OS Assignment #2

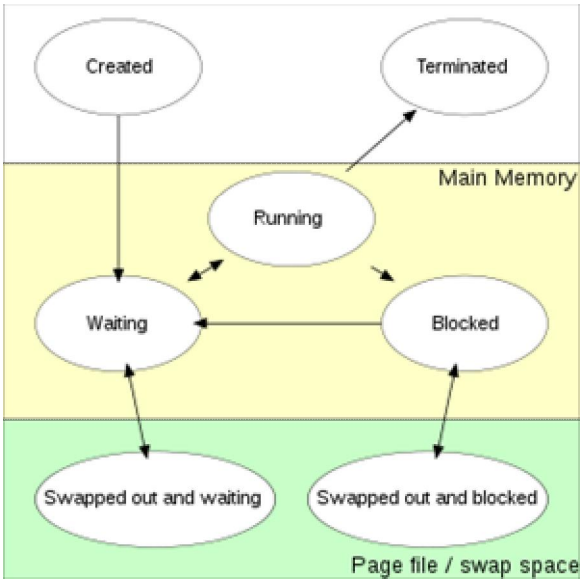
2015147531 서기원

[사전 보고서]

■ 프로세스와 스레드

√ 프로세스?

일반적으로 프로그램은 하드 디스크, SSD 등과 같은 보조 기억 장치에 저장되어 실행되기를 기다리는 코드와 정적인 데이터의 묶음이다. 이 프로그램의 명령어와 정적 데이터가 메모리에 적재되면 생명이 있는 프로세스가 되어 실행된다. 따라서, 프로세스는 프로그램을 구동하여 프로그램 자체와 프로그램의 상태가 메모리상에서 실행되는 작업 단위를 지칭한다.



위의 그림은 프로세스의 여러 가지 상태를 표현한 것으로 크게 생성(create), 실행(running), 준비(ready), 대기(waiting/block), 종료(terminated)로 나눌 수 있다. 생성 단계에서 프로세스가 생성되어 준비 단계에서 CPU가 할당하기를 기다리다가 실행단계에서 CPU가 해당 프로세스를 실행한다. 프로세스가 입출력, 시그널 수신과 같은 사건을 실행하게 되면 대기 상태로 넘어가 해당 사건이 끝나기를 기다린다. 그리고 프로세스는 종료 단계에서 실행이 종료된다.

√ 스레드?

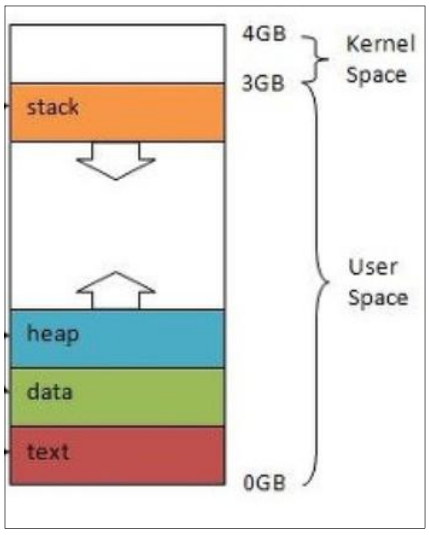
스레드는 한 프로세스 내에서 실행되는 흐름의 단위로 각각이 프로세스가 할당받은 자원을 이용하는 실행의 단위이다. 같은 프로세스 내에서의 스레드는 각각 Stack 영역은 따로 할당

받고 Code, Data, Heap 영역은 공유한다. 따라서, 한 스레드가 프로세스의 자원을 변경하면 다른 이웃 스레드도 그 변경 결과를 즉시 볼 수 있다.

√ 프로세스와 스레드의 차이점

커널의 관점에서 보면 프로세스의 목적은 시스템의 자원을 할당받는 것이라고 할 수 있다. 이러한 프로세스는 부모 프로세스와 자식 프로세스로 나눌 수 있는데 서로 별개의 데이터 복사본을 가지기 때문에 서로의 프로세스 내에서 데이터를 바꿔도 서로 알지 못한다. 이러한 문제를 해결하기 위해 프로세스보다 더 작지만 자원을 공유할 수 있는 스레드를 사용하게 되었다. 스레드는 자원 생성과 관리의 중복성을 최소화하여 실행 능력을 향상할 수 있다.

√ 프로세스의 리눅스에서의 구조 및 구현

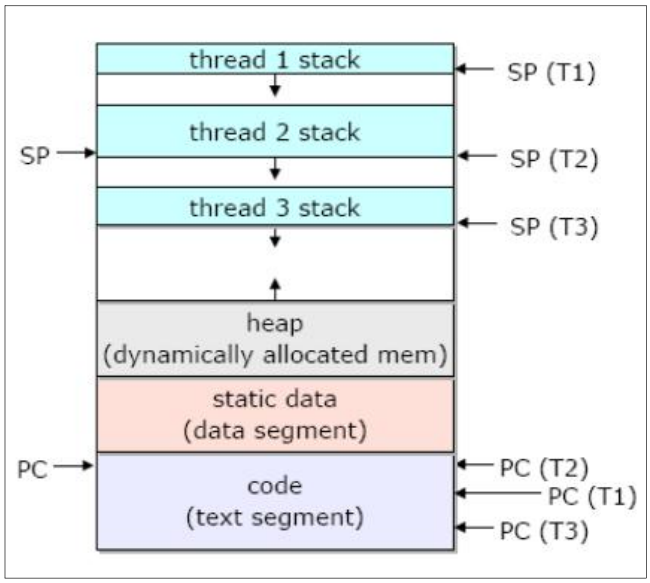


리눅스는 32bit 컴퓨터의 경우 4GB크기의 가상공간을 각 프로세스에게 할당을 하는데 그 중 0~3GB의 공간을 사용자 공간으로 사용하고, 나머지 3~4GB를 커널 공간으로 사용한다. 프로세스는 커널이 가진 여러 가지 자원의 할당 및 사용을 위해 커널 함수를 호출해야 하는데 이러한 커널 함수들을 일반적으로 시스템 호출(System call)이라고 한다. 시스템 호출의 예는 아래와 같다.

fork 함수를 통하여 독립적인 pid를 가지고 있지만 다른 부분은 부모 프로세스와 모두 같은 복사본을 가진 자식 프로세스를 생성한다. 그 뒤 exec 계열의 함수를 사용하여 실행시키고자 하는 프로그램의 이미지를 해당 프로세스에 덮어씌울 수 있다.

따라서, fork와 exec의 조합으로 프로세스 구현이 가능하다.

√ 스레드의 리눅스에서의 구조 및 구현



같은 프로세스 내의 스레드는 code, data, heap 영역의 정보를 공유하기 때문에 위 그림과 같은 메모리 구조를 가지게 됩니다. 따라서, pid는 부모 프로세스와 같지만 고유의 spid를 갖습니다.

구현 방식은 <pthread.h> 헤더파일을 추가하여 스레드를 구현할 수 있는 함수를 사용한다. pthread_create로 스레드를 생성하여 인자값으로 실행시킬 함수를 스레드에 알려준다. pthread_join을 통해서 생성시킨 스레드의 종료를 기다린다.

■ 멀티 프로세스와 멀티 스레딩 개념 및 구현 방법

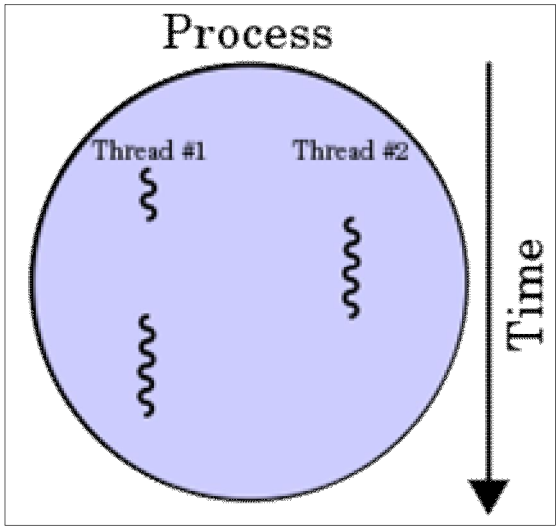
√ 멀티 프로세스?

프로세스 내에서 새로운 프로세스를 만들어내어 여러 가지 일을 처리하는 방법이다.

fork 함수를 통해 부모 프로세스에서 똑같은 이미지를 가진 자식 프로세스가 다른 pid를 가지고 복제된다. 이렇게 생성된 자식 프로세스에서 exec 계열의 함수들을 통해 부모 프로세스와 다른 이미지를 덮어씌워 실행시키고자 하는 프로그램들을 실행시키며 exit 함수를 통해 종료시킬 수 있고 부모 프로세스에서는 wait 함수를 이용하여 자식 프로세스의 죽음을 기다릴 수 있다. 모든 프로세스는 자신만의 메모리 영역을 가지고 있기에 프로세스간의 통신을 위해서는 IPC를 통해서 이루어져야 한다.

√ 멀티 스레딩?

프로세스 내에서 작업 단위를 나누어 처리하는 것이다. 예를 들어 1부터 10까지 더하는 프로세스가 있다면 이를 잘게 쪼개어 1~3, 4~6, 7~10까지 더하는 스레드로 분기하여 계산할 수 있을 것이다. 간단한 예시에 대한 그림은 다음과 같다.



멀티 스레딩의 장점으로는 응용 프로그램이 같은 주소 공간 내에 여러 개의 다른 활동성 스레드를 가질 수 있다는 이점이 있다. 또한, 프로세스 생성에 메모리와 자원을 할당하는 것은 비용이 많이 들지만 스레드는 자시이 속한 프로세스의 자원들을 공유하기 때문에, 스레드를 생성하고 문맥교환을 하는 편이 보다 경제적이다. 하지만 이러한 장점은 스레드 간의 데이터 변경에 대한 동기화 문제가 발생할 수 있다.
구현 방법은 위에서 설명했던 것과 마찬가지로 <pthread.h> 헤더 파일에 포함된 pthread_create 함수를 통해 스레드를 생성하여 스레드내에서 함수의 실행이 종료될 때까지 pthread_join 함수를 통해 기다리면서 프로그램 실행이 진행된다.

■ exec 계열의 시스템 콜

√ exec 계열의 시스템 콜 종류

exec 계열의 함수는 <unistd.h> 헤더파일 내에 존재하며 다음 6가지의 함수들이 있다.

```
int execl(const char *path, const char *arg(), …, (char *)0);
int execlp(const char *file, const char *arg(), …, (char *)0);
int execle(const char *path, const char *arg(), …, (char *)0, char *const envp[]);

int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execve(const char *path, char *const argv[], char *const envp[]);
```

위 6개의 함수들은 모두 현재 프로세스를 새로운 프로그램으로 대체해버리는 역할을 하며 프로그램 인수들을 받는 방식에 따라 두 종류로 나뉜다. 위의 3개는 NULL 포인터로 끝나는 가변적인 개수의 매개변수들을 통해서 프로그램 인수를 받고, 아래의 3개는 문자열 배열 매개변수를 통해서 프로그램 인수들을 받는다.

