

# Assignment 2

2015198005

허진욱

## 1. 사전조사 보고서

### 1) 프로세스와 스레드

#### i) 정의

Process: 컴퓨터에서 연속적으로 실행되고 있는 컴퓨터 프로그램

Thread: 프로세스 내에서 실행되는 여러 흐름의 단위

#### ii) 차이점

Process:

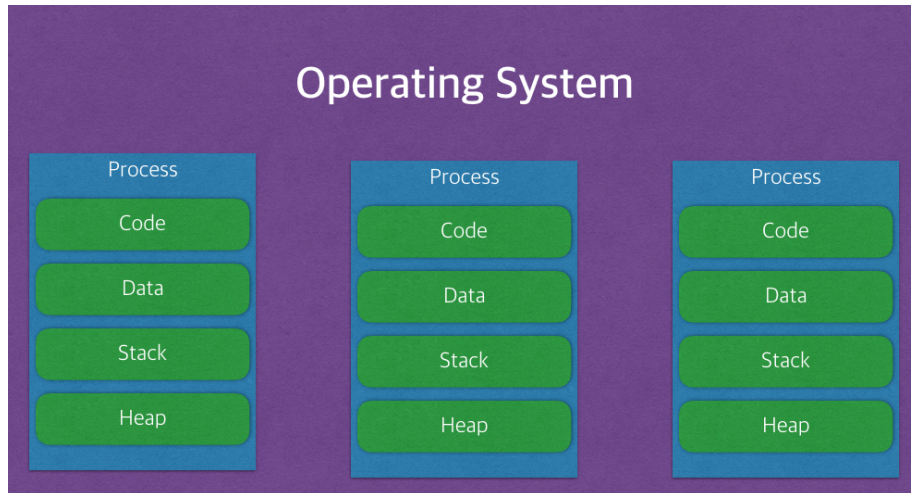
- 독립적이다
- 프로세스 각각은 독립적인 자원을 갖는다
- 프로세스는 자신만의 주소 영역을 가진다
- 프로세스는 IPC 를 이용해서만 통신이 가능하다
- swiching 이 상대적으로 느리다

Thread:

- 프로세스의 subset 이다
- 스레드는 서로 일부 자원을 공유한다
- 스레드는 주소 영역을 공유한다
- Switching 이 상대적으로 빠르다

### iii) 리눅스에서의 구조 및 구현

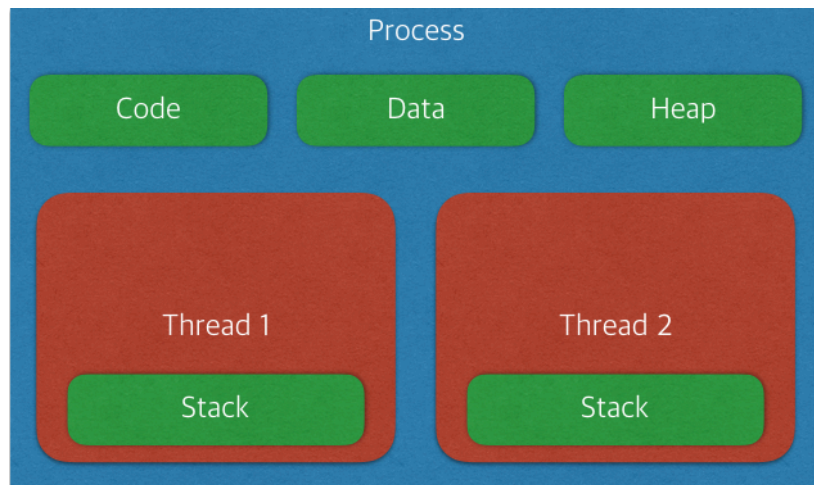
- Process(프로세스)



운영체제에서 프로그램이란 컴퓨터가 이해할 수 있는 명령어들과 명령을 수행하기 위한 데이터를 포함한 실행 가능한 객체이다. 이러한 프로그램이 실행된 객체를 프로세스라고 한다. 리눅스에서 프로세스는 image, program context, kernel context 로 이루어져 있는 자료구조이다. 이를 PCB 라고 부르는데 Image 는 코드, variable 데이터, stack, heap 으로 이루어져 있다. Program context 안에는 program counter, stack pointer, 그리고 데이터 레지스터가 있고, kernel context 에는 pid, gid, environment 등이 있다.

