

사전 조사 및 프로그래밍 수행 결과 보고서

-Assignment 2-



과목명		운영체제
담당 교수		차호정 교수님
학과		교육학과
학번		2018182058
이름		박주언
제출일		2020 년 5 월 1 일

사전조사 보고서

🖥️ 프로세스와 스레드

✓ 프로세스(Process)

프로세스는 시스템에서 현재 실행 중인 프로그램을 의미한다. 프로세스는 다른 말로 작업(job) 또는 태스크(task)라고 불리기도 한다. 작업이란 보통 사용자가 프로그램을 돌리기 위해 작성한 프로그램 코드와 그에 필요한 부수적인 입력 데이터들을 아울러 이르는 말이다. 그런데 프로그램이 실행되고 있는 프로세스가 되려면, 디스크에 위치해 있던 프로그램 코드가 메모리에 할당되어 운영체제, 즉 커널의 제어를 받는 상태가 되어야 한다. 메모리에 할당되어 자신만의 주소 공간을 가지고, 커널로부터 시스템 자원을 할당 받아 실행이 되는 것이다. 다시 말하자면 작업이 컴퓨터 시스템에 의해 커널에 전달되어 커널의 제어 하에서 실행이 되는 프로그램 인스턴스가 프로세스이며, 일종의 주기억장치 실행단위라고 볼 수 있다. 프로세스는 커널에 등록된 작업이라고 볼 수 있기 때문에 작업과 프로세스를 동일시하여 부르기도 하는 것이다. 이런 모든 것들을 간단하게 함축하여 프로세스를 '**실행 중인 프로그램**'이라고 정의 내린다.

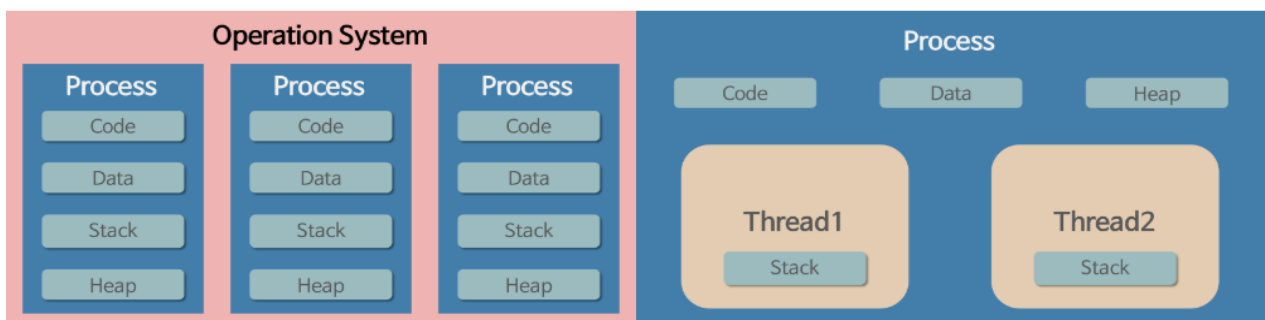
✓ 스레드(Thread)

스레드는 프로세스 안에서 실행되고 있는 **여러 흐름의 단위**이다. 모든 프로세스는 최소 1개의 메인 스레드를 가지고 있는데, 프로세스는 할당 받은 자원을 일종의 프로그램 실행 경로인 스레드를 통해 이용한다. 따라서 스레드는 프로세스의 자원을 사용하여 실제로 작업을 수행하는, 일종의 경량화 된 프로세스라고 볼 수 있는 것이다.

✓ 프로세스와 스레드 차이

프로세스가 실행되기 위해서는 다양한 부수적 정보들이 필요하다. 각각의 프로세스는 코드, 데이터, 스택, 힙을 포함하는 고유한 프로세스 이미지를 받아 독립된 메모리 영역에서 실행된다. 모든 프로세스는 이러한 이미지 정보를 가지고 고유한 공간에서 작업을 수행하기 때문에 다른 프로세스의 자료나 변수 등에 접근할 수가 없다.

반면에 같은 프로세스 안에 있는 여러 스레드들은 프로세스의 코드, 데이터, 힙은 공유를 하면서 각자 독자적인 스택만을 가진다. 스레드가 여러가지 메모리를 공유하고 있기 때문에, 하나의 스레드는 또 다른 스레드의 자원에 접근이 가능하여 간단한 방식으로 서로 통신할 수 있다. 단순히 공유되고 있는 정보의 변수 정보를 수정하는 식으로 통신을 구현할 수 있기 때문이다. 반면에 프로세스는 각기 다른 주소 공간에서 각기 다른 자료구조를 가지고 있기 때문에, 다른 프로세스의 자원에 접근하는 것이 쉽지 않다. 프로세스는 따라서 프로세스 간 통신(IPC, inter process communication)을 사용하여 서로 정보를 주고받는다. IPC에는 소켓, 파이프, 파일 등이 있다. 따라서 조금 더 보안적인 측면에서 안전하게 통신 할 수 있다.

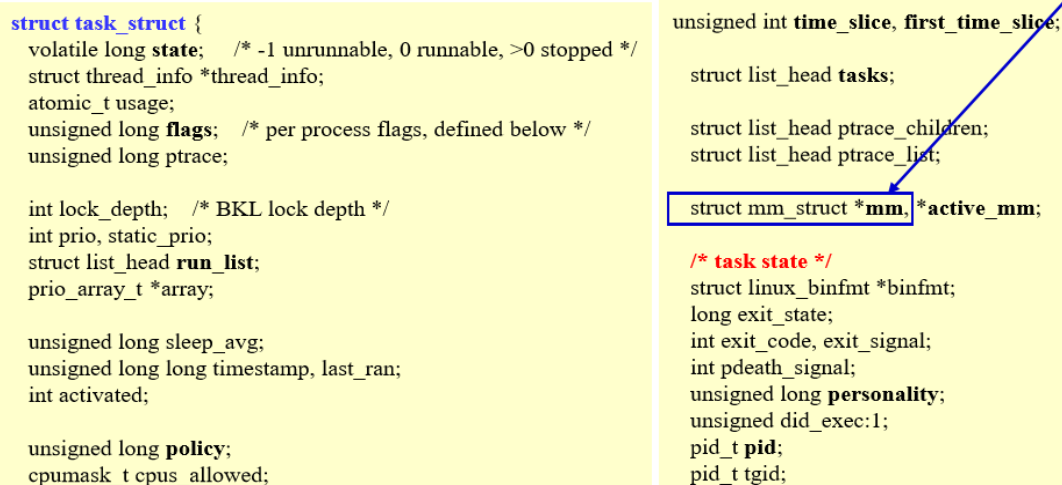


[그림 1] 프로세스(좌측)는 코드, 데이터, 스택, 힙을 모두 고유한 이미지로써 가지지만, 스레드는 자신의 스택만 가질 뿐 나머지 자원은 공유한다.

이처럼 프로세스와 스레드의 가장 근본적인 차이점은 바로 '공유'의 정도이다. 프로세스는 독립된 공간에서 각자의 자원을 할당 받아 모두 따로따로 실행되지만, 스레드는 하나의 프로세스 내부에서 주소공간을 포함한 다양한 자원을 공유하면서 한 번에, 즉 병렬적으로 업무를 수행한다. 따라서, 만약 코드가 딱 한 줄만 다르더라도 프로세스는 매번 새로운 프로세스를 생성하여 시스템 자원을 할당 받아야하기 때문에 시스템 자원소모도 심하고, 그만큼 작업 수행시간이 길다. 반면 스레드는 많은 공간을 공유하기 때문에 자원을 조금만 써서 효율적일 뿐만 아니라 그만큼 프로그램의 응답 속도도 빨라진다. 특히 Context switching 시에 드는 자원, 시간, 처리 비용 등이 보통의 경우 스레드가 프로세스보다 훨씬 적기 때문에 오버헤드가 적게 일어난다.

✓ 프로세스와 스레드의 구현 및 구조

리눅스에서는 프로세스와 스레드를 구분하지 않고 한데 묶어 태스크(Task)라고 부른다. 스레드와 프로세스가 구현적으로 크게 다르지 않기 때문에 작업의 실행단위를 태스크라고 묶어 부르는 것이다. 리눅스는 프로세스와 스레드를 처리하기 위해 각 프로세스마다 **task_struct** 라는 자료구조를 생성한다. 같은 이름이지만 자원이 얼마나 공유 되는가에 따라 프로세스가 되기도 하고 스레드가 되기도 하는 것이다.



```

struct task_struct {
    volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
    struct thread_info *thread_info;
    atomic_t usage;
    unsigned long flags; /* per process flags, defined below */
    unsigned long ptrace;

    int lock_depth; /* BKL lock depth */
    int prio, static_prio;
    struct list_head run_list;
    prio_array_t *array;

    unsigned long sleep_avg;
    unsigned long long timestamp, last_ran;
    int activated;

    unsigned long policy;
    cpumask_t cpus_allowed;

    unsigned int time_slice, first_time_slice;

    struct list_head tasks;

    struct list_head ptrace_children;
    struct list_head ptrace_list;

    struct mm_struct *mm, *active_mm;

    /* task state */
    struct linux_binfmt *binfmt;
    long exit_state;
    int exit_code, exit_signal;
    int pdeath_signal;
    unsigned long personality;
    unsigned did_exec:1;
    pid_t pid;
    pid_t tgid;

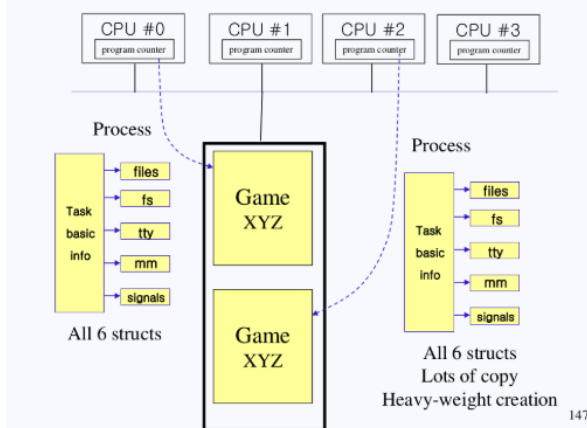
```