√ 리눅스에서의 구현

- execl은 디렉토리를 포함한 파일의 전체 이름을 받아오고 인수 값도 char의 포인터 배열로 받는 것이 아니라 각각 개개로 나뉘어진 parameter을 통해 전해받는다.
- execlp는 파일 이름과 parameter들의 리스트.
- execle는 전체 경로를 포함한 이름과 parameter들의 리스트.(execl과 다르게 마지막에 NULL 포인터 인자를 넘겨 줘야한다.)
- execv는 전체 경로를 포함한 이름과 char의 포인터 배열.
- execvp는 파일 이름과 char의 포인터 배열.
- execve는 전체 경로를 포함한 이름과 char의 포인터 배열.(execv와는 다르게 마지막에 NULL 포인터 인자를 넘겨 줘야한다.

√ 리눅스에서 동작하는 방법

코드 상의 구현에 대한 예시는 execvp 함수로 하였다. 이는 다음과 같다.

```
#include <unistd.h>
int main(void) {
        …
        char* argv[4] = {“./a.out”, “40”, “20”};
        argv[3] = NULL;
        execvp(argv[0], argv);
        …
}
```

이에 대한 결과로는 터미널에서 “~(해당 디렉토리)$ ./a.out 40 20 <enter>”를 입력한 것과 같은 효과를 기대할 수 있다.

■ 참고 문헌

프로세스 관련 참고:

https://ko.wikipedia.org/wiki/%ED%94%84%EB%A1%9C%EC%84%B8%EC%8A%A4

https://bowbowbow.tistory.com/16

http://www.spacek.xyz/mle/?p=511

OS_Lecture note “L3-process”

스레드 관련 참고:

https://ko.wikipedia.org/wiki/%EC%8A%A4%EB%A0%88%EB%93%9C_(%EC%BB%B4%ED%93%A8%ED%8C%85)

https://gmlwjd9405.github.io/2018/09/14/process-vs-thread.html
[그림참고] : OS_Lecture note "L4-threads"

멀티프로세스 관련 참고:
https://you9010.tistory.com/136

멀티스레딩 관련 참고:
https://ko.wikipedia.org/wiki/%EB%A9%80%ED%8B%B0%EC%8A%A4%EB%A0%88%EB%94%A9

exec 계열 함수 관련 참고:
https://sosal.kr/37

[프로그래밍 수행 결과 보고서]

■ 개발 환경

```
giwon@giwon-VirtualBox:~$ uname -a
Linux giwon-VirtualBox 5.4.28-2015147531 #1 SMP Sun Mar 29 02:17:02 KST 2020 x86_64 x86_64 x86_64 GNU/Linux
giwon@giwon-VirtualBox:~$ g++ --version
g++ (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0
Copyright (C) 2017 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

giwon@giwon-VirtualBox:~$ grep -c processor /proc/cpuinfo
2
```

```
giwon@giwon-VirtualBox:~$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 60
model name      : Intel(R) Core(TM) i5-4210M CPU @ 2.60GHz
stepping        : 3
cpu MHz         : 2594.012
cache size      : 3072 KB
physical id     : 0
siblings        : 2
core id         : 0
cpu cores       : 2
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx rdtscp lm constant_tsc rep_good nopl xtopology nonstop_tsc cpuid tsc_know
_freq pni pclmulqdq ssse3 cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt aes xsave avx rdrand hypervisor lahf_lm abm invpcid_single pti fsgsbase avx2 invpcid
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds swapgs itlb_multihit
bogomips        : 5188.02
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:

processor       : 1
vendor_id       : GenuineIntel
cpu family      : 6
model           : 60
model name      : Intel(R) Core(TM) i5-4210M CPU @ 2.60GHz
stepping        : 3
cpu MHz         : 2594.012
cache size      : 3072 KB
physical id     : 0
```

```
processor       : 1
vendor_id       : GenuineIntel
cpu family      : 6
model           : 60
model name      : Intel(R) Core(TM) i5-4210M CPU @ 2.60GHz
stepping        : 3
cpu MHz         : 2594.012
cache size      : 3072 KB
physical id     : 0
siblings        : 2
core id         : 1
cpu cores       : 2
apicid          : 1
initial apicid  : 1
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx rdtscp lm constant_tsc rep_good nopl xtopology nonstop_tsc cpuid tsc_know
_freq pni pclmulqdq ssse3 cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt aes xsave avx rdrand hypervisor lahf_lm abm invpcid_single pti fsgsbase avx2 invpcid
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds swapgs itlb_multihit
bogomips        : 5188.02
clflush size    : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:

giwon@giwon-VirtualBox:~$
```

g++ 컴파일러의 버전은 7.5.0 버전이었으며 cpu는 2개의 코어로 설정하여 프로그래밍을 진행하였다. cpu 정보는 2장에 걸쳐 그림으로 나타내었다.

```
giwon@giwon-VirtualBox:~$ free -h
              total        used        free      shared  buff/cache   available
Mem:           3.8G        879M        2.0G         38M        1.0G        2.7G
스왑:          2.0G          0B        2.0G
```

메모리 정보는 위의 그림과 같다.

■ 작성한 프로그램의 동작 과정과 구현 방법

✓ Program 1

※ 변수

filter_num : 필터의 개수

row : 필터의 행 / col : 필터의 열

inputX : 입력 데이터의 행 / inputY : 입력 데이터의 열

int**** filter : 필터의 개수, 채널(3), 행, 열의 4가지 정보를 담은 4차원 배열

int*** inArray : 입력 데이터의 채널(3), 행, 열의 3가지 정보를 담은 3차원 배열

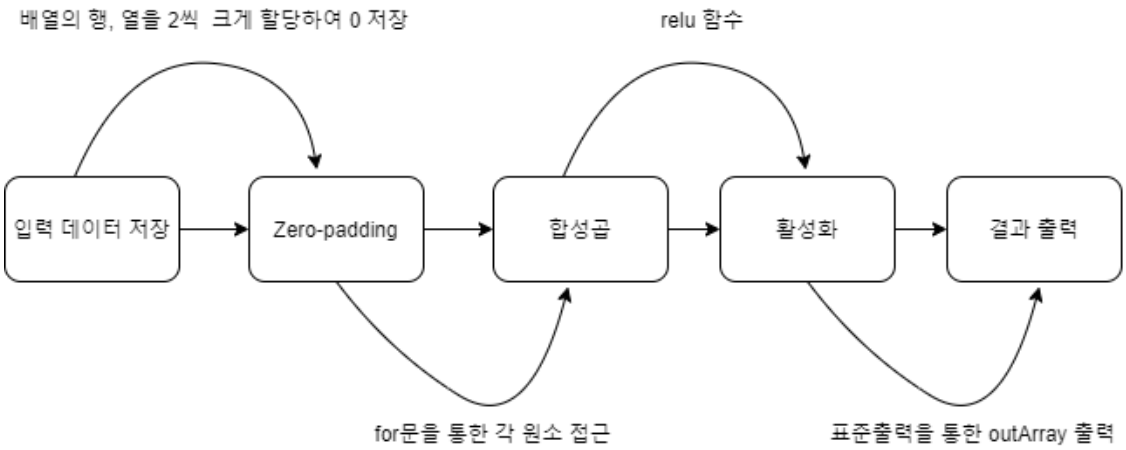
int*** outArray : 결과 데이터의 필터 개수, 행, 열의 3가지 정보를 담은 3차원 배열

※ 함수

int max(int a, int b) : 활성화 단계에서 0보다 큰 수를 리턴하기 위해 둘 중 큰 수를 리턴

int relu(int x) : 활성화 단계의 함수로 0과 비교해서 큰 수를 리턴

※ 동작 과정 과 구현 방법



- 필터와 입력 데이터를 각각 행렬로 표준 입력을 통해 주어진 값을 저장한다.
- 입력 데이터의 배열의 경우 zero-padding을 위해 주어진 행, 열보다 2를 큰 값으로 저장해 주어진 입력 데이터 주위에 0을 저장한다.
- for문을 통해 각각의 원소에 접근하여 합성곱을 한 뒤 outArray에 결과를 저장한다.
- outArray의 모든 값에 대하여 relu함수를 통해 활성화를 진행한다.

√ Program 2

※ 변수

pro : 주어진 프로세스의 개수

div : 프로세스에 알맞은 필터를 부여하기 위해 필터 개수를 프로세스 개수로 나눈 몫

remain : div와 비슷한 이유로 필터 개수를 프로세스 개수로 나눈 나머지

stk : 임시 입력 파일을 생성할 때 파일 번호가 remain보다 작은 지를 판단하는 변수

fstk : 임시 입력 파일을 생성할 때 필터 번호를 파악하는 변수

outstack : 임시 결과 파일을 읽어드릴 때 필터 번호를 파악하는 변수

nw : 임시 결과 파일을 임시 입력 파일과 구분하기 위해 추가할 이름 string

pid : 자식 프로세스의 생성 여부를 확인하는 pid_t* 변수

statloc : 자식 프로세스의 종료를 기다리기 위해 wait의 상태를 나타내는 변수

filter_num, row, col, inputX, inputY, filter, inArray, outArray → program1과 동일

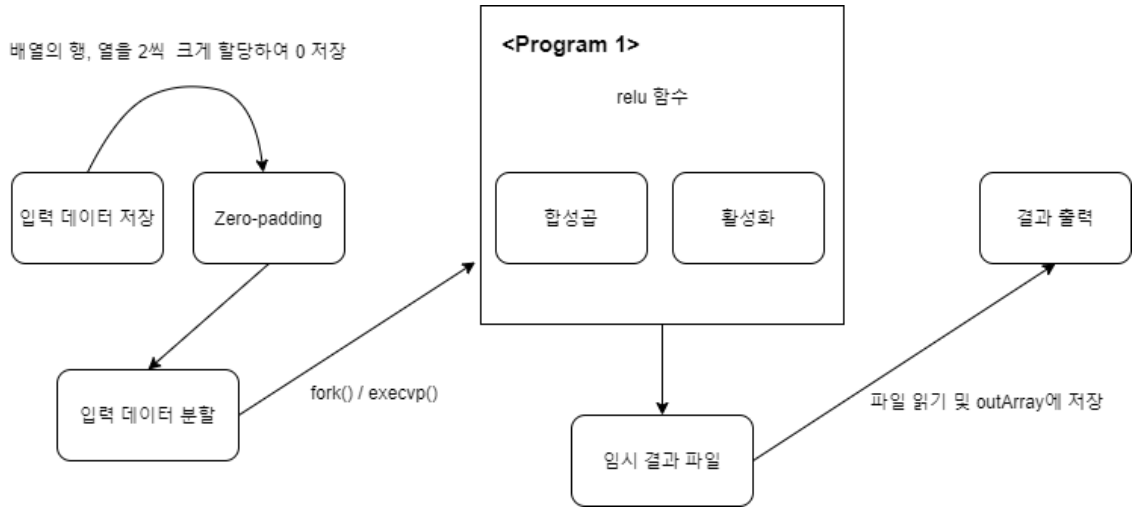
※ 함수

fork : 자식 프로세스 생성

execvp : 자식 프로세스에 program1 이미지를 전달 및 덮어씌움

자식 프로세스 내에서 program1을 호출하여 계산하기 때문에 계산을 위한 함수가 없다.

