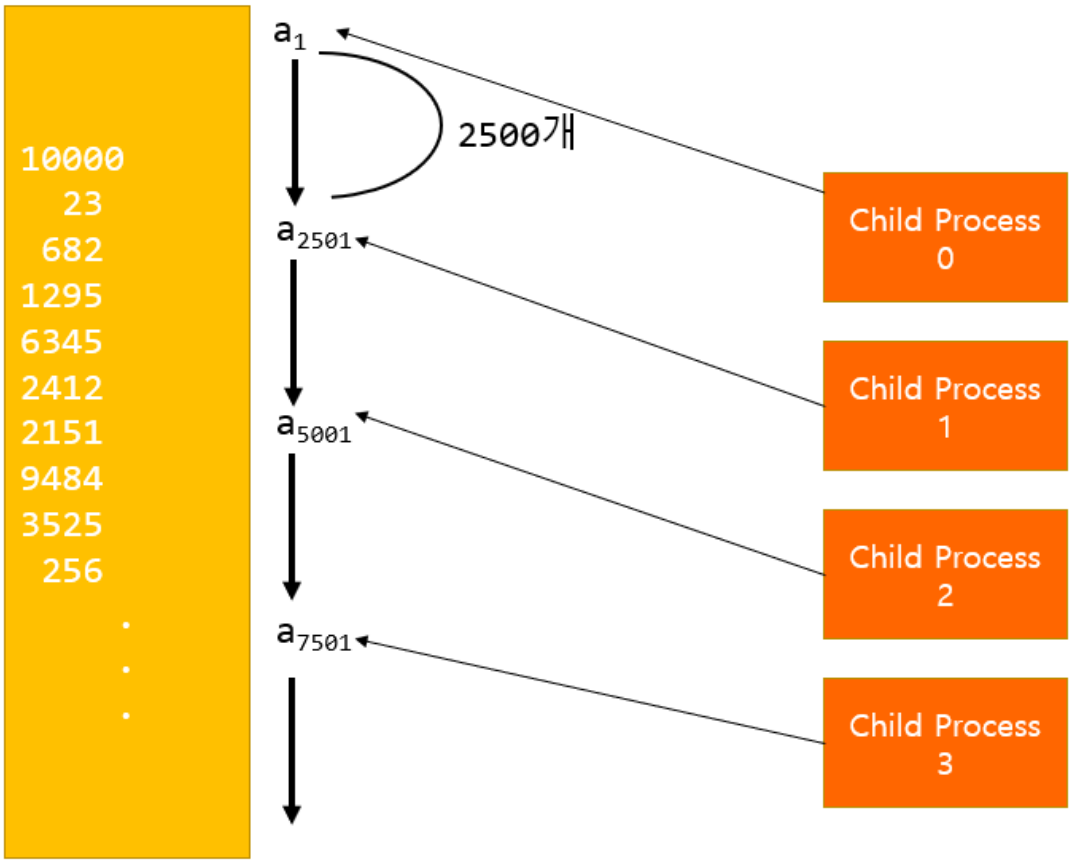
이 수열은 자료의 개수의 길이에 따라 결정됩니다.

이 경우에는 자료의 개수가 10000이니, 길이는 5가 됩니다.

이 수열의 초항은 길이 + 1이니, 초항 a1 = 6입니다.

따라서 이 경우에는 수열 an = 5n + 1이 되고, 첫 번째 숫자는 6, 두 번째 숫자는 11 ...이 나오는 것을

확인할 수 있습니다.



프로세스0은 1번째 숫자부터 2500개를 읽으니 그 프로세스에게는 a1 = 5 * 1 + 1 = 6을 주면 됩니다.

마찬가지로 프로세스1는 2501번째 부터 읽으니 a2501 = 5 * 2501 + 1 = 12506,

프로세스2는 a5001 = 5 * 5001 + 1 = 25006,

프로세스3은 a7501 = 5 * 7501 + 1 = 37506씩 주면 됩니다.

물론 프로세스들에게 분배될 이 offset은 int* offsetP라는 동적으로 할당된 배열이 가지고 있습니다.