[그림 2] task_struct의 일부분(출처 : 강의 노트)

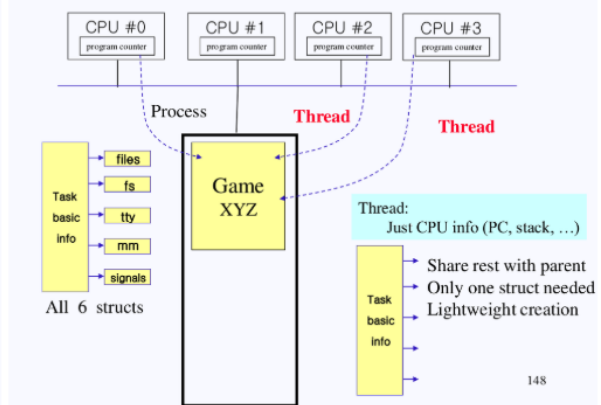
리눅스가 부팅이 되면, 부트 로더가 커널을 작동시킨다. 이 커널이 바로 모든 프로세스의 부모 프로세스이자 root 프로세스인 init 프로세스를 실행시킨다. 이 init 프로세스가 사용자가 직접 코드를 입력하고 수행할 수 있게 하는 shell 프로세스를 만든다. 이 shell이 만약 fork() 라는 system call을 통해 부모 프로세스와 똑같은 정보를 가진 자식 프로세스가 생성된다. 이때 exec() 시스템콜을 통해 사용자는 자식 프로세스가 부모 프로세스와 다른 코드를 실행할 수 있게끔 한다. 이렇게 shell 이 사용자와 상호작용하며 fork() 시스템콜을 통해 프로세스를 만들어내고 없애면서, 부모-자식 위계를 가진 일종의 트리구조와 같이 프로세스는 구현된다.

만약 이런 식으로 새로운 작업을 처리해야할 필요가 있을 때마다 fork() 콜을 사용하여 프로세스를 계속해서 만들어낸다면 시스템 자원이 굉장히 비효율적으로 사용될 것이며, 상당히 큰 오버헤드가 발생할 것이다. 따라서 이런 오버헤드를 해결하기 위해 생긴 것이 스레드인데, 스레드도 프로세스와 비슷한 방식으로 새로운 스레드를 만들기는 한다. 대신 프로세스는 fork()콜을 사용했다면 스레드는 clone()이라는 시스템 콜을 이용해 스레드를 생성해 여러 작업을 처리할 수 있다. clone() 콜이 불리면, 부모와 자식, 즉 새로 생기는 스레드가 얼마나 많은 정보를 기존의 것과 공유할 것인지 결정하는 flags를 인자로 보내스레드를 생성한다. 만약 아무런 flag도 보내지지 않는다면, clone()은 모든 정보를 독립적으로 가지는 태스크를 생성하기 때문에 fork()와 비슷한 동작을 하게 된다. 리눅스의 task_struct를 보면, 많은 부분들이 포인터로 이루어져 있는 것을 확인할 수 있다. 따라서 clone() 콜을 통해 생성된 스레드는, 여러가지 정보에 접근할 수 있는 포인터 정보만 가지고 사용하기 때문에, 프로세스보다 훨씬 오버헤드가 적고 가볍게 동작한다. 그래서 스레드를 경량화된 프로세스라고 부르기도 한다.

Comparison: Creating child as a **Process**



Comparison: Creating Child as a **Thread**



[그림 3] 이런식으로 프로세스는 모든 정보를 따로 갖는 프로세스를 매번 생성하고, 스레드는 여러가지 정보를 공유하는 태스크를 만든다는 차이점이 있다. (출처 : <https://talkingaboutme.tistory.com/entry/Study-Linux%EC%97%90%EC%84%9C%EC%9D%98-Process-%EC%99%80-Thread>)

멀티 프로세스와 멀티 스레딩

✓ 멀티 프로세스와 멀티 스레딩의 개념

사용자는 한 번에 하나의 작업을 하면서 컴퓨터가 그를 다 처리할 때까지 기다렸다가 다른 작업을 수행하지 않기 때문에, 운영체제 입장에서는 다양한 작업을 컨트롤할 수 있는 여러가지 태스크들이 필요하다. 또한 하나의 응용프로그램에 대해서도, 하나의 태스크가 아니라 여러가지 태스크들이 함께 일을 병렬적으로 수행한다면 더더욱 실행시간이 단축된다.

멀티 프로세스는 여러가지 프로그램을 여러 개의 프로세스에게 할당하여 처리하게 함으로써 수행시간을 높이는 기법을 말한다. fork()를 통해 생성된 많은 프로세스들이 각자 작업을 맡아 IPC를 통해 통신하며 일을 수행하는 것이다. 하지만 기본적으로 하나의 CPU는 한 가지 프로세스밖에 처리하지 못한다. 시대가 변화하며 멀티코어 등의 고성능 하드웨어가 개발되어도 하나의 프로세스를 여러 CPU가 나눠서 처리할 수 없기 때문에 그 이점을 이용하기 어렵고, 따라서 단일 프로세스의 처리속도 또한 향상하기 어려운 문제가 발생한다.

이러한 상황을 멀티 스레딩으로 보완할 수 있다. 멀티 스레딩은 하나의 프로세스를 여러개로 나누어서 하나의 프로그램을 병렬화 하는 기법이다. 한 프로세스 내에서 생성된 여러 스레드들은 멀티 프로세스와 다르게 RAM에 저장된 프로세스의 이미지의 많은 부분을 공유하고 있기 때문에 각각 다른 코드를 각기 다른 CPU에서 나누어 처리하는 것이 가능하다. 멀티 스레딩을 사용하면 프로그램을 병렬화해서 서비스의 속도는 물론 프로세스 처리 속도도 향상시킬 수 있는 것이다.

✓ 멀티 프로세스와 멀티 스레딩의 비교

여러가지 프로세스와 스레드를 처리하다 보면, 어떤 프로세스가 점유하고 있던 CPU 사용을 멈추고 다른 프로세스가 실행될 경우가 있다. 두 번째로 실행되던 프로세스가 실행을 멈추고 다시금 첫 번째로 돌아가야하는 경우를 대비하여, 프로세스는 작업을 중단할 때의 프로세스 Context 정보를 PCB(Program Control Block)에 저장하게 된다. 이후 해당 프로세스의 실행 차례가 되면 PCB에 저장된 정보를 불러와 그 상태에서 작업을 재개하게 되고, 이렇게 Context를 바꿔가며 여러 프로세스를 왔다 갔다 하는 것을 Context Switching이라고 부른다. 멀티 프로세스를 다루게 되면 상당히 많은 빈도로 Context Switching이 일어날텐데, 그 때마다 스택 포인터나, 프로그램 카운터 등을 추적하여 많은 명령을 처리하기 때문에 컴퓨터에 상당한 오버헤드가 발생한다.

이런 상황에서, 만약 싱글 스레드로 구성된 멀티 프로세스들만 있는 컴퓨터의 경우, 각 프로세스들이 가진 Context가 모두 다르기 때문에 자원낭비도 심하고 오버헤드도 심할 것이다. 반면에, 멀티 스레딩을 활용하는 멀티 프로세스의 경우에는 스레드가 많은 정보를 공유하며 각자 작은 양의 작업을 맡아 처리하기 때문에 Context Switching시에 발생하는 오버헤드도 적고, 처리 시간도 빠르다.

하지만 멀티 스레딩이 무조건적인 장점만 있는 것은 아니다. 멀티 스레딩은 한 프로세스의 작업을 각자 나누어 처리한 뒤 그 작업물을 한데 모아서 다시 합쳐야 하기 때문에, 만약 하나의 스레드에서 문제가 발생하였을 경우에는 전체 프로세스에 문제가 생기게 된다. 이처럼 동기화가 까다롭기 때문에 주의 깊은 설계가 필요하고 디버깅도 힘들다. 반면 여러 개의 프로세스가 실행 중 하나의 자식 프로세스에 문제가 발생하더라도 각자 병렬화 되어있고 연관이 없기 때문에 나머지 프로세스들에 영향을 미치지 않아 보안 관점에서 멀티 스레딩보다 이점이 있다.

✓ 멀티 프로세스와 멀티 스레딩의 구현

멀티 프로세스는 여러 개의 CPU를 장착하여 각 CPU가 하나의 프로세스를 담당하게 하는 방식으로 처리할 수 있다. 앞서 설명한 **fork()**콜과 **exec()** 계열의 시스템콜을 적절히 활용하여 여러 작업을 다룰 수 있는 자식 프로세스들을 만들고 그를 병렬화 하여 다양한 프로그램을 실행한다. 프로세스를 구분할 수 있는 pid가 부모, 자식 프로세스마다 다르기 때문에 fork() 콜을 한뒤 pid가 0인지 아닌지의 여부에 따라 수행하는 코드 블록이 달라지게 함으로써 다중 프로세스를 통해 여러가지 작업을 수행할 수 있다. 혹은 exec() 콜을 사용하여 새로운 프로그램을 실행시킬 수도 있다.