※ 동작 과정 과 구현 방법



- zero-padding 과정까지는 program1과 동일하다.
- 주어진 프로세스 개수만큼 필터를 분할하여 입력 데이터와 같이 임시 파일에 저장한다.
- fork와 execvp을 통해 자식 프로세스를 생성하여 program1과 같은 이미지로 만들어 각각의 임시 파일을 입력 데이터로 하여 실행하고 결과값을 임시 결과 파일에 저장한다.
- 임시 결과 파일을 불러들여 outArray에 순서대로 저장하여 출력한다.

√ Program 3

※ 변수

전역 변수를 공유하는 멀티 스레드의 장점을 활용하고자 스레드 내에서 사용하는 변수들을 전역 변수로 설정하였다.

t_n : 주어진 스레드의 개수

total_time : 각각의 스레드내에서 계산 시간을 저장하기 위한 배열

divided : program2에서의 div와 유사하다

remain : program2에서의 remain과 똑같다

from : convolution 함수에서 계산을 할 필터의 시작 번호

to : convolution 함수에서 계산을 할 필터의 마지막 번호 + 1

filter_num, row, col, inputX, inputY, filter, inArray, outArray → program1과 동일

※ 함수

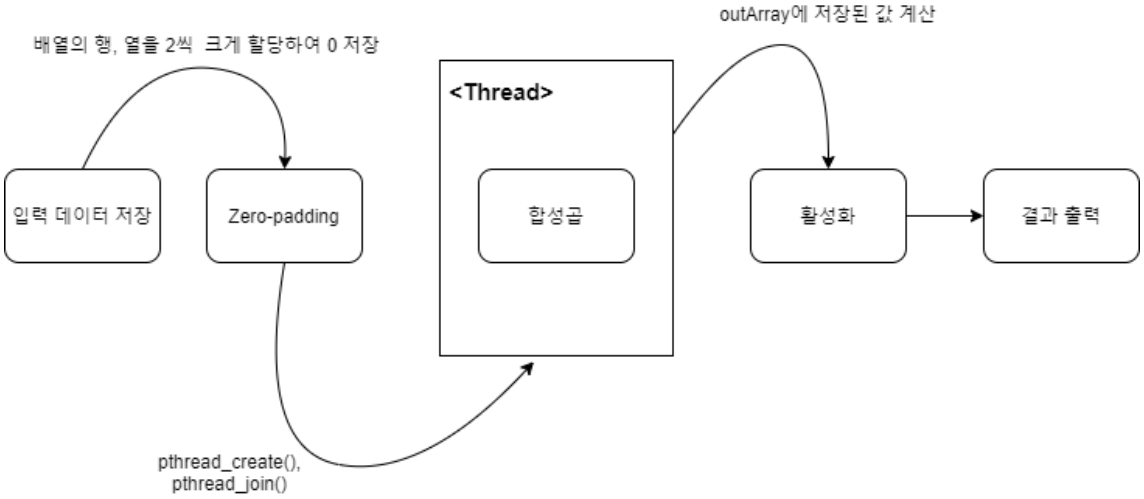
thread_function : 각각의 스레드에서 실행시킬 함수

convolution : thread_function에서 합성곱할 필터의 번호를 받아 실제 계산을 하는 함수

do_relu : outArray에 모든 원소에 대하여 relu 함수를 적용하는 함수

pthread_create, pthread_join을 통해 멀티 스레딩을 구현하였다

※ 동작 과정 과 구현 방법



- zero-padding까지의 과정은 program1과 동일하다.
- 주어진 스레드 개수만큼 스레드를 생성하여 스레드 번호를 인자로 넘겨주어 스레드내에서 스레드별로 계산해야할 필터의 정보를 계산한다.
- 스레드별로 주어진 필터에 대하여 합성곱 연산을 진행하고 결과를 outArray에 저장한다.
- do_relu 함수를 통해 outArray의 모든 원소에 대하여 활성화 함수를 적용한다.

■ 작성한 Makefile

```
CC = g++
OBJ = program1 program2 program3
CXXFLAGS = --std=c++11

main:: $(OBJ)
program1:
	$(CC) $(CXXFLAGS) -o program1 p1.cpp

program2:
	$(CC) $(CXXFLAGS) -o program2 p2.cpp

program3:
	$(CC) $(CXXFLAGS) -o program3 p3.cpp -lpthread

clean:
	rm -f $(OBJ)
~
~
```

작성한 makefile 전문

시간 측정을 위해 사용한 chrono 헤더 파일 내 함수는 c+11 버전 함수이기 때문에 컴파일 과제에서 c+11 옵션을 추가하였다.

make 명령어를 사용 시 program1, program2, program3 실행 파일이 생성되고 make clean 명령어를 다시 사용하면 생성된 실행 파일이 모두 삭제된다.

```
giwon@giwon-VirtualBox:~/Desktop/2015147531$ make
g++ --std=c++11 -o program1 p1.cpp
g++ --std=c++11 -o program2 p2.cpp
g++ --std=c++11 -o program3 p3.cpp -lpthread
```

makefile 실행시 결과 화면

■ 애로 사항 및 해결 방법

<메모리 참조 오류>

- 입력 데이터를 배열로 동적 할당하여 저장하였기 때문에 할당되지 않은 메모리에 접근할 시 segmentation fault 에러가 발생하였다.

  ★ 각각의 배열들의 각 차원에서 의미하는 내용이 달라 혼동하여 쓰는 경우가 많았다.

  ★ 결과값의 행과 열이 계산하고자 하는 입력 데이터와 달라 할당을 잘못 했었다.

  해결) 결과값의 행 = (inputX + 2) − row + 1,

  열 = (inputY + 2) − col + 1

<구동 시간 계산>

- clock을 활용하여 프로그램의 실행 시간을 계산하였을 때, millisecond단위의 시간이 제대로 계산되지 않았다.

  해결) c+11에 새롭게 추가된 chrono를 사용하여 시간을 보다 정확하게 계산하였다.

<Program 2>

- wait 함수를 사용시 하나의 자식 프로세스만 종료되어도 부모 프로세스가 다시 작동하는 경우도 있었다.

  해결) waitpid 함수를 활용하여 배열로 저장한 각각의 pid를 대응시켜 모든 pid의 자식 프로세스가 종료되어야 부모 프로세스가 main 함수를 계속 진행하게끔 하였다.

- 주어진 필터의 순서대로 결과값이 출력되어야 한다는 조건

  ★ 처음 프로그래밍을 할 때, 프로세스에 필터 개수를 적절히 배분한 후 남은 필터를 프로세스에 순차적으로 배분하여 주어진 입력 데이터의 순서대로 결과값이 출력되지 않았다.

  해결) outstack변수를 선언하여 필터의 순서를 정확하게 파악하여 배분하였다.

<Program 3>

- pthread_create로 생성된 스레드에서 합성곱 연산을 하기 위해 필요한 필터의 정보를 어떤 방식으로 참조를 해야할지 고민하였다.

  해결) 필터, input 배열, output 배열을 전역 변수로 선언하여 각각의 스레드가 데이터에 대해서 접근할 수 있게끔 하였고, 접근하는 데이터가 겹치지 않게끔 스레드의 번호를 인자로 넘겨주어 접근하는 배열의 시작점과 끝점을 계산하였다.

■ 결과 화면과 결과에 대한 토의

program1 : 일반적인 프로그램 실행

program2 : 멀티 프로세스를 활용한 프로그램 실행

program3 : 멀티 스레딩을 활용한 프로그램 실행

위의 목적에 알맞은 결과를 분석하기 위해 input data의 경우 2개를 준비

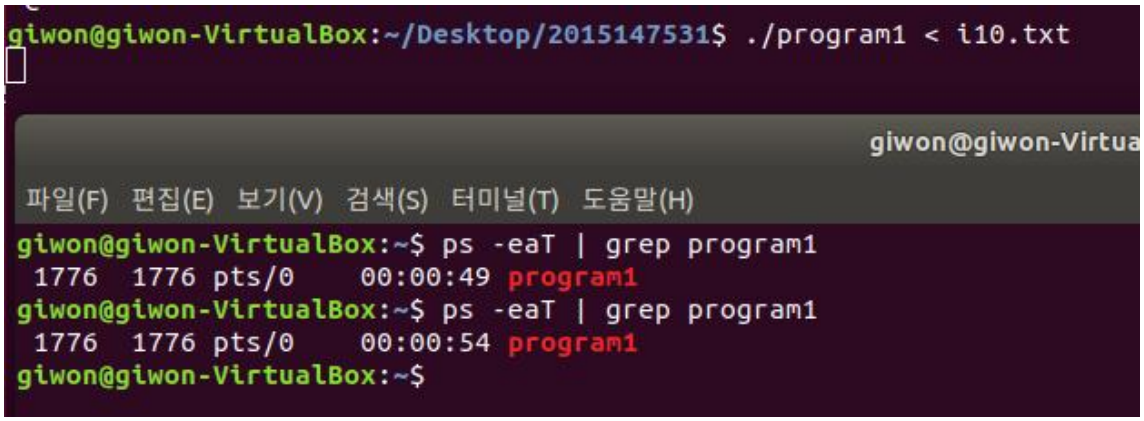
√ 개발 환경에서 cpu core의 개수가 2개인 관계로 프로세스와 스레드의 개수를 1, 2, 4 일 경우 모두 실행해서 비교

√ 정확한 비교를 위하여 실행 시간의 경우 convolution layer와 relu함수를 적용하는 부분에 대해서 측정

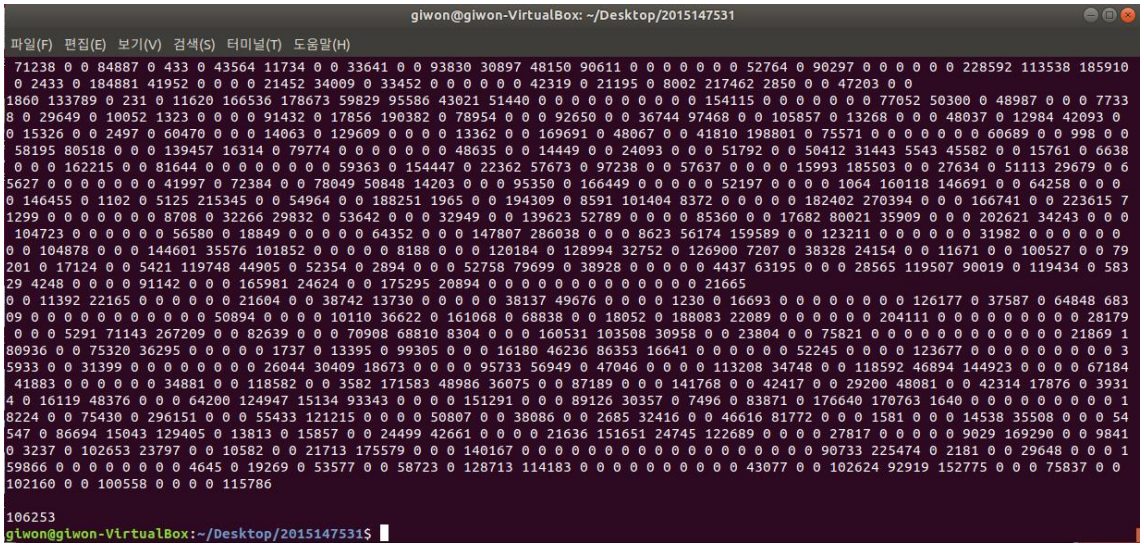
√ 부모 프로세스의 경우 자식 프로세스를 생성하고 모든 자식 프로세스가 종료할 때까지의 시간으로 측정