이 offset들이 주어지면 각 프로세스들은 숫자 크기와 개행 문자를 포함한 5바이트씩 offset을 더해가며 2500개씩 읽으면 됩니다.

이렇게 각각 프로세스들은 2500개씩 읽게 됩니다.

프로세스들은 숫자들을 하나씩 메세지 큐에 넣고 부모 프로세스인 메인에게

계속해서 보냅니다.

메인은 받은 숫자들을 int* intArr에 저장합니다.

```
int interval = 1000

int* intArr = (int*) malloc(sizeof(int) * (10000 / interval))
```

0 ~ 999	1000 ~ 1999	2000 ~ 2999	3000 ~ 3999	4000 ~ 4999	5000 ~ 5999	6000 ~ 6999	7000 ~ 7999	8000 ~ 8999	9000 ~ 9999
0	0	0	0	0	0	0	0	0	0
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

```
intArr[value / interval]++
```

이 intArr은 구간의 크기 (interval)에 따라 배열 크기도 달라집니다.

파일에서 읽어온 숫자들 (value)은 value / interval로 구간을 결정해서

그 구간에 해당하는 숫자를 더해줍니다.

구간의 크기가 1000, 숫자가 2879라면

value / interval = 2879 / 1000 = 2, 즉 intArr[2]++가 됩니다.

프로세스로 구현한 방식에서는 자식 프로세스들이 sender, senderX란 함수로 파일을 읽어서

숫자들을 계속 보내면, 메인에서는 receiver, receiverX로 메세지 큐들을 읽어서

intArr에 구간 내 숫자들을 세어주고 다 읽고난 후 메세지 큐들을 unlink해줍니다.

프로세스 reaping은 새로 만든 handler, reaper라는 함수가 SIGCHLD 시그널을 받았을 때 계속

reaping해 줍니다.

Concurrent Programming에 관한 자료를 찾아보고 공부하다가 알게되었습니다.

https://www.cs.cmu.edu/~213/lectures/24-concprog.pdf

원래는 main에서 다 반복문으로 reaping해줄 생각이었지만 이렇게 reaping 해줄 수도 있구나를 새롭게 알게 되었습니다.

Documentation

Function Name	Arguments	Description	Return Value

Function Name	Arguments	Description	Return Value
reaper	int sig	프로세스 reaping용 handler.	void
sender	int fd, int size, int interval, int numP, int offset, char* mqName	File Descriptor, 자료 개수, 구간 크기, 프로세스 개수, 파일 읽을 위치, 메세지 큐 이름을 받아와서 메세지큐에 읽은 숫자를 계속 보낸다.	void
senderX	int fd, int size, int interval, int numP, int remainder, int offset, char* mqName	sender와 같은 인자들이지만 remainder까지 더 받는다! 남은 숫자들까지 더 읽고 메세지 큐에 보낸다.	void
receiver	int size, int interval, int numP, int* intArr, char* mqName	메세지 큐를 열고 숫자들을 계속 intArr에 세어주며 다 읽고난 후 메세지 큐를 지운다.	void
receiverX	int size, int interval, int numP, int remainder, int* intArr, char* mqName	receiver와 같은 역할을 하지만 remainder만큼 더 읽어온다.	void
nameInc	char* str	메세지 큐 이름 정할 때 쓰는 함수. /mq0000부터 시작해서 프로세스 수 만큼 1씩 올릴 때 쓰는 함수이다. eg) /mq0189 → /mq0190	void
getLen	int fd	파일 첫 줄에 있는 숫자의 길이를 리턴한다	int
getSize	int fd, int len	파일 첫 줄에 있는 자료의 개수를 읽을 때 쓰는 함수.	int
printFD	int* intArr, int interval	도수분포표 출력용 함수	void
FDCheck	int* intArr, int interval	도수분포표에 있는 숫자의 개수 체크용 함수	void
Untitled			