반면 멀티 스레딩은 운영체제에서 기존에 없던 개념을 새로 만들어내는 것이기 때문에 다양한 구현 방법이 논의되었다. 먼저 유저레벨에서 처리할 수 있도록 라이브러리를 만들어 구현하는 시도가 있었다. 이는 커널 레벨의 변경을 최소화할 수 있다는 점에서 장점이 있지만 실질적으로는 하드웨어의 병렬성의 이점을 취하지 못하기 때문에 실효성이 떨어진다는 평가를 받았다. 따라서 커널 레벨에서 프로세스 테이블과 함께 스레드 정보를 담는 스레드 테이블도 집어넣어 구현을 해야 했다. 커널 레벨에서는 'one to one' 모델 혹은 'many to many' 모델 혹은 둘 다를 혼합하여 유저레벨의 스레드와 커널 레벨의 스레드를 적절히 연결하여 멀티 스레딩을 구현하고자 하는 시도가 있었고 실제로 많은 운영체제들이 해당 방법을 사용하여 멀티 스레딩을 구현했다. 리눅스 같은 경우에는 커널의 변화를 최소화하기 위해 프로세스와 스레드를 구분 짓지 않았다. 모든 운영체제에서 멀티 스레딩 기능을 제공할 때, 통일성과 이식성을 높이기 위해 **pthread** 라는 API를 공용으로 사용하자고 약속하였다. 따라서 현존하는 운영체제에서 멀티 스레딩을 사용하려면, pthread_create, pthread_join 등의 명령어를 사용하여 구현해낼 수 있다.

```
int pthread_create (pthread_t *tid,
pthread_attr_t *attr,
void *(start_routine)(void *),
void *arg);

void pthread_exit (void *retval);

int pthread_join (pthread_t tid,
void **thread_return);
```

Thread creation/termination

```
int pthread_cond_init
(pthread_cond_t *cond,
const pthread_condattr_t *cattr);

void pthread_cond_destroy
(pthread_cond_t *cond);

void pthread_mutex_wait
(pthread_cond_t *cond,
pthread_mutex_t *mutex);

void pthread_cond_signal
(pthread_cond_t *cond);

void pthread_cond_broadcast
(pthread_cond_t *cond);
```

Condition variables

```
int pthread_mutex_init
(pthread_mutex_t *mutex,
const pthread_mutexattr_t *mattr);

void pthread_mutex_destroy
(pthread_mutex_t *mutex);

void pthread_mutex_lock
(pthread_mutex_t *mutex);

void pthread_mutex_unlock
(pthread_mutex_t *mutex);
```

Mutexes

[그림 4] 멀티 스레딩 구현 명령어 예시

📱 exec() 계열 시스템 콜

✓ exec() 계열 시스템콜과 그 구현

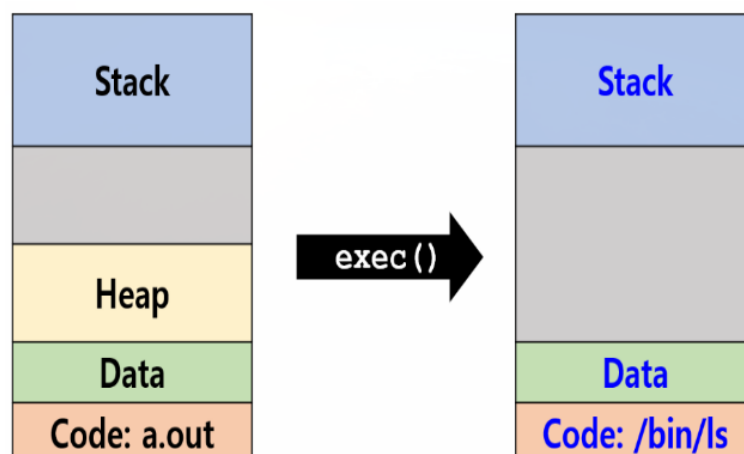
```
int execl(const char *pathname, const char *arg0, ... /* (char*) 0 */);
int execlp(const char *pathname, const char *arg0, ... /* (char*) 0 */);
int execlx(const char *pathname, const char *arg0, ...
            /* (char*) 0, char *const envp[] */);
int execve(const char *pathname, const char *argv[], char *const envp[]);
int execlp(const char *filename, const char *arg0, ... /* (char*) 0 */);
int execlvp(const char *filename, const char *arg0, ... /* (char*) 0 */);
```

[그림 5] exec() 계열의 시스템 콜

exec() 계열의 시스템콜에는 위 6가지 명령어가 있다. exec() 계열의 시스템 콜은 현재의 프로세스를 인자로 전달된 프로그램으로 대체해버리는 일을 한다. 현재 프로세스의 메모리 공간을 새로운 프로세스로 덮어 씌우는 것이다. 이렇게 새롭게 호출된 프로그램은 코드의 첫 줄부터 수행하게 된다. exec() 계열은 다시 execl 계열과 execlx 계열로 나눌 수 있다. 둘의 가장 큰 차이점은 바로 전달하는 인자의 형태이다. execl 계열은 두번째 인자로 char형 포인터인 arg를 설정했고, execlx 계열은 두번째 인자로 char형 포인터 배열인 argv를 보내고 있다. 대체할 프로세스의 인수로서 포인터를 보내는지, 포인터 배열을 보내는지의 차이에 따라 execl 과 execlx 로 나눌 수 있는 것이다. 다만 execlx의 경우 이 argv 인자의 마지막 원소로 NULL값을 추가해줘야 한다는 주의점이 있다. 그리고 execl역시 더 이상 인수가 없다는 것을 나타내기 위해 (char*) 0, 즉 NULL 포인터를 작성해주어야 한다.

그 외에 뒤에 붙는 접미사는 공통적이다. e 가 붙는 경우에는 마지막 인자로 환경변수를 넘겨준다. 이 가변인수를 통해 새롭게 대체될 프로그램의 환경변수를 설정해줄 수 있다. 그리고 p 가 붙는 경우에는 파일의 절대경로를 입력할 필요가 없게 된다. p가 붙은 시스템콜의 경우 알아서 환경변수 PATH를 참조하기 때문이다. 그렇기 때문에 6가지 시스템콜의 공통적인 첫번째 인자에, 경로를 작성하되 execlp와 execlvp는 파일이름을 작성하면 되는 것이다.

즉 exec() 계열 시스템 콜은 첫번째 인자로 넘겨받은 경로명이나 파일이름을 가진 프로그램을, 뒤에 오는 인자들과 환경변수를 참고하여 콜을 한 프로세스의 메모리 영역에 덮어씌우는 역할을 하는 명령어라고 볼 수 있는 것이다. 예를 들어 a.out 프로세스 실행 중 execl("/bin/lx", "/bin/lx", "-al", "/tmp", NULL) 이라는 커맨드를 입력하여 exec콜을 수행한다고 가정해보자. 그러면 리눅스 커널 내부적으로 오른쪽 그림과 같은 과정에 의해 프로세스가 대체 되어 새로운 코드가 처음부터 실행되게 될 것이며, a.out에 남아있던 뒤 줄의 코드는 실행되지 않게 되는 것이다. 그 외에 자세한 커널 내의 구현 방식은 리눅스 오픈 소스 깃허브에서 찾아볼 수 있다.



[그림 6] exec() 콜의 실제 구현 예시

프로그래밍 수행 결과 보고서

개발 환경

✓ 메모리 정보

```
mingbee@mingbee-VirtualBox:~$ free -h
```

	total	used	free	shared	buff/cache	available
Mem:	3.8G	910M	2.3G	28M	651M	2.7G
Swap:	1.4G	0B	1.4G			

\$free -h 명령어를 사용해 메모리 사용 현황을 확인했다. -h 옵션으로 가독성을 높였다.

```
mingbee@mingbee-VirtualBox:~$ cat /proc/meminfo
```

MemTotal:	4029820 kB
MemFree:	2339316 kB
MemAvailable:	2821128 kB
Buffers:	97380 kB
Cached:	573548 kB
SwapCached:	0 kB
Active:	1119648 kB
Inactive:	355268 kB
Active(anon):	804904 kB
Inactive(anon):	28384 kB
Active(file):	314744 kB
Inactive(file):	326884 kB
Unevictable:	32 kB
Mlocked:	32 kB
SwapTotal:	1459804 kB
SwapFree:	1459804 kB
Dirty:	0 kB
Writeback:	0 kB
AnonPages:	804112 kB
Mapped:	241640 kB
Shmem:	29300 kB
KReclaimable:	86488 kB
Slab:	121616 kB
SReclaimable:	86488 kB
SUnreclaim:	35128 kB
KernelStack:	7808 kB
PageTables:	30004 kB
NFS_Unstable:	0 kB
Bounce:	0 kB
WritebackTmp:	0 kB
CommitLimit:	3474712 kB
Committed_AS:	3792756 kB
VmallocTotal:	34359738367 kB
VmallocUsed:	32920 kB
VmallocChunk:	0 kB
Percpu:	2592 kB
HardwareCorrupted:	0 kB
AnonHugePages:	0 kB
ShmemHugePages:	0 kB
ShmemPmdMapped:	0 kB
CmaTotal:	0 kB
CmaFree:	0 kB
HugePages_Total:	0
HugePages_Free:	0
HugePages_Rsvd:	0
HugePages_Surp:	0
Hugepagesize:	2048 kB
Hugetlb:	0 kB
DirectMap4k:	137152 kB
DirectMap2M:	4057088 kB

\$cat /proc/meminfo 명령어를 사용해 전체 메모리 정보 또한 확인할 수 있었다.

✓ CPU 정보

```
mingbee@mingbee-VirtualBox:~$ cat /proc/cpuinfo
```

processor	: 0
vendor_id	: GenuineIntel
cpu family	: 6
model	: 142
model name	: Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
stepping	: 10
cpu MHz	: 1992.001
cache size	: 8192 KB
physical id	: 0
siblings	: 6
core id	: 0
cpu cores	: 6
apicid	: 0

\$cat /proc/cpuinfo 명령어를 사용해 CPU 정보를 확인했다. CPU 코어는 6 개다.

✓ 컴파일러 버전

```
mingbee@mingbee-VirtualBox:~$ gcc --version
gcc (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.
```

\$ gcc --version 명령어를 사용하여 gcc 컴파일러의 버전을 확인하였다.

```
mingbee@mingbee-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.
```

\$ g++ --version 명령어를 사용하여 g++ 컴파일러의 버전을 확인하였다.