1. 필터 : 10개 50행 50열 / 데이터 : 500행 500열

[program 1]



프로세스 동작 확인 결과

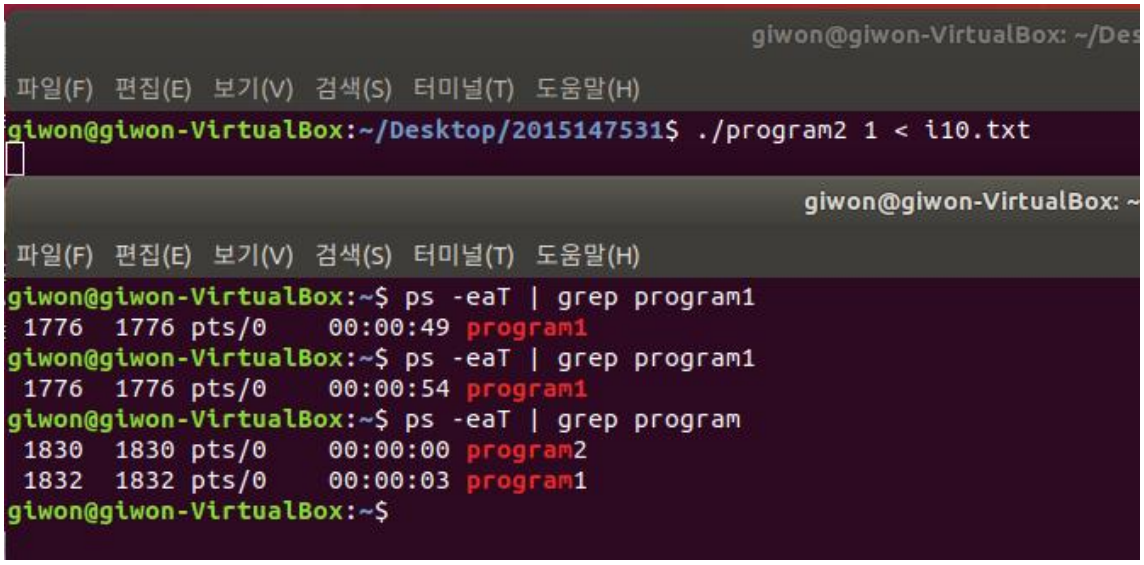


★ 시간 결과

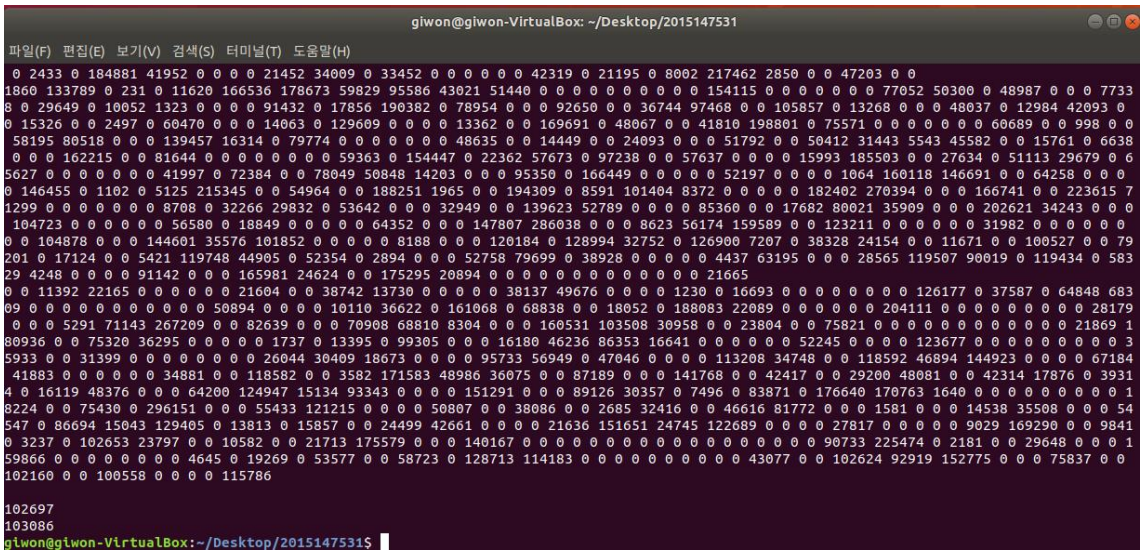
프로세스 내 실행 시간 : 106253 milliseconds

[program 2]

√ 프로세스의 개수를 1개로 지정해줄 경우



프로세스 동작 확인 결과



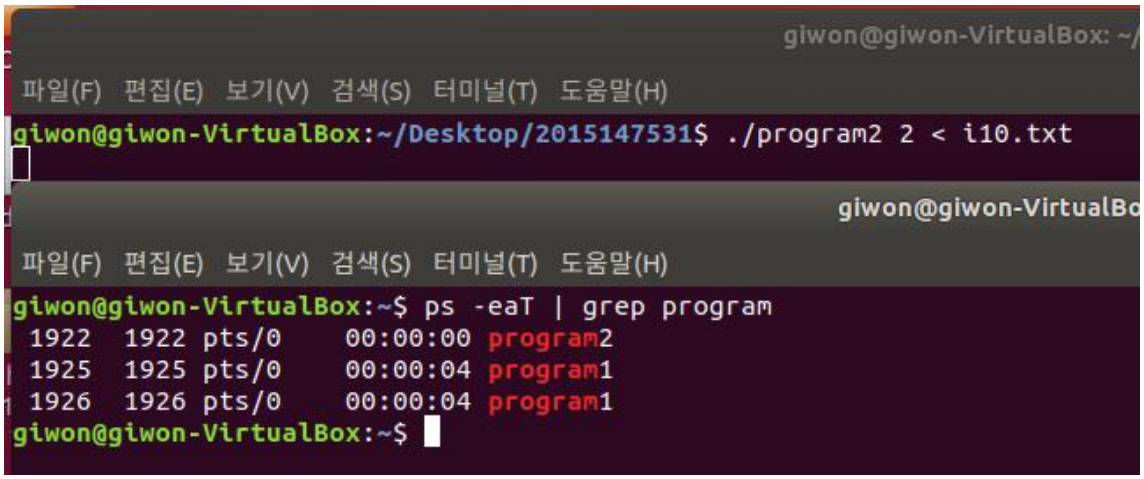
★ 시간 결과

부모 프로세스 내 실행 시간 : 103086 milliseconds

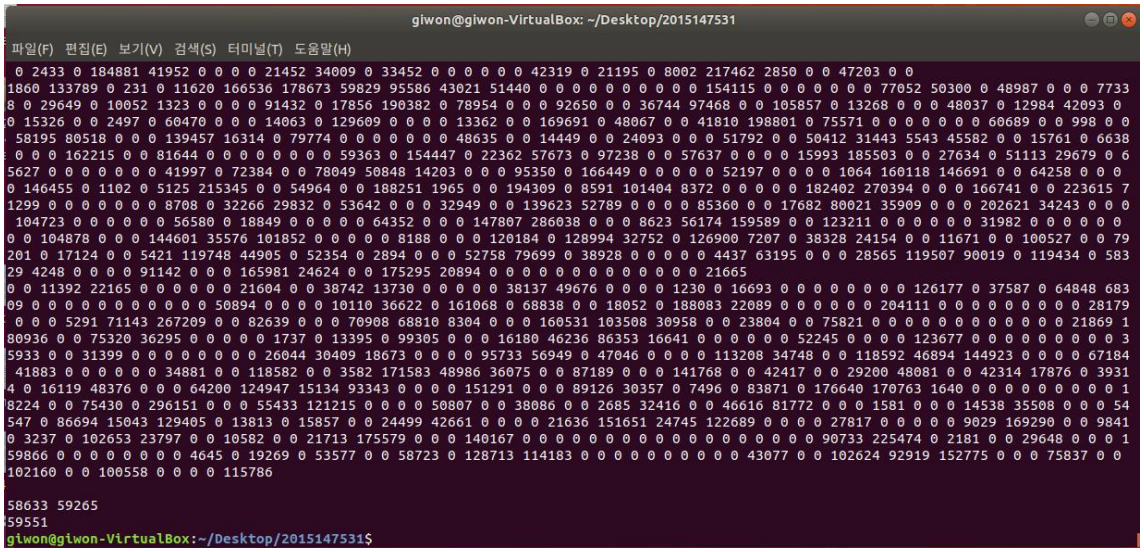
자식 프로세스 내 실행 시간 : 102697 milliseconds

- 실행 도중 프로세스의 동작을 확인해보면 program2 실행 이후 program1이 1개의 프로세스에 할당되어 실행되고 있음을 알 수 있다.