프로세스의 생성은 fork()라는 함수를 통해 이루어진다. 부모 프로세스가 fork() 함수를 호출하면 부모 프로세스와 동일하게 복사된 자식 프로세스가 생긴다. 이후 자식 프로세스에서 exec 계열의 함수를 통해 프로세스의 내용을 바꿀 수 있고, 이러한 방식으로 다양한 프로세스가 생성 및 구현 된다.

- Thread(스레드)



스레드는 프로세스의 생성과 구현이 효율성이 떨어진다는 문제 속에서 생겨났다. `fork()`를 이용해서 새로운 프로세스를 생성하게 되면 모든 데이터가 복사가 된다. 이러한 경우, 어떤 프로그램을 병렬로 실행하고자 할 때 효율성이 많이 떨어진다. Thread는 프로세스와 달리, 꼭 필요한 자원만 복사를 하고, 나머지 데이터는 공유를 한다. 때문에 생성과 구현이 프로세스에 비해 굉장히 효율적이다. 새로운 스레드를 생성할 때 code, data, 그리고 kernel context는 공유하고, 그 외에 부분인 stack과 program context만 새로 생성한다.

Thread를 생성하고 구현하는 함수는 운영체제마다 상이하다. 때문에 특정 운영체제에서 구현된 프로그램이 다른 운영체제에서 구현이 되지 않는 문제가 생겼는데, 이에 대한 해결책으로 pthread가 생겼다. Pthread 함수는 스레드를 생성하고 구현하는 하나의 통일된 함수로서 이 함수들을 이용하여 프로그램을 만들면, 다른 운영체제에서도 프로그램 구현이 가능하다.

## 2) 멀티프로세스와 멀티스레딩

### i) 멀티프로세스

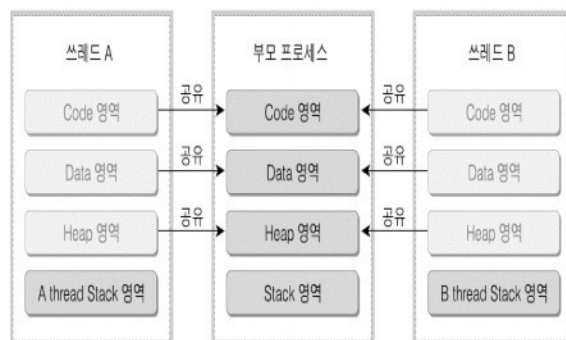
#### 1) 개념 및 구현 방법



앞서 언급했듯이 프로세스는 `fork()` 함수를 통해 자식 프로세스를 만들 수 있다. 이때 동일한 구조의 자식 프로세스를 다중으로 생성하여 하나의 프로그램을 병렬적으로 나눠서 수행할 수 있는데, 이를 멀티 프로세스라고 한다. 이러한 방식을 사용하면 단일 프로세스 보다는 효율적이지만 프로세스 간에는 메모리 공간을 공유하지 않기 때문에 IPC 통신이 반드시 필요하다. 또한 context switching 을 할 때, 작업이 무겁고 오버헤드가 발생한다.

### ii) 멀티스레딩

#### 1) 개념 및 구현 방법

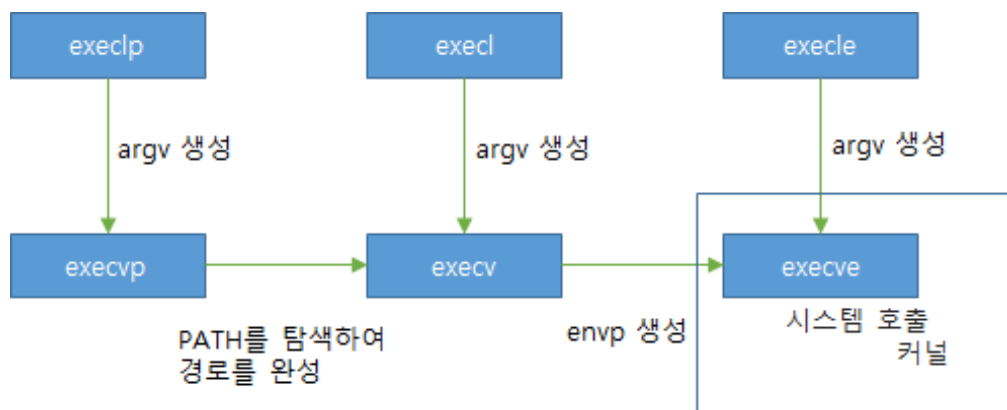


멀티 프로세스와는 달리 멀티스레딩은 메모리를 공유한다. 때문에 스레드 간에 데이터를 주고 받는게 간단해지고 시스템 자원 소모가 줄어들게 된다. Code, Data, Heap 영역을 공유하므로 Stack 영역만 처리를 하면 되고, 덕분에 오버헤드 문제를 해결 어느정도 해소할 수 있다. Pthread 함수들을 이용하여 멀티스레딩을 구현할 수 있다.