채점환경과 동일한 컴파일 환경을 만들기 위해, gcc 5, g++ 5 패키지도 설치하여 테스트해보았다.

```
mingbee@mingbee-VirtualBox:~$ sudo update-alternatives --config gcc
There are 2 choices for the alternative gcc (providing /usr/bin/gcc).

  Selection    Path                        Priority  Status
  ----
  0            /usr/bin/gcc-7              80      auto mode
* 1            /usr/bin/gcc-5              60      manual mode
  2            /usr/bin/gcc-7              80      manual mode
```

```
mingbee@mingbee-VirtualBox:~$ sudo update-alternatives --config g++
There are 2 choices for the alternative g++ (providing /usr/bin/g++).

  Selection    Path                        Priority  Status
  ----
  0            /usr/bin/g++-7              80      auto mode
* 1            /usr/bin/g++-5              60      manual mode
  2            /usr/bin/g++-7              80      manual mode
```

각각 \$sudo update-alternatives --config gcc 그리고 \$sudo update-alternatives --config g++ 커맨드로 우분투에 깔려 있는 여러가지 컴파일러 버전을 확인할 수 있었다. 버전을 확인하는 명령어를 입력하는 경우에는 priority 가 높은 7.5 버전이 출력되게 설정하였다.

✓ 시스템 정보

```
mingbee@mingbee-VirtualBox:~$ uname -a
Linux mingbee-VirtualBox 5.3.0-46-generic #38~18.04.1-Ubuntu SMP Tue Mar 31 04:17:56 UTC 2020 x86_64 x86_64 x86_64 GNU/Linux
```

\$uname -a 명령어를 사용하여 과제 수행을 진행한 시스템의 정보를 확인하였다.

Makefile

✓ Makefile 작성

```
CXX=g++
.SUFFIXES : .cpp .o

OBJ1=program1.o
OBJ2=program2.o
OBJ3=program3.o
SRCS=$(OBJ1: .o=.cpp) $(OBJ2: .o=.cpp) $(OBJ3: .o=.cpp)

all : program1 program2 program3
program1: $(OBJ1)
|      |      |      $(CXX) -o program1 $(OBJ1)
program2 : $(OBJ2)
|      |      |      $(CXX) -o program2 $(OBJ2)
program3 : $(OBJ3)
|      |      |      $(CXX) -pthread -o program3 $(OBJ3)

clean :
rm -f program1 program2 program3
rm -f *.o
```

```
CXX=g++
.SUFFIXES : .cpp

all : program1 program2 program3
program1:
|      $(CXX) -std=c++11 -o program1 program1.cpp
program2 :
|      $(CXX) -std=c++11 -o program2 program2.cpp
program3 :
|      $(CXX) -std=c++11 -pthread -o program3 program3.cpp

clean :
rm -f program1 program2 program3
```

처음에는 makefile의 작동법을 잘 이해하지 못해 왼쪽과 같이 매크로를 활용하여 작성하였다. CXX, OBJ1, OBJ2, OBJ3, SRCS 등은 makefile 안에서 변수로 취급되고 다뤄진다. CXX는 앞으로 사용할 컴파일러가 무엇인지 변수에 저장해준 것이고, OBJ는 program1, program2, program3 실행파일을 생성하기 위한 중간 산물이다. 이 중간 산물들이 모여 최종 실행파일을 만드는 것인데, 여러가지 헤더파일, 코드 등이 하나의 실행파일을 생성하기 위해 필요할 때 object를 활용한다. 그 것을 이해하고 나서, 굳이 이 코드에서는 중간산물을 활용할 필요가 없다는 생각이 들어 오른쪽과 같이 조금 더 간단한 모습으로 makefile 내용을 변경하였다. 또한 처음에 C++11 환경을 명시해주지 않았기 때문에 추가해 주었다. **\$make** 커맨드를 입력하면, 아래 그림과 같이 program1.cpp, program2.cpp, program3.cpp이 차례대로 컴파일 되어 실행파일을 생성한다. 그 후에 **\$make clean** 명령어 입력 시 생성됐던 실행파일이 삭제될 수 있도록 내용을 작성했다. program3 같은 경우는 pthread API를 사용하기 때문에 -pthread 옵션을 추가하여 컴파일 해준다.

```
mingbee@mingbee-VirtualBox:~/Desktop/OS/assignment2$ make
g++ -c -o program1.o program1.cpp
g++ -o program1 program1.o
g++ -c -o program2.o program2.cpp
g++ -o program2 program2.o
g++ -c -o program3.o program3.cpp
g++ -pthread -o program3 program3.o
```

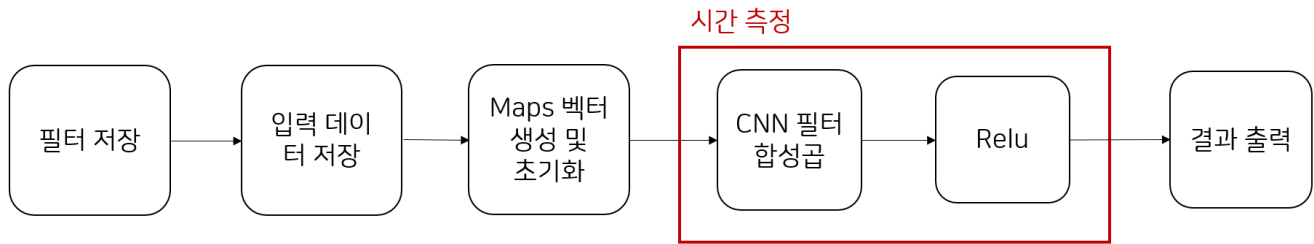
첫 번째 Makefile 을 사용하여 **\$make** 명령을 수행할 시, **programNum.o** 라는 중간산물이 생성되고 그 이후에 해당 오브젝트 파일을 활용하여 program1 2 3 실행파일이 생성되는 것을 확인할 수 있다.

```
mingbee@mingbee-VirtualBox:~/Desktop/OS/assignment2$ make
g++ -std=c++11 -o program1 program1.cpp
g++ -std=c++11 -o program2 program2.cpp
g++ -std=c++11 -pthread -o program3 program3.cpp
```

최종 Makefile에서는 그러한 번거로운 과정 없이 한번에 program 1 2 3 실행파일이 생성될 수 있도록 수정했다.

📱 프로그램 동작과 구현

✓ program1



프로그램1에서는 과제 설명에서 제시한 대로 데이터를 입력 받아, CNN 합성곱 연산과 활성화 함수를 적용시켜 출력하는 순서대로 코드를 짰다. 먼저 필터를 입력 받기 전, 필터에 대한 정보를 담고 있는 변수들을 선언하여 값을 받는다. **fil_num, fil_row, fil_col** 은 각각 입력 순서대로 필터의 개수, 필터 행의 사이즈, 필터 열의 사이즈를 담고 있다. 변수의 값을 저장하고 나면 필터의 값을 저장할 다차원 벡터 **filter**와 각 채널, 행, 열의 정보를 담은 임시 벡터도 생성해준다. 각 필터가 3채널이기 때문에, 다음 **fil_num * 3 * fil_row** 의 행 개수만큼 읽어서 **3 * fil_row** 줄 만큼 읽을 때마다 필터 벡터에 추가해준다. 그 후에는 같은 방식으로 인풋 데이터의 정보와 인풋 데이터를 입력 받는다. **irow, icol** 이라는 변수에 처음 행과 열의 개수를 저장하고, **input** 벡터를 생성하였는데, 데이터 상하좌우로 생긴 패딩을 적용하기 위해 **input** 벡터는 처음부터 **irow+2 * icol+2** 사이즈로 생성하였다. 그 후 각 모서리 부분을 제외한 가운데 부분에 순서대로 입력데이터를 입력받아 저장하였다.

데이터 처리가 끝난 뒤에는 결과 연산을 저장할 **maps** 벡터를 생성하여 초기화 해주었다. 이는 전체 인풋 데이터에서 필터를 옮겨가며 측정하는 것이기 때문에 **maprow**와 **maprcol** 에 각각 **input** 벡터의 행, 열의 개수에서 필터의 행, 열 개수를 뺀 다음 +1 해준 값을 저장해주었다.

```
int maprow=irow+2-fil_row+1;
int mapcol=icol+2-fil_col+1;
```

maps 벡터는 필터의 개수만큼 생성되며, CNN 연산 결과 3채널로 처음에 계산이 되고 3채널을 하나의 채널로 합하는 과정을 거쳐야 하지만, 3채널로 나누어 다시 하나의 채널로 변경하는 것이 번거롭게 느껴졌기 때문에 처음부터 각 필터별로 하나의 채널로 만든 다음, 채널 별 연산 결과를 해당 원소에 합산해주는 식으로 CNN 연산 알고리즘을 짰다. 따라서 **maps** 벡터는 **filter** 벡터와 다르게 3차원으로 구성되었으며, **filter** 벡터는 채널 차원이 하나 더 추가된 4차원 벡터이다. CNN 연산은 현재 필터 넘버를 **i**, 현재 행을 **k**, 현재 col을 **l** 이라고 했을 때 **maps[i][k][l]**에 다음 계산 값을 저장했다.

```
maps[i][k][l]+=filter[i][j][x][y]*input[j][x+k][y+l];
```

여기서 **x**와 **y**는 **filter** 의 행과 열이다. 매 연산의 반복마다 **input** 의 시작점이 되는 **k**와 **l**에서 **filter** 의 위치 만큼을 더해 주어서 곱셈을 해준 뒤, **maps**의 해당 시작점 좌표의 값에 더하기를 해주는 방식이다.

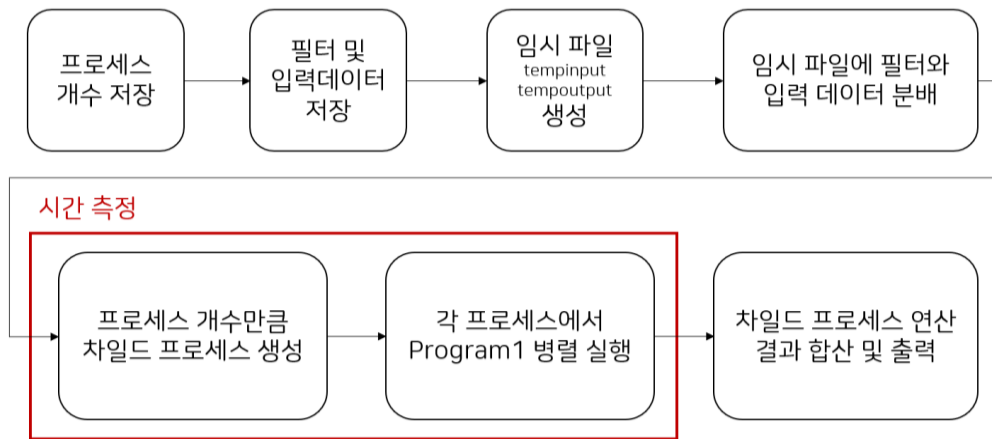
각 원소별로 더하여 **maps** 벡터에 연산을 끝낸 뒤, **ReLU** 활성화 함수를 적용시켜 준다. **ReLU**함수는 간단하게 **maps** 의 각 원소를 읽어 0보다 작은 경우 0으로 바꾸어 주는 형식을 취했다.