√ 프로세스의 개수를 2개로 지정해줄 경우



프로세스 동작 확인 결과



★ 시간 결과

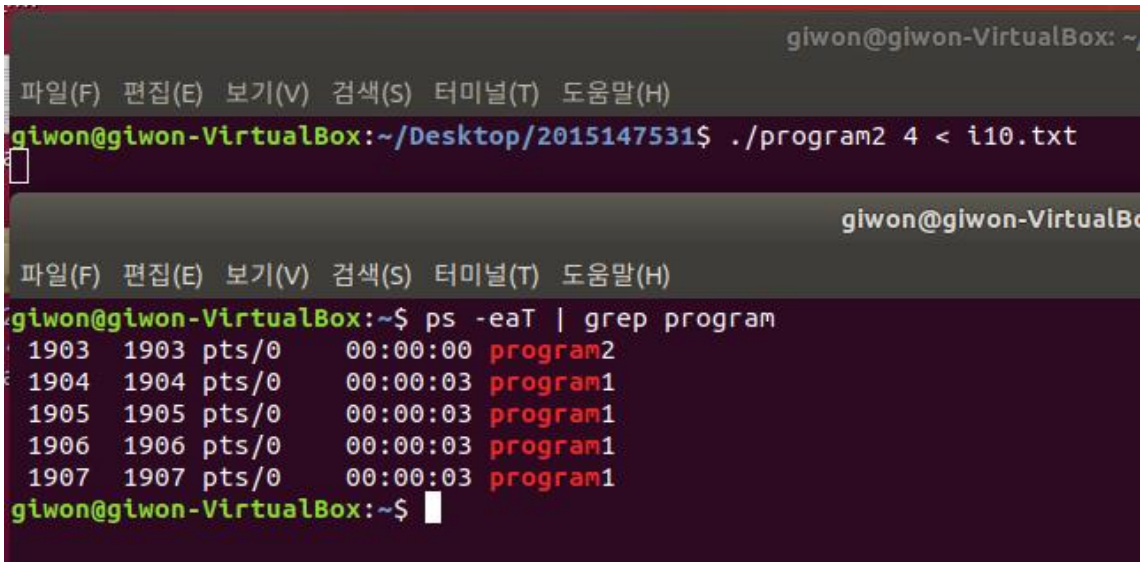
부모 프로세스 내 실행 시간 : 59551 milliseconds

자식 프로세스 1 내 실행 시간 : 58633 milliseconds

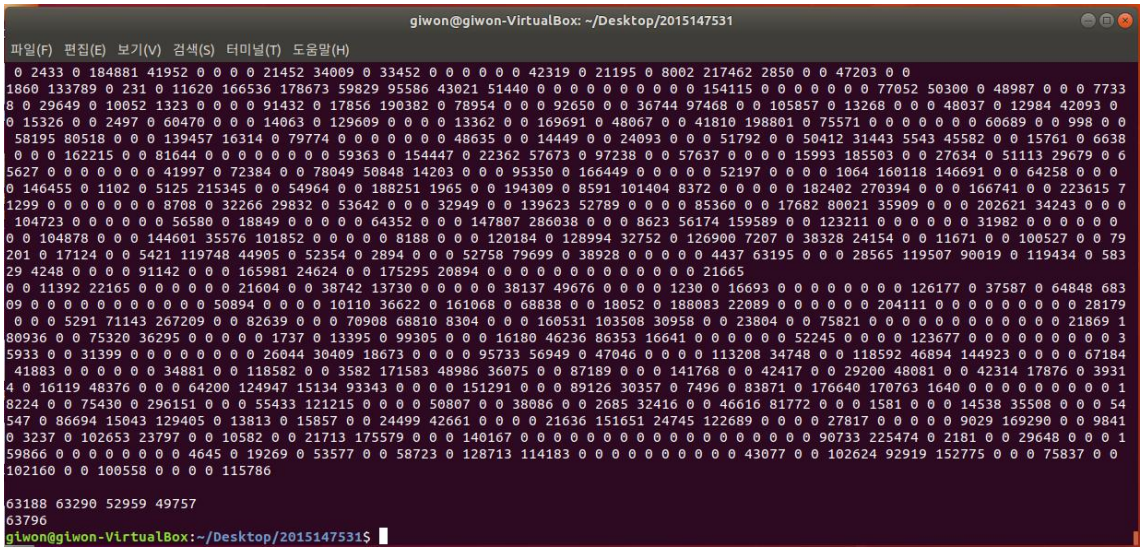
자식 프로세스 2 내 실행 시간 : 59265 milliseconds

- 실행 도중 프로세스의 동작을 확인해보면 program2 실행 이후 program1이 2개의 프로세스에 할당되어 실행되고 있음을 알 수 있다.

√ 프로세스의 개수를 4개로 지정해줄 경우



프로세스 동작 확인 결과



★ 시간 결과

부모 프로세스 내 실행 시간 : 63796 milliseconds

자식 프로세스 1 내 실행 시간 : 63188 milliseconds

자식 프로세스 2 내 실행 시간 : 63290 milliseconds

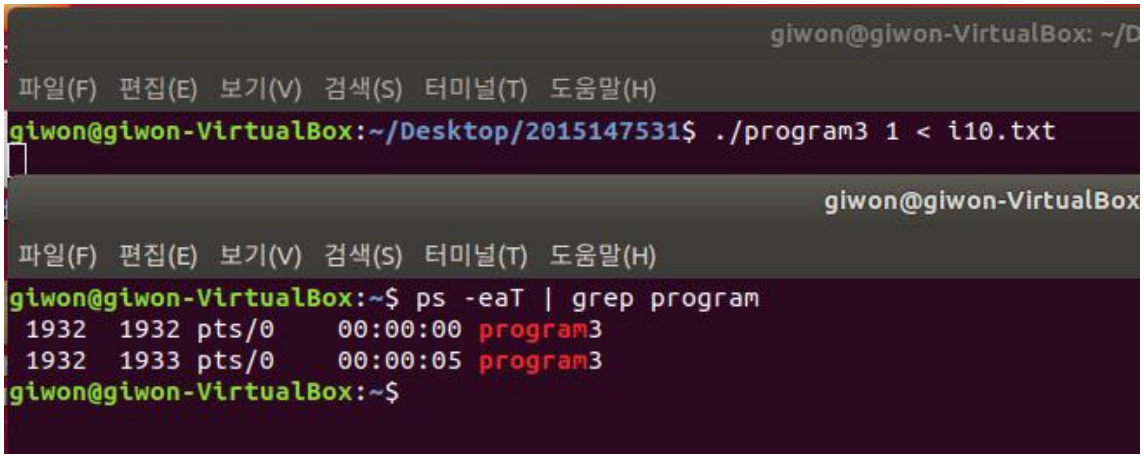
자식 프로세스 1 내 실행 시간 : 52959 milliseconds

자식 프로세스 2 내 실행 시간 : 49757 milliseconds

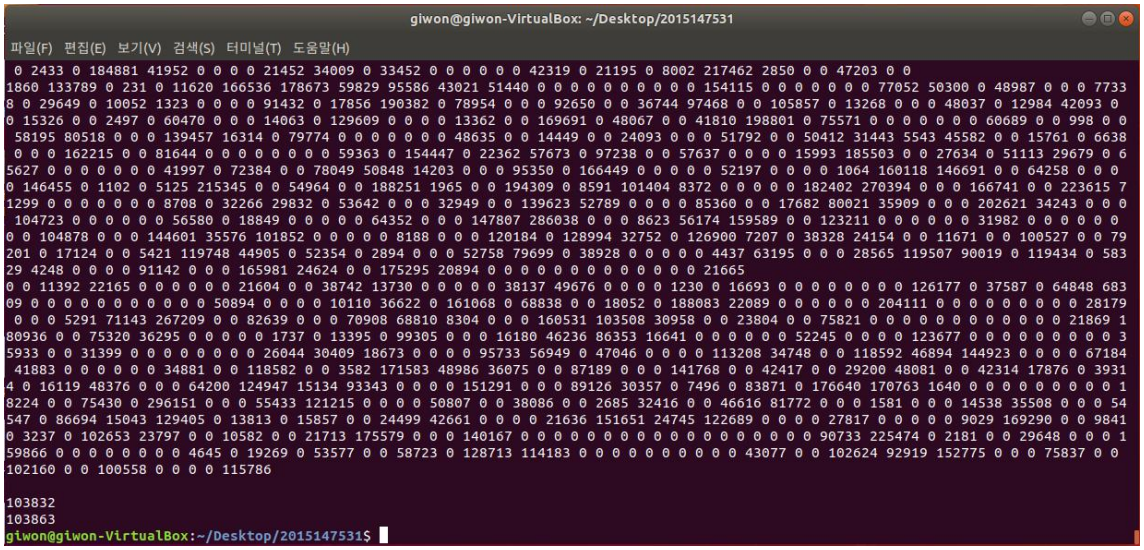
- 실행 도중 프로세스의 동작을 확인해보면 program2 실행 이후 program1이 2개의 프로세스에 할당되어 실행되고 있음을 알 수 있다.

[program 3]

√ 스레드의 개수를 1개로 지정해줄 경우



스레드 동작 확인 결과



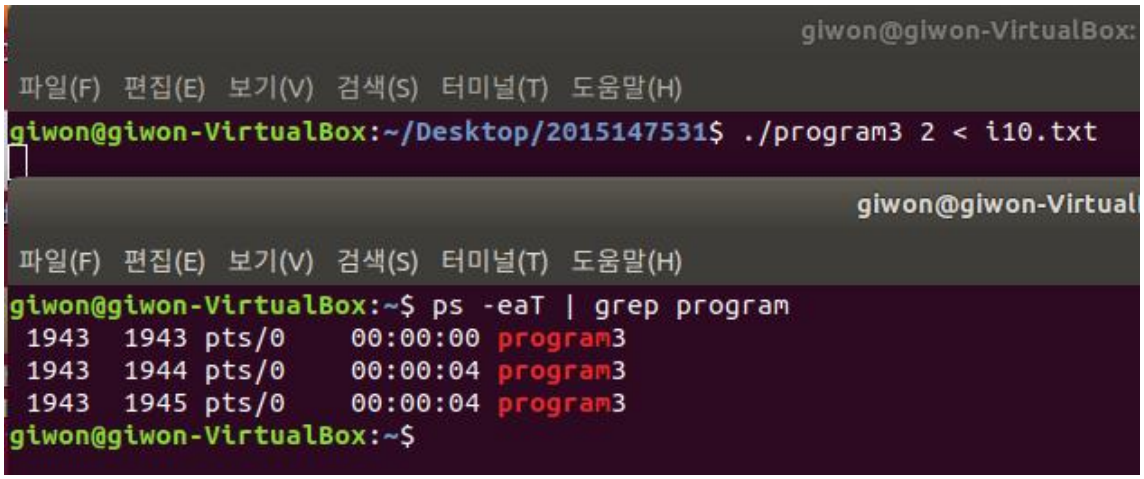
★ 시간 결과

메인 프로세스 내 실행 시간 : 103863 milliseconds

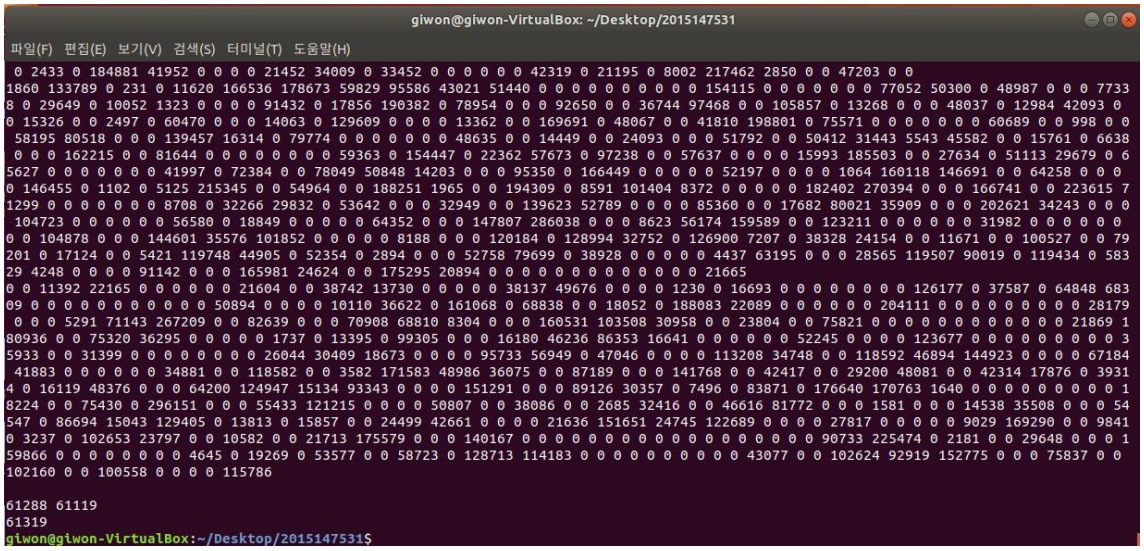
스레드 1 내 실행 시간 : 103832 milliseconds

- 메인 프로세스 내에서 스레드 1개가 동작하고 있기 때문에 동작 확인 결과에 program3가 2개 출력된다.