Performance Evaluation

data 개수: 100000개

interval: 1000


pridom@DripBox:~/Desktop/SSLab/Assignment$ time ./ku_pfred 1 1000 dataset
10090
9981
10083
9941
9881
10052
10029
10018
9896
10029
real 0m1.576s
user 0m0.039s
sys 0m0.784s
pridom@DripBox:~/Desktop/SSLab/Assignment$ time ./ku_pfred 2 1000 dataset
10090
9981
10083
9941
9881
10052
10029
10018
9896
10029
real 0m1.740s
user 0m0.000s
sys 0m0.845s


pridom@DripBox:~/Desktop/SSLab/Assignment$ time ./ku_pfred 3 1000 dataset
10090
9981
10083
9941
9881
10052
10029
10018
9896
10029
real 0m1.524s
user 0m0.043s
sys 0m0.783s
pridom@DripBox:~/Desktop/SSLab/Assignment$ time ./ku_pfred 4 1000 dataset
10090
9981
10083
9941
9881
10052
10029
10018
9896
10029
real 0m1.289s
user 0m0.000s
sys 0m0.641s

```
pridom@DripBox:~/Desktop/SSLab/Assignment$ time ./ku_pfred 5 1000 dataset
10090
9981
10083
9941
9881
10052
10029
10018
9896
10029

real    0m1.274s
user    0m0.010s
sys     0m0.596s
pridom@DripBox:~/Desktop/SSLab/Assignment$ time ./ku_pfred 10 1000 dataset
10090
9981
10083
9941
9881
10052
10029
10018
9896
10029

real    0m1.474s
user    0m0.067s
sys     0m0.657s
```

프로세스 개수가 많아지면 걸리는 시간이 줄지만, 너무 많아지면 오히려 실행시간이 늘어납니다.

Multi-threaded program

Mutex를 사용해서 쓰레드들이 공유하는 값 (전역변수)이 원하지 않게 수정되는 것을 막는다.

쓰레드도 역시 똑같은 방식으로 구성됩니다. 단, 이번엔 메세지 큐를 통해서 공유를 하지 않고

전역 변수와 mutex를 통해서 자원들을 공유하고 그 값들을 수정합니다.

(전역 변수들은 다 대문자로 표기했으며, 나머지를 뜻하는 REMAINDER는 volatile로 선언 되었습니다)

Documentation

Function Name	Arguments	Description	Return Value
readT	void* data	파일을 열고 읽는다. 그리고 숫자들을 intArr에 세어준다.	void*
readTX	void* data	역시 파일을 열고 읽은 후 읽은 숫자들을 세어주는데 나머지 만큼 더 읽어준다.	void*
FDCheck	void	도수분포표에 숫자가 잘 세어졌는지 확인해준다.	void
printFD	void	도수분포표 출력용 함수	void
getLen	void	파일 첫 줄에 있는 숫자의 길이를 리턴한다	int
getSize	void	파일 첫 줄에 있는 자료의 개수를 읽을 때 쓰는 함수.	int

Performance Evaluation

역시 자료 개수 100000, 구간 크기 1000

```
pridom@DripBox:~/Desktop/SSLab/Assignment$ time ./ku_tfred 1 1000 dataset
10090
9981
10083
9941
9881
10052
10029
10018
9896
10029

real    0m0.065s
user    0m0.032s
sys     0m0.032s
pridom@DripBox:~/Desktop/SSLab/Assignment$ time ./ku_tfred 2 1000 dataset
10090
9981
10083
9941
9881
10052
10029
10018
9896
10029

real    0m0.044s
user    0m0.014s
sys     0m0.065s
```