CNN 합성곱 연산과, **ReLU** 활성화 함수를 취해주는 부분이 실질적인 '연산' 과정이고 데이터를 사용하여 결과를 만들어내는 부분이기 때문에 이 두 단계를 끝내는데 시간이 얼마나 걸리는 지를 확인하여 프로그램 수행시간으로 측정했다.

시간 측정에는 **chrono** 라이브러리를 사용하였다. 그 후 **maps**의 행 별 필터 별 결과와 수행시간을 출력하였다.

```
chrono::system_clock::time_point Start=chrono::system_clock::now();
/*CNN ReLu */
chrono::system_clock::time_point End=chrono::system_clock::now();
```

✓ program2



프로그램2는 main() 함수의 argv를 활용하여 인풋을 처리했다. 먼저 argv[1] 을 읽어 **oriProNum** 이라는 변수에 저장했다. 그 후 프로그램1과 마찬가지로 필터 정보를 각 변수에 저장해주었다. 그 후, fil_num 과 oriProNum 을 비교하여서, 만약 oriProNum 이 fil_num 보다 큰 경우에는 **pronum**을 fil_num으로, 아닌 경우에는 oriProNum으로 설정해주었다. 프로세스 개수가 필터의 개수보다 큰 경우에는 분배할 데이터가 없기 때문에, 그 두 값의 차이 만큼 없는 것 처럼 취급하는 과정이다. 데이터 처리와 연산 과정에서는 pronum이라는 변수를 실제 프로세스 개수인 것처럼 진행을 하고, 마지막 프로세스별 시간 출력을 할 경우에만 차이만큼의 프로세스 개수만큼 0을 출력하는 식으로 구성했다.

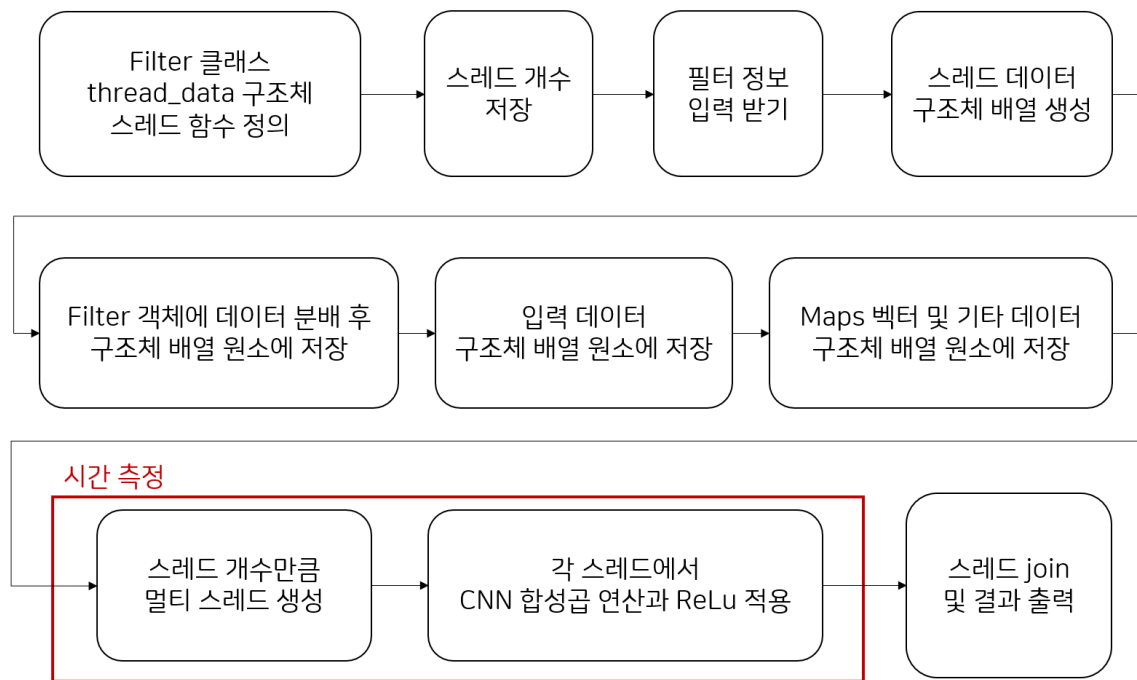
그 후 filter 배열에 모든 필터 정보를 일단 먼저 저장하였다. 프로그램1, 3과 달리 프로그램2에서는 배열을 사용하여 입력 데이터를 처리하는데, 그 이유에 대해서는 수행 과정 중 애로사항에서 자세히 다룰 것이다. 그 후, 마찬가지로 input 다차원 배열에 입력 데이터를 저장해주었다. 사전에 모든 필터 데이터와 입력 데이터를 저장해두는 것은, 입력 데이터는 모든 프로세스에서 공용으로 사용되는 정보이기 때문에 임시파일을 생성한 뒤 한 번에 분배 된 필터 데이터와 입력 데이터를 입력해주기 위해서다. 여기서 input에 패딩 작업을 하지 않는 이유는 이 데이터가 연산을 위한 데이터가 아니라 프로그램1 수행을 위한 표준 입력 데이터가 될 것 이기 때문에 원본 입력 데이터의 형태를 유지하기 위함이다.

임시파일을 생성하기에 앞서, 앞으로 파일을 읽고 쓰는 과정에서 파일 이름이 많이 사용될 것이기 때문에 미리 파일 이름을 저장하는 배열 **inS** 와 **outS** 을 각각 pronum 만큼의 원소를 가지도록 생성하였다. 그리고 필터를 적절히 프로세스 개수만큼 나누어 주어야 하기 때문에 fil_num 을 pronum 으로 나눈 몫과 나머지를 각각 q , rem 변수에 저장했다. 일단 각 프로세스는 몫만큼의 필터를 나누어 가지는데, 만약 필터 개수가 프로세스 개수로 나누어 떨어지지 않는다면 첫 프로세스부터 순차적으로 하나씩 더 맡아서 처리해야할 것이다. 그렇기 때문에 rem이 0이 될 때까지의 프로세스는 q+1 만큼의 필터를 분배 받고, rem 이 0이 되고 난 이후의 프로세스들은 원래 분배 개수인 몫 q 만큼을 나누어 가진다. 그리고 지금 파일에 입력을 해야 하는 필터가 몇 번째 필터인지도 확인해야 했기 때문에 그 위치를 저장하는 idx 변수를 선언하여 매 반복마다 위치를 옮겨주었다. 따라서 **tempinput (i)** 라는 임시 인풋 파일과 **tempoutput (i)** 라는 임시 아웃풋 파일을 만들어서, 각 이름을 inS, outS에 저장해준 뒤 임시 인풋 파일에 분배된 필터의 정보와 입력데이터를 입력했다.

그 후 프로세스 개수만큼 pid를 저장하는 배열 **pids**를 만든다. 프로그램 시간 측정을 시작하고, fork() 콜을 사용하여 프로세스들을 생성한다. 그 후에 dup2 시스템 콜을 활용하여서, 각 프로세스 별로 생성되어 있는 임시 파일을 redirection 하여 프로그램1을 각각 실행시킨다. 임시파일을 프로그램1의 표준 입력으로 대신하고, 프로그램1이 수행되고 난 뒤의 결과 표준 출력을 임시 아웃풋 파일에 저장한다. 각 프로세스의 작업이 끝나면 시간 측정을 끝내준다.

결과를 출력하기 전에, 각 프로세스에서 걸린 시간을 저장할 배열 times를 생성해주는데 이때 사이즈는 pronum 이 아니라 원래 인풋으로 들어온 **oriProNum**으로 해야한다. 그 뒤, 첫 번째 임시 아웃풋 파일부터 차례대로 읽으면서 출력해준다. **peek()** 을 사용하여서 만약 다음 라인이 EOF 즉 파일의 끝이면, 그 프로세스의 출력을 중지하고 다음 라인의 값을 times에 저장해준다. 결과 출력이 끝나면 각 프로세스 별로 걸린 시간을 쫓 출력한 뒤, 다음 라인에 프로그램2 수행 시간을 출력한다. 모든 과정이 끝나면 생성되었던 tempinput, tempoutput 파일을 **remove()** 를 통해 삭제시켜준다.

✓ program3



프로그램3은 데이터를 입력받고 처리하는 과정을 하기 전, 이 프로그램에 필요한 구조체와 클래스를 정의 주었다. 이는 프로그램2와 달리 파일의 형식으로 각 스레드에 데이터를 전달하지 않기 때문에, 연산에 필요한 정보를 모두 담은 구조체 필터와 그를 포함한 구조체 인스턴스를 인자로 전달해주기 위함이다. 먼저 **Filter** 클래스는 필터의 개수, 행 열의 개수와 필터 데이터를 담고 있는 filter 벡터로 구성되어 있다. 생성자에 값을 전달하여 값들을 지정해줄 수 있다. 또한 **set_Filter** 메소드는 인자로 들어온 첫 네 가지 숫자를 각각 필터의 개수, 채널, 행, 열의 위치로 하여서 해당 위치의 원소 값을 마지막 인자인 val 의 값으로 지정하는 작업을 수행한다.

```
void set_Filter(int a, int b, int c, int d, int val) {
    filter[a][b][c][d]=val;
}
```

thread_data 구조체는 합성곱 연산에 필요한 정보와, 그 연산의 결과를 저장하기 위한 벡터로 구성되어 있다. 먼저 입력 데이터와, 결과 데이터의 행과 열 정보 및 데이터를 가지고 있다. 또한 프로그램 수행시간을 저장할 수 있는 time 변수와, 각 스레드에 분배된 필터의 정보를 모두 담고 있을 Filter 클래스 객체를 가지고 있다. 이 프로그램에서는 map과 output이 동일한 의미로 사용되고 있다.