√ 스레드의 개수를 2개로 지정해줄 경우



스레드 동작 확인 결과



★ 시간 결과

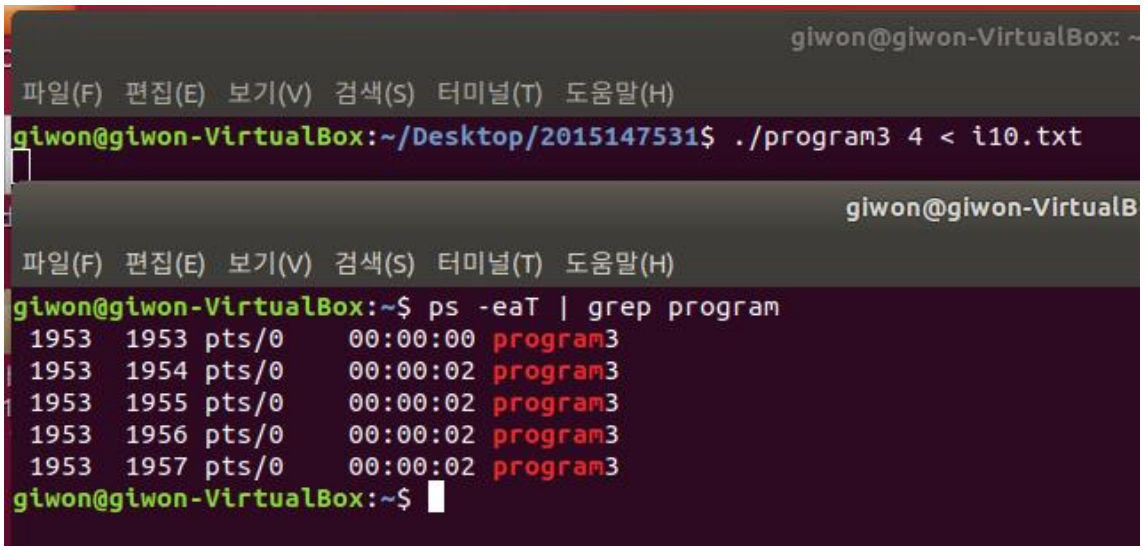
메인 프로세스 내 실행 시간 : 61319 milliseconds

스레드 1 내 실행 시간 : 61288 milliseconds

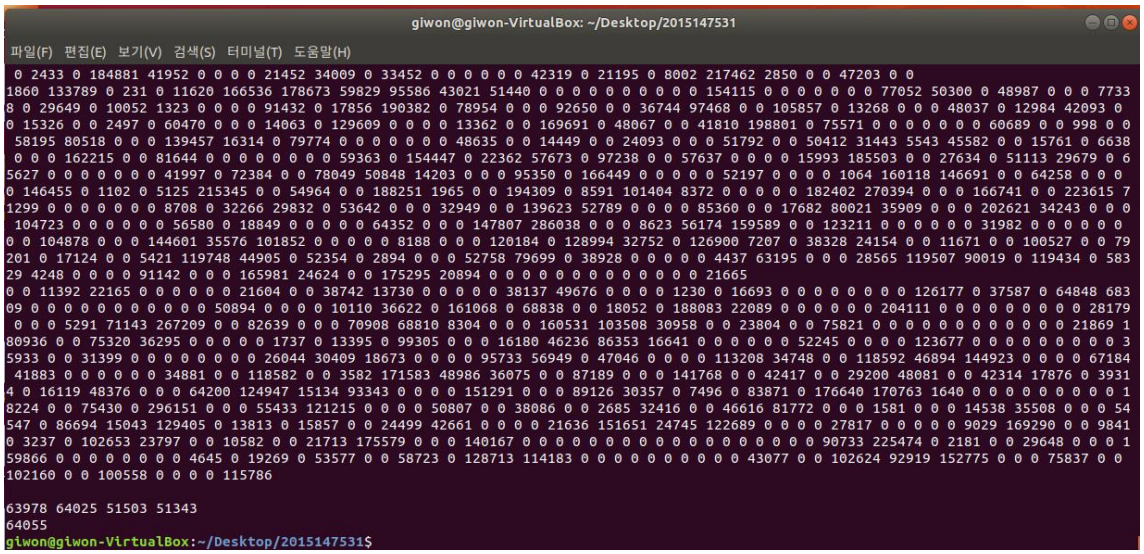
스레드 2 내 실행 시간 : 61119 milliseconds

- 메인 프로세스 내에서 스레드 2개가 동작하고 있기 때문에 동작 확인 결과에 program3가 3개 출력된다.

√ 스레드의 개수를 4개로 지정해줄 경우



스레드 동작 확인 결과



★ 시간 결과

메인 프로세스 내 실행 시간 : 64055 milliseconds

스레드 1 내 실행 시간 : 63978 milliseconds

스레드 2 내 실행 시간 : 64025 milliseconds

스레드 3 내 실행 시간 : 51503 milliseconds

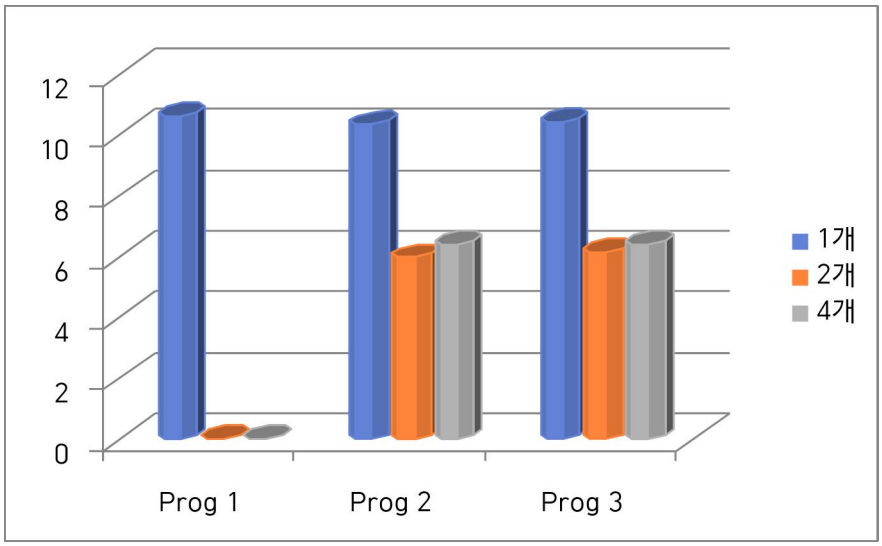
스레드 4 내 실행 시간 : 51343 milliseconds

- 메인 프로세스 내에서 스레드 4개가 동작하고 있기 때문에 동작 확인 결과에 program3가 5개 출력된다.

[각 결과값 비교]

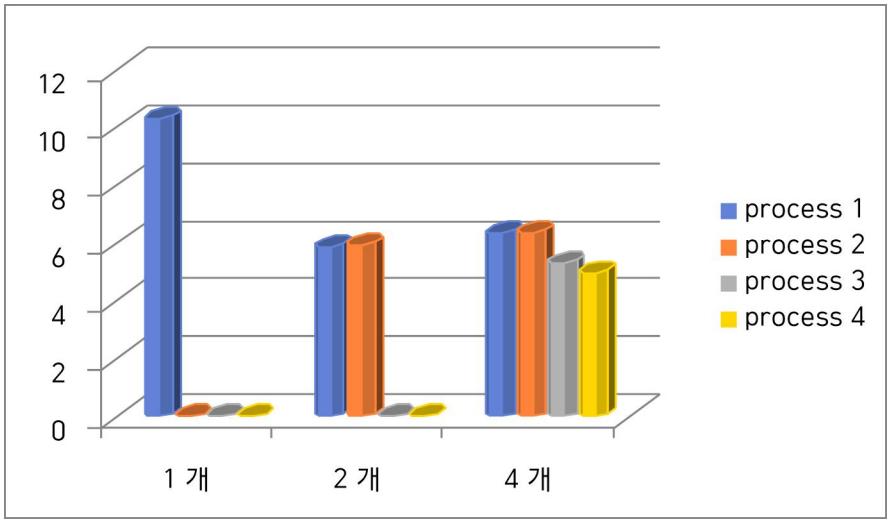
※ 가시적인 효과를 위하여 시간을 10000으로 나누어 표기하였다.

√ 각 프로그램의 프로세스 및 스레드 개수 별 전체 수행 시간 비교



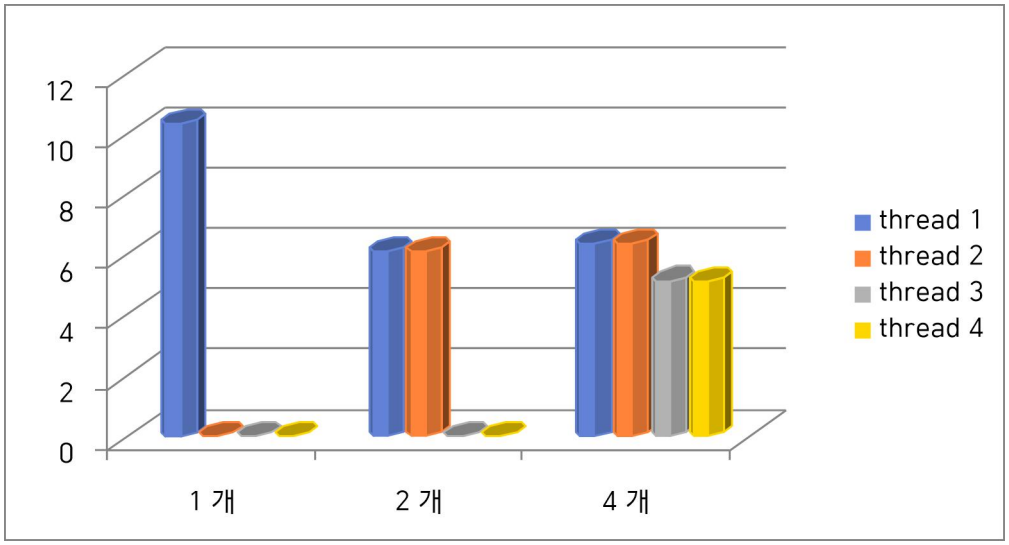
프로세스와 스레드 개수를 2개로 설정하여 실행할 경우 수행 시간이 절반으로 줄어들었다. 4개로 설정하여 실행할 경우 2개일 경우의 시간보다 줄어들 것으로 예상하였으나 컴퓨터의 cpu core 개수가 2개로 한정되어 있어 오히려 시간이 늘어난 것으로 보인다.

√ Program2의 프로세스 개수별 각 자식 프로세스의 수행 시간 비교



프로세스를 1개로 설정하였을 때보다 2개로 설정하였을 때 수행시간이 각각 절반으로 줄어들었으며 이는 각 프로세스별로 처리해야하는 필터의 개수가 절반으로 줄어들어 시간이 줄어들었다고 해석할 수 있다. 그러나 프로세스가 4개일 경우는 core 개수 부족으로 성능 향상을 결과 값으로 확인하지 못하고 오히려 실행 시간의 최대치가 증가한 것으로 측정되었다.