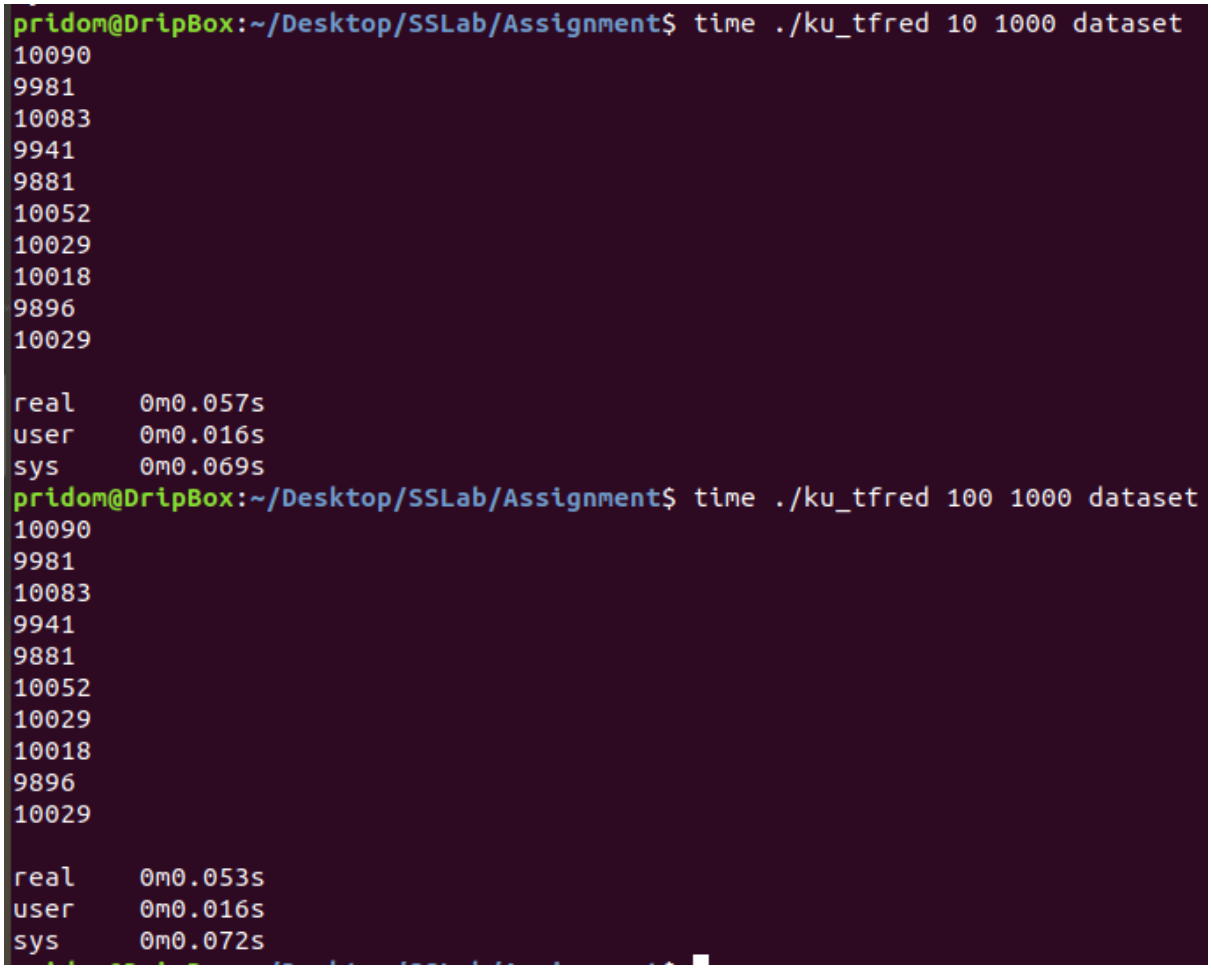
```
pridom@DripBox:~/Desktop/SSLab/Assignment$ time ./ku_tfred 3 1000 dataset
10090
9981
10083
9941
9881
10052
10029
10018
9896
10029

real    0m0.044s
user    0m0.004s
sys     0m0.075s
```

```
pridom@DripBox:~/Desktop/SSLab/Assignment$ time ./ku_tfred 4 1000 dataset
10090
9981
10083
9941
9881
10052
10029
10018
9896
10029

real    0m0.049s
user    0m0.009s
sys     0m0.072s
pridom@DripBox:~/Desktop/SSLab/Assignment$ time ./ku_tfred 5 1000 dataset
10090
9981
10083
9941
9881
10052
10029
10018
9896
10029

real    0m0.048s
user    0m0.017s
sys     0m0.064s
```



역시 쓰레드의 수를 늘려갈 수록 대체적으로 실행 속도가 줄어들긴 하지만,

쓰레드를 너무 많이 생성하면 오히려 실행 시간이 늘어납니다. context switch를 과도하게 하다보니

오버헤드가 느는 것 같습니다.

코드에도 주석을 달아놨지만 메인이 조금 지저분합니다. 죄송합니다.