다음은 스레드가 생성 될 때 수행할 함수에 대한 정의이다. 이 함수에서는 프로그램1에서 진행했던 CNN 연산과 ReLu 연산을 그대로 적용한다. 다만, 인자로 받은 구조체 파일의 정보를 지역 변수에 저장해주는 과정이 필요하다.

```
struct thread_data *temp=(struct thread_data*)data;
Filter tempfil=temp->tempfil;
/*그외에도 다양한 변수들을 이렇게 선언해준다 */
```

이런 식으로 정보를 저장할 temp라는 새로운 구조체에 인자로 받은 구조체를 할당해주고, temp의 값들을 지역 변수 및 객체에 저장해준다. 이런 처리 과정을 거친 뒤 실제 연산이 시작되기 때문에 그 때부터 프로그램 시간을 측정하기 시작한다. 이 때 output 벡터에 연산 과정을 저장하게 되는데, 연산이 끝나고 프로그램 시간 측정도 끝난 후에 이 output 벡터를 다시 temp 구조체의 output에 할당해주는 작업이 필요하다. 처음 연산에 사용 된 output 벡터는 temp로부터 빈 벡터를 할당받아 사용한 것이기 때문에, 연산이 끝난 벡터의 값을 다시 temp 에 저장해주는 과정이다. 마찬가지로 수행 시간 또한 temp 구조체의 time 변수에 저장해주고, 스레드 작업이 끝남과 동시에 temp 구조체를 리턴해주면 된다.

```
pthread_exit(temp);
```

main()함수에서는 프로그램2화 마찬가지로 argv를 활용하여 스레드 개수 **oriThNum** 을 읽어 오고, 표준 입력으로 **filnum, filrow, filcol** 에 각각 필터의 개수, 행, 열의 개수를 저장한다. 여기서도 스레드 개수가 필터의 개수보다 클 경우를 대비하여서 **thnum**으로 연산에서 사용될 스레드의 값을 조정해준다.

thread_data 구조체의 배열인 splits 를 생성해주고, 프로그램2에서 한 것처럼 필터를 분배하기 위해 필터 개수에서 프로세스 개수를 나눈 몫과 나머지를 q, rem 에 저장해준다. 프로그램2와 다른 점은, 필터 정보를 벡터에 먼저 다 저장해 뒀다가 분배하는 방식이 아니라, 입력을 받을 때부터 분배하여 저장한다는 것이다. 따라서 필터 클래스 인스턴스를 생성하고, 그 클래스에 순서에 따라 q+1 혹은 q 개의 필터를 분배하여 저장한다. 필터의 값을 지정할 때는 Filter 클래스의 set_filter() 메소드를 활용한다. 매 반복마다 이렇게 분배하여 입력 완료 된 필터 클래스 객체를 splits 배열에 저장해준다.

필터 분배 작업이 끝나면 입력데이터를 입력받는다. 이 때는 프로그램2와 다르게, 프로그램1처럼 패딩 작업을 해주어야한다. 따라서 input 벡터를 행과 열의 사이즈가 원본 입력 데이터보다 각 2만큼씩 크게 설정하여 생성해주고, 테두리를 뺀 가운데 부분에 표준입력으로 들어오는 값들을 저장해준다. 그 후 연산 결과를 저장할 maps 벡터도 생성하고 초기화해준다. 이 작업이 끝나면 splits 원소 구조체에 공통 된 값들을 넣어주면 된다. 아웃풋과 인풋 벡터 및 그 정보, 그리고 수행측정 시간을 0으로 초기화하여 저장한다.

지금부터 프로그램3의 수행시간 측정을 시작한다. 스레드 풀을 생성하고, pthread_create()을 통해 스레드를 만들어서 t_function으로 splits의 원소를 인자로 전달해준다. 함수가 불리면 각 스레드에서 할당받은 데이터로 연산 작업을 실시하고 종료 된 결과를 pthread_join 을 통해 result로 전달받는다.

```
pthread_join(p_thread[i], &result);  
struct thread_data* tttt=(struct thread_data*)result;
```

각 스레드의 결과를 저장하기 위한 구조체의 배열인 results를 하나 더 생성하고, 그 안에 순서대로 전달받은 구조체 정보를 저장한다. 이 작업이 끝나면 수행시간 측정을 종료한다.

프로그램2 처럼 각 스레드 별 수행시간을 저장할 times 배열을 생성하고 0으로 초기화해준다. 그리고 results 구조체 배열을 순서대로 돌면서 각 구조체의 output 벡터를 통해 결과를 출력한다.

```
cout << results[i]->output[x][y][j]<<" ";
```

결과 출력이 끝나면, times 배열의 원소도 하나하나씩 출력해주고 프로그램3 자체 수행시간 또한 다음 줄에 출력하여 마무리 한다.

윈도우와 리눅스의 에디터가 달라, 보고서에 등장하는 캡처본이 상이할 수 있습니다. 양해 부탁드립니다!

결과 분석

✓ 주어진 input.txt 에 대한 결과

[프로그램 1 의 결과]

```
mingbee@mingbee-VirtualBox:~/Desktop/OS/assignment2$ ./program1 <input.txt
4998 12187 8544 13742 7343 1410 20095 16173 20362 21800 17465 19193 11664 11859 6539 3928 1734 19064 153

0 1793 0 0 4615 1137 0 0 0 3258 0 0 0 0 0 0 0 4254 0

1
```

[프로그램 2 의 결과]

```
mingbee@mingbee-VirtualBox:~/Desktop/OS/assignment2$ ./program2 2 <input.txt
4998 12187 8544 13742 7343 1410 20095 16173 20362 21800 17465 19193 11664 11859 6539 3928 1734 19064 153
32

0 1793 0 0 4615 1137 0 0 0 3258 0 0 0 0 0 0 0 4254 0

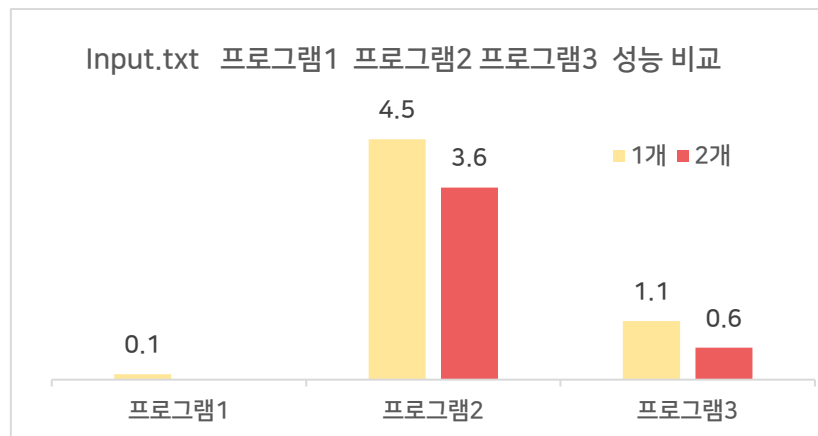
0 0
4
```

[프로그램 3 의 결과]

```
mingbee@mingbee-VirtualBox:~/Desktop/OS/assignment2$ ./program3 2 <input.txt
4998 12187 8544 13742 7343 1410 20095 16173 20362 21800 17465 19193 11664 11859 6539 3928 1734 19064 153
32

0 1793 0 0 4615 1137 0 0 0 3258 0 0 0 0 0 0 0 4254 0

0 0
1
```

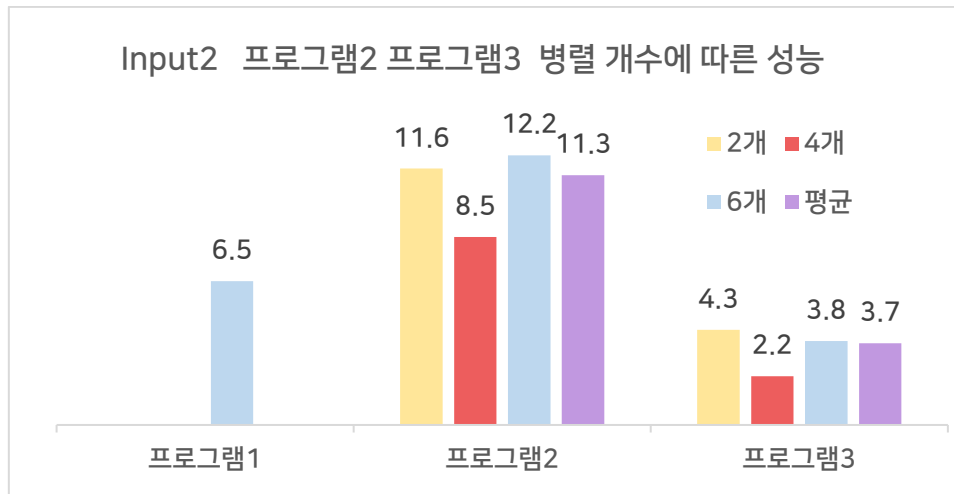


각가 프로그램마다 15 번 정도의 결과 값을 비교하여 평균치를 낸 그래프이다. 노란색 막대는 프로그 2(이하 p2), 프로그 3(이하 p3)에서 1 개의 프로세스/스레드로 진행하였을 때의 결과이고, 빨간색은 2 개로 진행한 결과이다. 프로그램 1(이하 p1)의 결과는 노란색 막대로 나타났다. p2 와 p3 모두 병렬화가 2 개로 되었을 때가 하나의 프로세스/스레드로 처리했을 때보다 확연히 빨랐다. 하지만 p1 과 비교해서 비교적 많이 큰 값들이 나왔다. 이는 주어진 인풋의 사이즈가 너무 작아서, 병렬화를 하는 의미가 크게 없기 때문이라고 볼 수 있다. 인풋 사이즈가 작을 때는 굳이 병렬화를 하여 처리하는 것 보다, 그냥 하나의 프로세스로 한꺼번에 처리하는 것이 더 효율적이라는 것을 알 수 있다.