√ Program3의 스레드 개수별 각 스레드의 수행 시간 비교



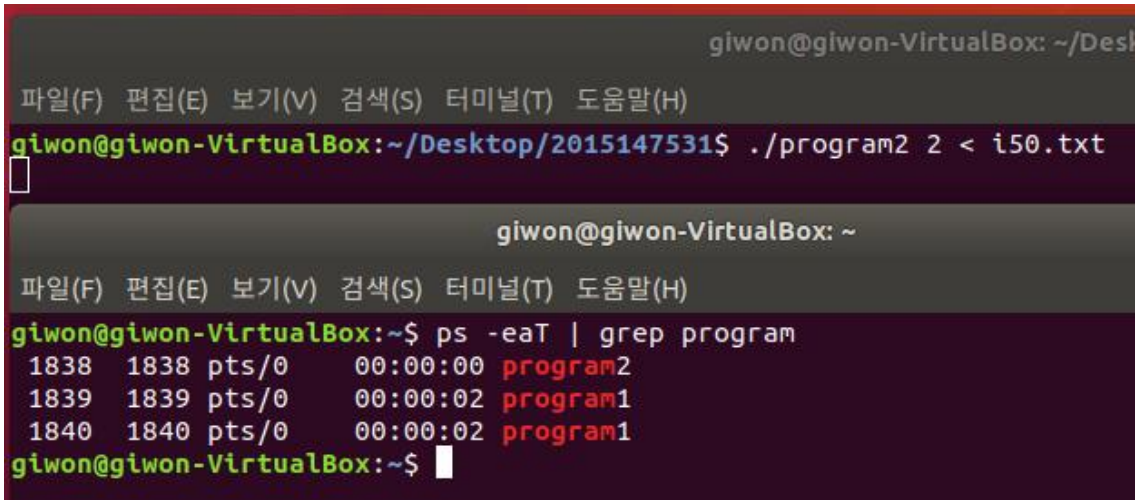
프로세스를 여러 개 실행시켜 프로그램을 실행시키는 program2와 마찬가지의 결과가 나왔다. 스레드를 1개 사용하여 프로그램을 실행시키는 시간보다 스레드를 2개 사용하여 프로그램을 실행시키는 시간이 더 적게 나왔으면, 4개의 스레드를 사용하는 경우 오히려 실행시간의 최대치가 더 높게 측정되었다.

2. 필터 : 50개 50행 50열 / 데이터 : 500행 500열

※ 위의 결과로 인하여 프로세스와 스레드의 개수를 1로 지정하는 것은 별다른 차이가 없다는 것을 확인하였기 때문에 프로세스와 스레드의 개수를 2개와 4개일 때를 측정해보았다.

[program 2]

√ 프로세스의 개수를 2개로 지정해줄 경우



프로세스 동작 확인 결과

★ 시간 결과

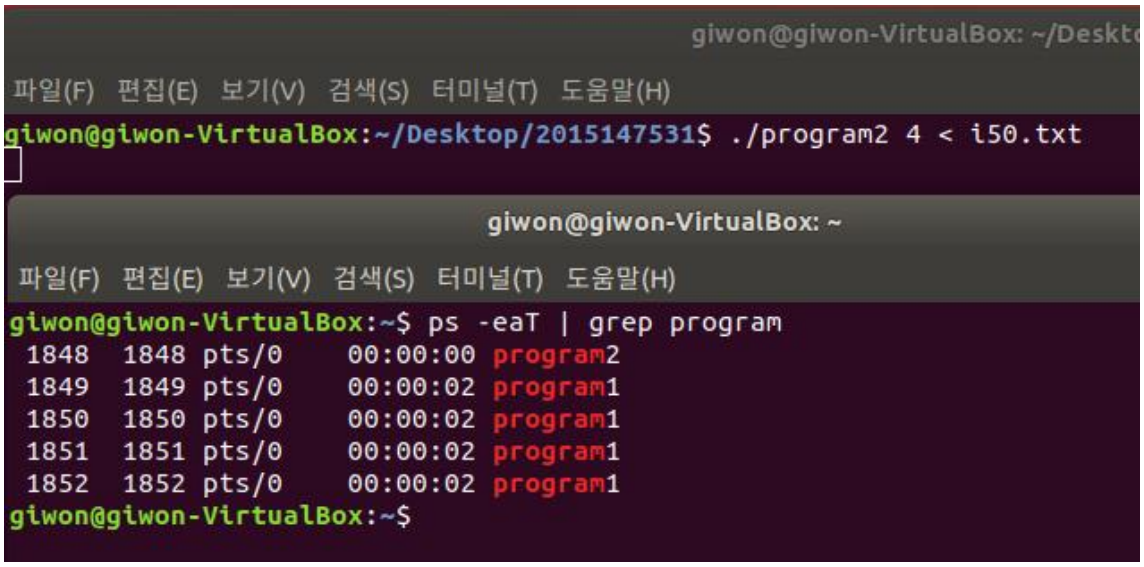
부모 프로세스 내 실행 시간 : 284949 milliseconds

자식 프로세스 1 내 실행 시간 : 284275 milliseconds

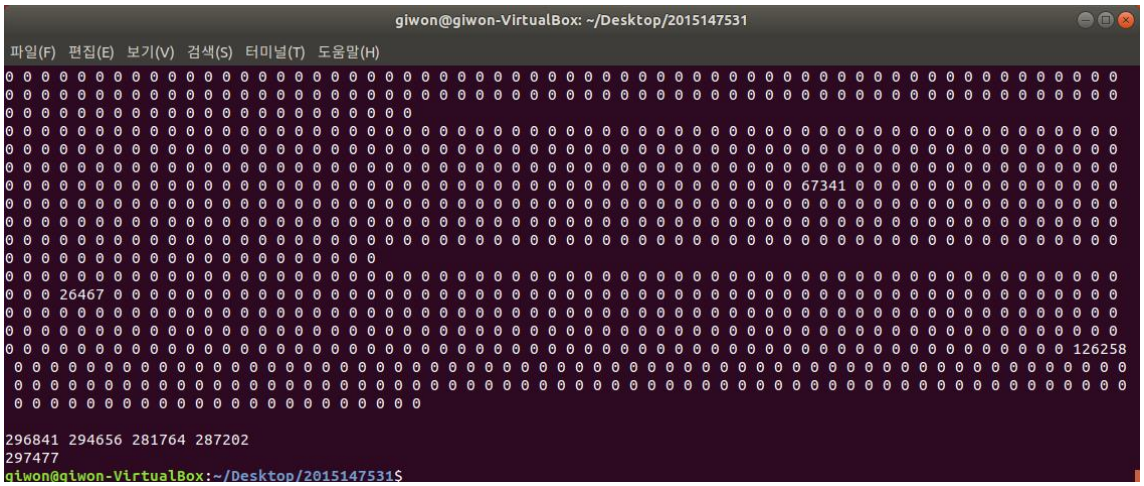
자식 프로세스 2 내 실행 시간 : 283614 milliseconds

- 실행 도중 프로세스의 동작을 확인해보면 program2 실행 이후 program1이 2개의 프로세스에 할당되어 실행되고 있음을 알 수 있다.

√ 프로세스의 개수를 4개로 지정해줄 경우



프로세스 동작 확인 결과



★ 시간 결과

부모 프로세스 내 실행 시간 : 297477 milliseconds

자식 프로세스 1 내 실행 시간 : 296841 milliseconds

자식 프로세스 2 내 실행 시간 : 294656 milliseconds

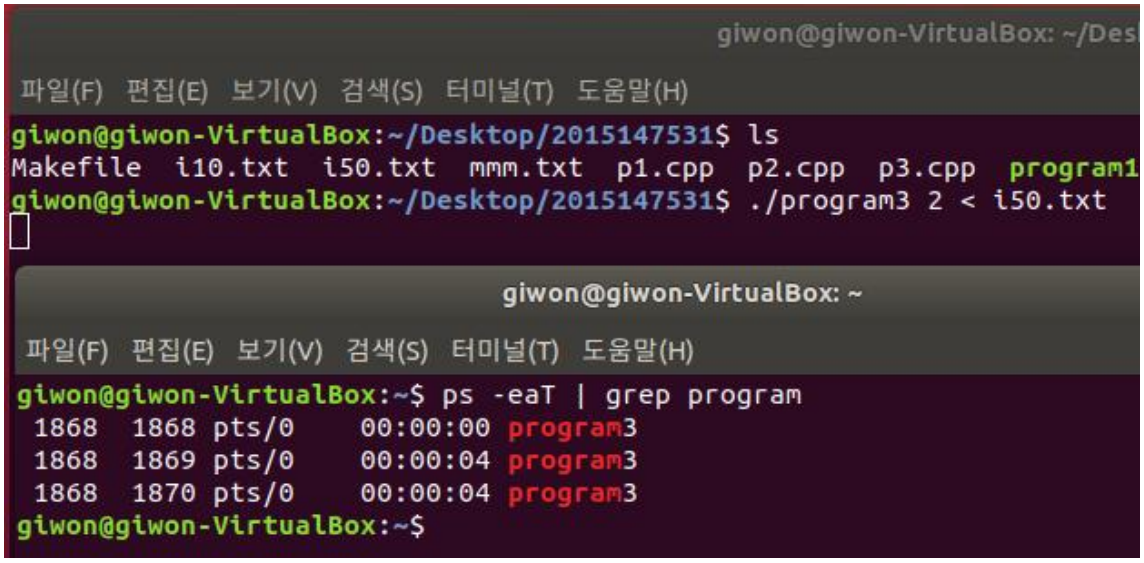
자식 프로세스 1 내 실행 시간 : 281764 milliseconds

자식 프로세스 2 내 실행 시간 : 287202 milliseconds

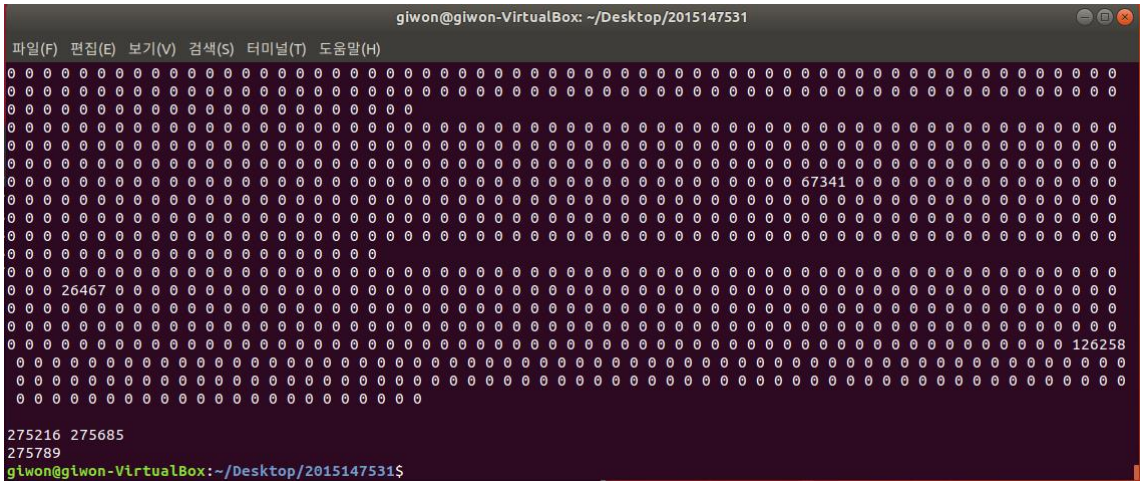
- 실행 도중 프로세스의 동작을 확인해보면 program2 실행 이후 program1이 2개의 프로세스에 할당되어 실행되고 있음을 알 수 있다.