● 알게 된점

- 2 개로 병렬화 한 경우가 한 개로 진행했을 때 보다 더 효율 적이다.
- 인풋 사이즈가 너무 작으면 병렬화를 하는 의미가 없다.
- 그럼에도 멀티 스레드가 멀티 프로세스보다 빨랐다.

✓ input2 에 대한 결과

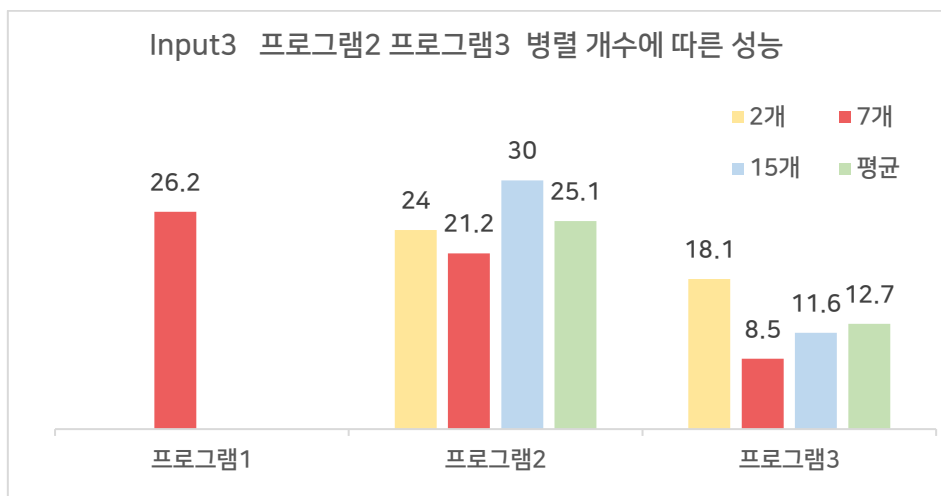


따라서 조금 더 인풋 사이즈를 키워서 테스트 해보았다. 필터의 개수는 8 개로 늘리고 필터의 사이즈는 동일하게, 입력 데이터 15*20 으로 증가시켰다. 이 경우에서도 아직 p1 이 p2 보다는 빠른 것으로 나타났다. 각 평균치와 비교해보면 p2는 p1 보다 아직 오래걸리는 반면 멀티 스레딩의 경우는 단일 프로세스보다 작업 속도가 빨라진 것을 확인 할 수 있었다. 아마 아직 멀티 프로세스의 장점이 나타나기에 조금 작은 인풋 사이즈인 것 같다고 판단했다. 그리고 또 하나 흥미로운 점은, 무조건 프로세스/스레드 수가 많다고 속도가 빨라지는 것이 아니라는 점이다. p2 와 p3 모두 병렬 개수가 4 개일때가 6 개 일 때보다 더 빠른 결과를 보였다. 아마 현재 필터의 사이즈가 8 개이기 때문에, 6 개로 진행을 하여도 2 개씩 맡은 필터가 작업이 끝날 때까지 기다려야하는 것은 똑같으며, 연산의 규모가 크지 않아서 병렬화보다 한 꺼 번에 처리하는 게 아직은 효율적인 것 같다고 생각을 했다. 따라서 8 개의 반인 4 개로 돌렸을 때가 가장 빠르게 나타났던 것 같다고 분석했다.

● 알게된 점

- 이 정도 크기의 인풋은 멀티 프로세스보다 멀티 스레딩으로 처리하는 게 효과적이다.
- 병렬 개수가 무조건적으로 많다고 프로그램이 효과적으로 수행되는 것이 아니며, 병렬화와 일괄적 처리 중 어떤 것이 효율적일 지는 인풋 사이즈에 따라 달라진다.

✓ input3 에 대한 결과



인풋 사이즈를 필터 크기 15, 행 3, 열 4, 입력 데이터 15*40 으로 늘린 뒤 한 번 더 결과를 파악해보았다. 이 정도 인풋을 다루게 되니, p2 와 p3 모두 p1 보다 훨씬 효율적으로 일을 처리하고 있음을 확인할 수 있었다. 그리고 또한 필터 15 개를 15 개로 병렬화 하는 것이 가장 효과적일 것이라고 예상했는데 그 성능이 7 개인 것보다 떨어져서 의외였다.

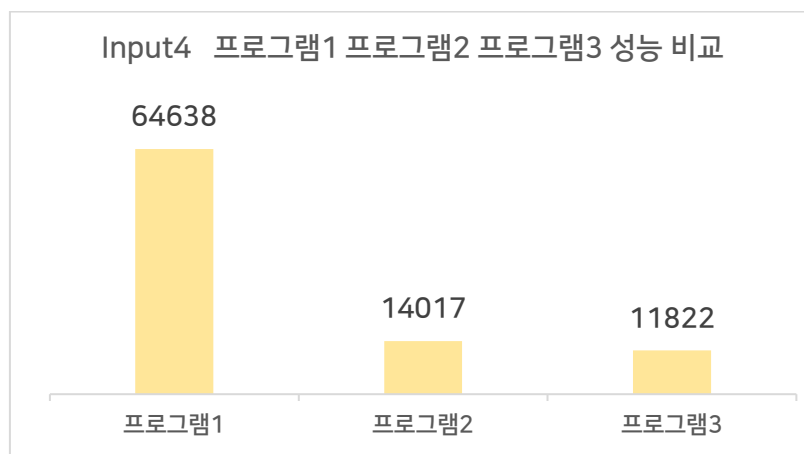
만약 한가지 필터를 각각 처리해서 합하는 것이 훨씬 더 빠를 것이라고 생각했는데, 결과는 그렇지 않았다. 그 이유를 예측해보니, 병렬성과 일괄성 그 외에 더 중요한 요인이 있음을 깨달았다. 바로 **CPU 코어의 개수**이다.

멀티 프로세싱은 멀티 코어의 하드웨어적 이점을 사용하여 각 CPU 마다 하나의 프로세스를 돌리는 것이다. 그렇기 때문에, 만약 6 개의 CPU 가 있고 프로세스가 6 개 이하라면 각 CPU 를 모두 사용하여 멀티 프로세싱의 이점을 최대로 사용할 수 있지만, 6 개가 넘어가는 프로세스를 돌린다면 CPU 의 코어보다 많은 수의 프로세스들은 다른 프로세스와 하나의 CPU 를 나누어 써야하기 때문에 성능이 더 떨어지거나 비슷할 수 있는 것이다. 내가 과제를 진행했던 시스템의 CPU 코어 수는 6 개로, 6 개와 근사치인 7 개의 프로세스를 할당하여 연산을 처리한 것이 가장 효과적인 결과를 내고, 그 이상으로 가면 오히려 성능이 나빠지거나 유의미한 차이가 없어질 수 있다는 점을 알게 되었다. 멀티 스레딩의 경우에도 결국 하나의 프로세스 내에서 다중 스레드들이 쪼개어져서 작업을 병렬화하는 것이기 때문에, 프로세스 자체가 CPU 수 보다 많게 되면 성능 저하가 오거나 큰 변화가 없는 것은 마찬가지로 일 것이라고 판단했다.

● 알게 된 점

- CPU 코어의 수보다 프로세스/스레드의 개수가 많을 경우 오히려 성능이 저하되거나 효과가 없을 수 있다.
- 인풋 사이즈가 커질수록 단일 프로세스 모델보다 멀티 프로세스/스레드 모델이 효과가 더 좋아졌다.

✓ input4 에 대한 결과



	Program2		Program3	
병렬 수	7개	30개	7개	30개
걸린 시간	12178	14017	11708	11822

인풋 사이즈를 엄청나게 크게 만들어서 마지막으로 성능을 측정해보았다. 필터의 개수는 60 개, 7*8 모양으로 만들었고 입력 데이터는 300*300 사이즈로 만들었다. 이렇게 큰 크기의 인풋으로 성능 비교를 해보니 p1 과 p2, p3 간의 성능 차이가 확연하게 나타났다. 그래프는 병렬화를 각 60 개의 반인 30 개로 돌렸을 때의 값을 p1 과 비교한 것이다. 멀티 스레딩과 싱글 프로세스는 약 6 배의 성능차이가 나고 있음을 확인할 수 있었다. p2 와 p3 는 성능차이가 크게 나지는 않았지만 그래도 여전히 멀티 스레딩이 멀티 프로세스보다 훨씬 빨랐다. 아마 CPU 코어의 개수가 더 높아진다면 더욱 더 큰 차이가 날 것으로 예상된다.

옆의 도표는 CPU 개수와 비슷한 7 개와 30 개로 병렬화를 했을 때 성능 차이를 비교한 것이다. p2 와 p3 모두 7 개와 그 4 배에 가까운 30 개로 병렬화를 하였을 때 큰 차이가 없는 것으로 볼 수 있다. 다만, p2 는 멀티 프로세스 모델이라서 그런지 오히려 7 개로 돌렸을 때의 평균값이 훨씬 작게 나오는 걸 알 수 있었다. 하지만 멀티 스레딩은 7 개, 30 개, 위 도식에는 없지만 그 외의 어떤 수로 병렬화를 하여 돌리더라도 모두 10000~12000 정도의 값 사이로 나와 큰 성능의 변화가 없는 것으로 확인 되었다.

✓ 실험 결과 정리

실험으로 얻은 결과를 정리하면 다음과 같다.

1. 싱글 프로세스 모델보다 멀티 프로세스, 멀티 스레딩이 확연하게 성능이 좋다.
2. 그 성능차이는 인풋의 사이즈가 커질수록 더욱 더 크게 드러난다.
3. 따라서 인풋 사이즈가 너무 작을 때에는 오히려 싱글 프로세스가 더 효율적일 수 있다.
4. 또한 멀티 프로세스보다 멀티 스레딩이 모든 상황에서 더 성능이 좋다
5. 인풋 사이즈에 따라서 병렬화를 하는 것과 일괄적 처리 중에 더 효과적인 기법이 다를 수 있다.
6. 하지만 성능에 가장 큰 영향을 미치는 것은 CPU 코어의 개수이다.
7. CPU 코어의 개수 이상의 병렬화를 하게 되면, 그 성능은 저하되거나 크게 차이가 없을 수 있다.

✓ 수행 확인

```
mingbee@mingbee-VirtualBox:~$ ps -eaT | grep program1
17396 17396 pts/0    00:00:02 program1
17397 17397 pts/0    00:00:02 program1
17398 17398 pts/0    00:00:02 program1
17399 17399 pts/0    00:00:02 program1
17400 17400 pts/0    00:00:02 program1
17401 17401 pts/0    00:00:02 program1
17402 17402 pts/0    00:00:02 program1
```

```
mingbee@mingbee-VirtualBox:~$ ps -eaT | grep program3
17377 17377 pts/0    00:00:00 program3
17377 17378 pts/0    00:00:04 program3
17377 17379 pts/0    00:00:04 program3
17377 17380 pts/0    00:00:03 program3
17377 17381 pts/0    00:00:04 program3
17377 17382 pts/0    00:00:04 program3
17377 17383 pts/0    00:00:04 program3
17377 17384 pts/0    00:00:04 program3
```

다음과 같이 프로그램 2 와 프로그램 3 이 잘 작동하고 있는 것을 확인 할 수 있다.

✓ makeinput

테스트에 사용 된 인풋 파일은 makeinput.cpp 파일을 생성하여 랜덤으로 범위에 맞는 인풋을 생성하여 진행했다.

```
 srand(time(NULL));
ofstream input("input");
input<<filn << " "<<filr<< " "<<filc<<"\n";
for (int i=0; i<filn; i++) {
    for (int j=0; j<3; j++) {
        for (int k=0; k<filr; k++) {
            for (int l=0; l<filc; l++) {
                int ran=rand()%62)-31;
                input<<ran<< " ";
            }
            input<<"\\n";
        }
    }
}
input<<inr<< " "<<inc<<"\\n";
for (int i=0; i<3; i++) {
    for (int j=0; j<inr; j++) {
        for (int k=0; k<inc; k++) {
            int ran=rand()%256;
            input<<ran<< " ";
        }
        input<<"\\n";
    }
}
```

📱 과제 수행 중 애로 사항

✓ ISO warning

```
mingbee@mingbee-VirtualBox:~/Desktop/OS$ g++ program2.cpp
program2.cpp: In function 'int main()':
program2.cpp:16:43: warning: ISO C++ forbids converting a string constant to 'char*' [-Wwrite-strings]
    char *args[] = {"/program1", NULL};
                                ^
```

프로그램2를 컴파일 하는 과정에서 다음과 같은 경고를 확인할 수 있었다. 이 경고는 string constant를 char* 로 변경하는 것을 권장하지 않는다는 경고문이다. 이 경고는 해당 string을 type casting 해줌으로써 손쉽게 해결할 수 있었다.

```
char *args[] = {(char*)"/program1", NULL};
```

이렇게 오류가 났던 부분을 type casting 해주고 나면, 컴파일 과정에서 오류가 발생하지 않는 것을 확인하였다.

✓ vector 와 array

프로그램 동작 과정을 살펴보면, 데이터를 처리할 때 프로그램1과 프로그램3은 벡터를 사용하고 있고, 프로그램2는 배열을 사용하고 있음을 알 수 있다. 과제를 진행하면서 프로그램1부터 순차적으로 코딩을 하기 시작했는데, 벡터보다 배열이 크기가 정해져 있을 때는 더욱 빠르기 때문에 배열을 사용했다. 하지만 프로그램3을 짜는 과정에서 구조체에 정보를 담을 때나 Filter 클래스에서 정보를 처리할 때 등 배열을 사용할 수 없었기 때문에 벡터를 사용해야 했고, 그렇게 하게 되니 프로그램1 2와 수행시간 상에서 큰 차이가 나게 되었다. 프로그램3은 벡터에서 연산을 하고 프로그램1과 2는 배열을 통해 계산을 하니 프로그램1이 프로그램3에 비해 약 3배 정도 적은 시간이 걸렸다. 따라서 프로그램3에 맞추어 프로그램1 또한 배열 대신 벡터를 사용하게끔 바꾸었고, 조금 더 처리를 빨리 하기 위해 reserve() 를 사용했다.

다만, 프로그램2에서 데이터를 처리하는 부분에서 배열을 쓴 이유는, 그 배열을 가지고 합성곱 연산을 하는 것이 아니라 그것을 이용해 임시파일에 내용을 입력하는 것이기 때문이었다. 임시 파일에 저장된 내용이 표준 입력으로 대체 되어 프로그램1에 전달되기 때문에 프로그램2에서는 코드를 수정하지 않고 배열을 사용했다.

✓ t_function 의 인자

대부분의 예시들은 t_function의 인자로 단순 데이터 하나만을 전달하고 있었다. 그러나 합성곱 연산을 수행하기 위해서는 다양한 변수와 벡터 들이 필요했기 때문에, 이 데이터를 전달할 수 있는 방법이 필요했다. 스펙에 명시된 공유메모리를 활용하는 방법도 있었지만, 스펙 참고문헌에서 struct를 사용하여 pthread_create을 하는 방법을 찾아냈다. 그렇게 해서 합성곱 연산에 필요한 정보들을 모두 담은 구조체를 만들었고, 그를 또 간소화하고 필터 분배를 용이하게 하기 위해서 필터 클래스도 만들었다.

✓ program3 segmentation error

이렇게 구조체를 전달하여 코드를 짰 뒤 실행을 해보았지만, 0 0 0 0 ... 으로 0이 몇 번 뜨다가 segmentation 오류가 뜨는 것을 발견했다. 인자로 전달한 구조체의 내용을 t_function에서 수정하면 그 메모리 속 내용이 바뀌는 것으로 착각했기 때문에 발생한 오류였다. 따라서 함수 내에서 구조체에 다시금 연산 결과를 저장해준 다음 pthread_exit(구조체)를 통해 main 함수로 그 값을 리턴해주는 식으로 디버깅을 했다. c++에 익숙하지 않아서 (void*) 형의 자료를 처리함에 있어서도 애를 먹었다. 하지만 구조체 포인터형으로 잘 받아주고 내용도 잘 처리해서 결과를 출력할 수 있었다.

✓ 프로그램 수행시간 측정

프로그램 수행시간을 측정할 때, 처음에는 코드 처음부터 끝까지 다 측정을 하는 식으로 했다. 그런데 생각해보니, 이렇게 하게 되면 프로그램1~3은 각각 데이터 전처리 방식이 다르고, 그 외에 작업을 수행하기 위한 추가 기능들의 구현을 위한 코드도 모두 다르기 때문에 그런 점에 있어서 시간 차이가 많이 발생할 것이라고 생각하였다. 따라서 프로그램 별로 큰 차이가 없는 합성곱 연산을 수행하는 부분과 활성화 함수를 적용하는 부분만을 측정할 수 있도록 시간을 측정했다.

또한 처음에도 시간을 측정할 때 `clock()` 함수를 썼었다. 그러나 이렇게 하게 되니 의도한 대로 프로세스 생성, 스레드 생성 및 연산 수행의 시간이 제대로 측정되는 것 같지 않고, 인풋을 키워도 프로그램1의 수행시간이 가장 작게 나오는 오류가 있어서 `chrono` 라이브러리를 사용하는 것으로 바꾸었다.

✓ 컴파일러 버전 다운그레이드

코드를 작성하고 테스트를 하면서, 채점 환경에 대해 전혀 고려를 하지 않고 있었다. 그러다가 해당 채점환경에서 혹시 오류가 날 상황을 대비하여 컴파일러 패키지를 다중관리하는 방법을 찾아보았다.

```
$ sudo add-apt-repository ppa:ubuntu-toolchain-r/test
```

```
$ sudo apt-get update
```

명령어를 통해 새로운 패키지를 다운받을 준비를 하고

```
$ sudo apt-get install g++-5
```

```
$ sudo apt-get install gcc-5-base
```

커맨드로 채점서버와 같은 버전의 컴파일러를 다운로드해주었다. 그 후

```
$ sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-5 60
```

```
$ sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-5 60
```

명령어로 우분투 환경에 해당 컴파일러 버전을 추가해준 뒤, 특정 버전을 선택하여 진행할 수 있도록 하였다. 7 버전의 컴파일러를 재 다운로드 받아 우선순위를 5 버전보다 높게 설정해주었기 때문에 `--version` 커맨드로 버전을 확인하면 7버전이 나오게끔 하였고, **`$ sudo update-alternatives --config gcc`** 를 입력하여 어떤 패키지가 깔려있는지, 어떤 버전을 사용할 것인지 선택할 수 있게 하였다.

참고 사이트

- <https://awesometic.tistory.com/15>
- <https://coding-start.tistory.com/198>
- <https://coding-factory.tistory.com/307>
- <https://juyoung-1008.tistory.com/47>
- https://blackinkgj.github.io/fork_and_exec/
- <https://talkingaboutme.tistory.com/entry/Study-Linux%EC%97%90%EC%84%9C%EC%9D%98-Process-%EC%99%80Thread>
- <https://m.blog.naver.com/PostView.nhn?blogId=skout123&logNo=50133652334&proxyReferer=https:%2F%2Fwww.google.com%2F>
- 강의자료
- Operating System Concepts 9th Edition
- <https://m.blog.naver.com/PostView.nhn?blogId=gq844&logNo=221365833618&proxyReferer=https:%2F%2Fwww.google.com%2F>
- <https://stackoverflow.com/questions/20944784/why-is-conversion-from-string-constant-to-char-valid-in-c-but-invalid-in-c>
- <https://computing.llnl.gov/tutorials/pthreads/>
- <https://jacking.tistory.com/988>
- <https://www.it-note.kr/19>
- <https://bitsoul.tistory.com/157>
- https://www.joinc.co.kr/w/Site/Thread/Beginning/Example_pthread
- <https://computing.llnl.gov/tutorials/pthreads/#Joining>
- <https://bitsoul.tistory.com/160>
- <https://stackoverflow.com/questions/34487312/return-argument-struct-from-a-thread>