[program 3]

√ 스레드의 개수를 2개로 지정해줄 경우



스레드 동작 확인 결과



★ 시간 결과

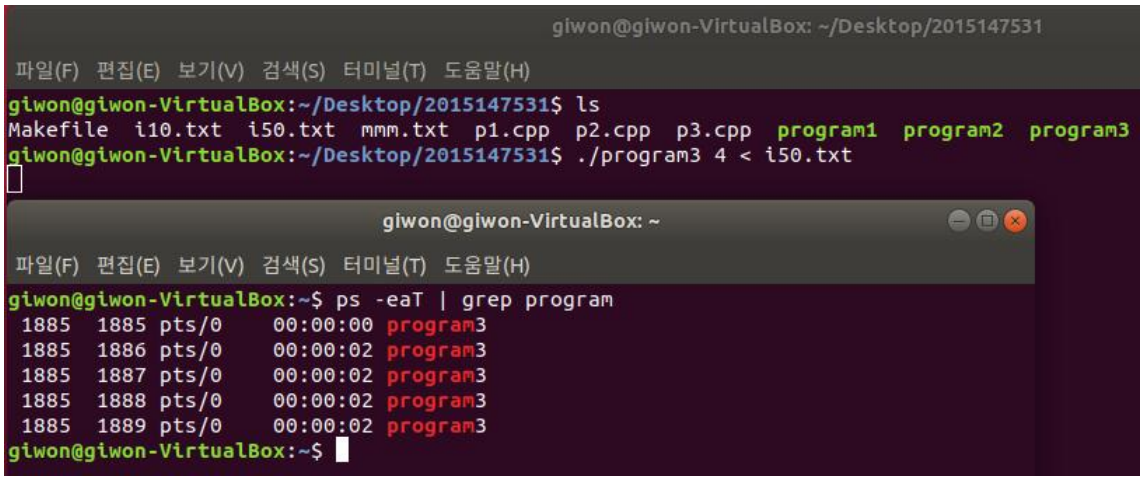
메인 프로세스 내 실행 시간 : 275789 milliseconds

스레드 1 내 실행 시간 : 275216 milliseconds

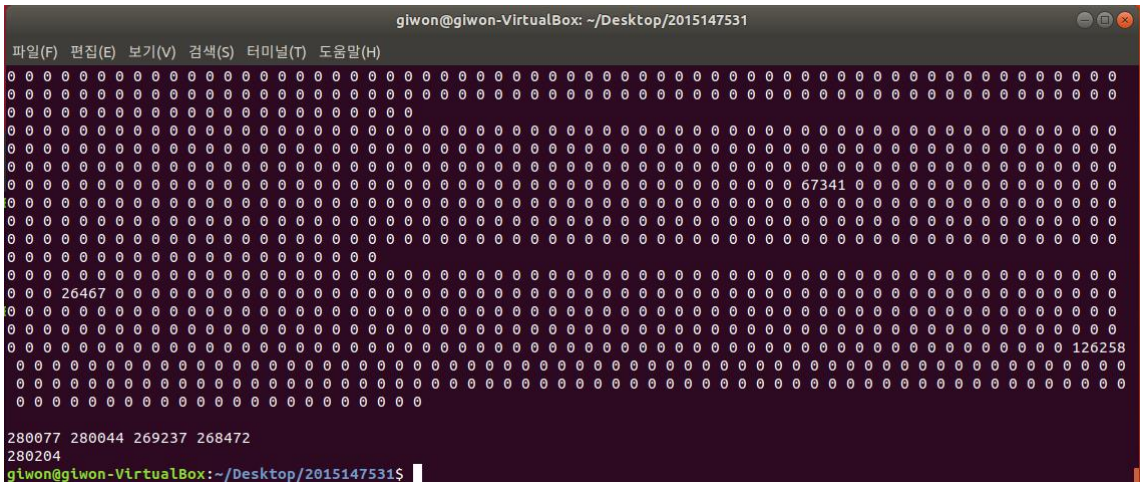
스레드 2 내 실행 시간 : 275685 milliseconds

- 메인 프로세스 내에서 스레드 2개가 동작하고 있기 때문에 동작 확인 결과에 program3가 3개 출력된다.

√ 스레드의 개수를 4개로 지정해줄 경우



스레드 동작 확인 결과



★ 시간 결과

메인 프로세스 내 실행 시간 : 280204 milliseconds

스레드 1 내 실행 시간 : 280077 milliseconds

스레드 2 내 실행 시간 : 280044 milliseconds

스레드 3 내 실행 시간 : 269237 milliseconds

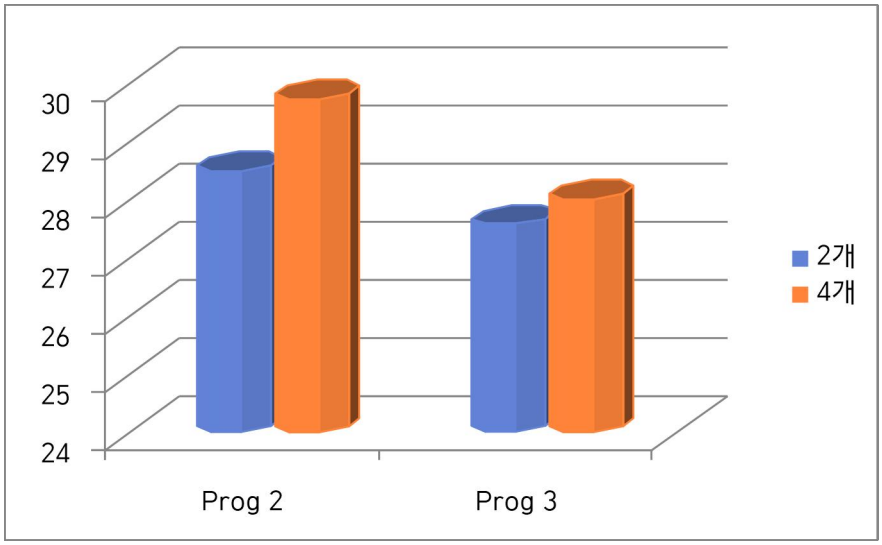
스레드 4 내 실행 시간 : 268472 milliseconds

- 메인 프로세스 내에서 스레드 4개가 동작하고 있기 때문에 동작 확인 결과에 program3가 5개 출력된다.

[각 결과값 비교]

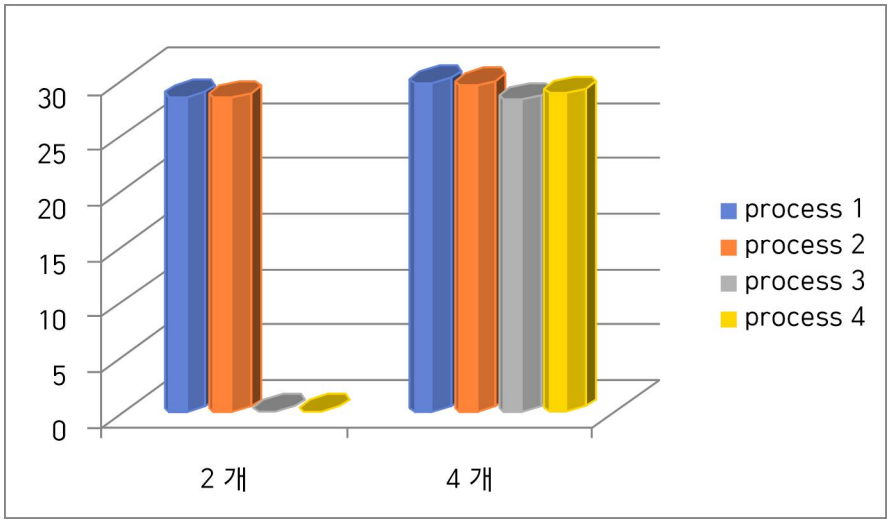
※ 가시적인 효과를 위하여 시간을 10000으로 나누어 표기하였다.

√ 각 프로그램의 프로세스 및 스레드 개수 별 전체 수행 시간 비교



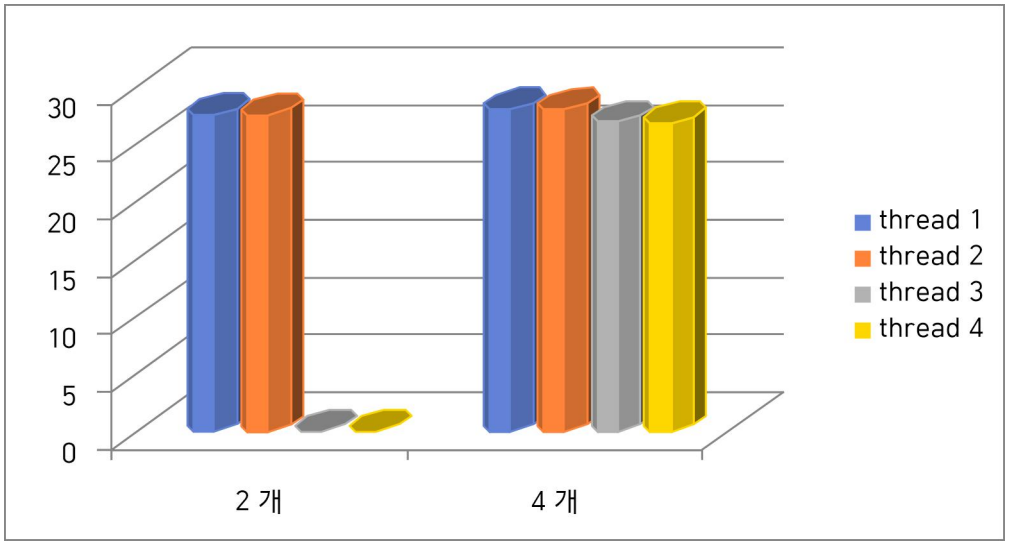
program2와 program3 모두 4개를 지정했을 때보다 2개를 지정했을 때의 수행 시간이 더 적은 것으로 측정되었다. CPU core의 개수가 2개이기 때문에 3개 이상의 프로세스와 스레드를 설정하는 효과를 확인하지 못하였다.

√ Program2의 프로세스 개수별 각 자식 프로세스의 수행 시간 비교



프로그램 전체 수행시간에서 확인했던 것과 비슷한 결과를 보여주었다.

√ Program3의 스레드 개수별 각 스레드의 수행 시간 비교



Program2와 비슷한 결과를 보여 주었다.

■ 최종 결과

멀티 프로세스와 멀티 스레딩 기술을 사용하면 단일 프로레스나 스레드를 사용하였을 때보다 수행시간이 크게 줄어든다는 장점이 있다. 하지만 컴퓨터의 성능(Cpu core)에 따라 프로세스와 스레드의 개수를 너무 높이면 오히려 수행시간이 늘어나는 점을 알 수 있었다.