### 3) 시스템 콜

<code>#include&lt;unistd.h&gt;</code>
<code>int execl(const char *path, const char *arg0, ..., const char *argn, (char *)0);</code> path에 지정한 경로명의 파일을 실행하며 arg0~argn을 인자로 전달한다. 관례적으로 arg0에는 실행 파일명을 지정한다. execl함수의 마지막 인자로 인자의 끝을 의미하는 NULL 포인터((char*)0)를 지정해야 한다. path에 지정하는 경로명은 절대 경로나 상대 경로 모두 사용할 수 있다.
<code>int execv(const char *path, char *const argv[]);</code> path에 지정한 경로명에 있는 파일을 실행하며 argv를 인자로 전달한다. argv는 포인터 배열이다. 이 배열의 마지막에는 NULL 문자열을 저장해야 한다.
<code>int execlp(const char *path, const char *arg0, ..., const char *argn, (char *)0, char *const envp[]);</code> path에 지정한 경로명의 파일을 실행하며 arg0~argn과 envp를 인자로 전달한다. envp에는 새로운 환경 변수를 설정할 수 있다. arg0~argn을 포인터로 지정하므로, 마지막 값은 NULL 포인터로 지정해야 한다. Env는 포인터 배열이다. 이 배열의 마지막에는 NULL 문자열을 저장해야 한다.
<code>int execvp(const char *path, char *const argv[], char *const envp[]);</code> path에 지정한 경로명의 파일을 실행하며 argv, envp를 인자로 전달한다. argv와 envp는 포인터 배열이다. 이 배열의 마지막에는 NULL 문자열을 저장해야 한다.
<code>int execlp(const char *file, const char *arg0, ..., const char *argn, (char *)0);</code> 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.
<code>int execvp(const char *file, char *const argv[]);</code> file에 지정한 파일을 실행하며 argv를 인자로 전달한다. argv는 포인터 배열이다. 이 배열의 마지막에는 NULL 문자열을 저장해야 한다.

Exec 계열의 함수를 이용하여 생성된 프로세스에 새로운 프로그램을 만들 수 있다. 이 중 `execve()`만이 kernel mode 에서 수행되는 system call 이고 나머지 함수들은 wrapper function 이다. 때문에 모든 exec 계열 함수들은 내부적으로 `execve()`를 콜 한다.



#### 4) pthread 함수

- #include<pthread.h>를 추가해야 한다.
- 컴파일 옵션 -lpthread 를 반드시 추가해야 한다.

```
int pthread_create( pthread_t *th_id, const pthread_attr_t *attr, void* 함수명,  
void *arg );
```

- Thread 를 생성하는 함수
- 스레드가 성공적으로 생성되면 첫번째 인자에 tid 가 주어진다.
- 스레드에 분기할 수 있는 함수는 void\*타입이다.
- 분기되는 함수에 함께 전달할 매개변수도 void\* 이다.

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

- 특정 thread 가 종료되기까지 기다리게 만드는 함수이다.
- 각 인자는 기다려야할 thread, thread 의 리턴값에 해당한다.

```
void pthread_exit( void* ret_value );
```

- 실행중인 스레드를 종료시킬 때 사용한다.

#### 5) 참조 문헌

pthread 기본 함수: <https://bitsoul.tistory.com/156>

thread 와 multi-thread:

[https://www.joinc.co.kr/w/Site/system\\_programing/Book\\_LSP/ch07\\_Thread](https://www.joinc.co.kr/w/Site/system_programing/Book_LSP/ch07_Thread)

process 와 multi-process:

[https://www.joinc.co.kr/w/Site/system\\_programing/Book\\_LSP/ch05\\_Process](https://www.joinc.co.kr/w/Site/system_programing/Book_LSP/ch05_Process)

thread, process 개념: <https://magi82.github.io/process-thread/>

exec 계열 함수:

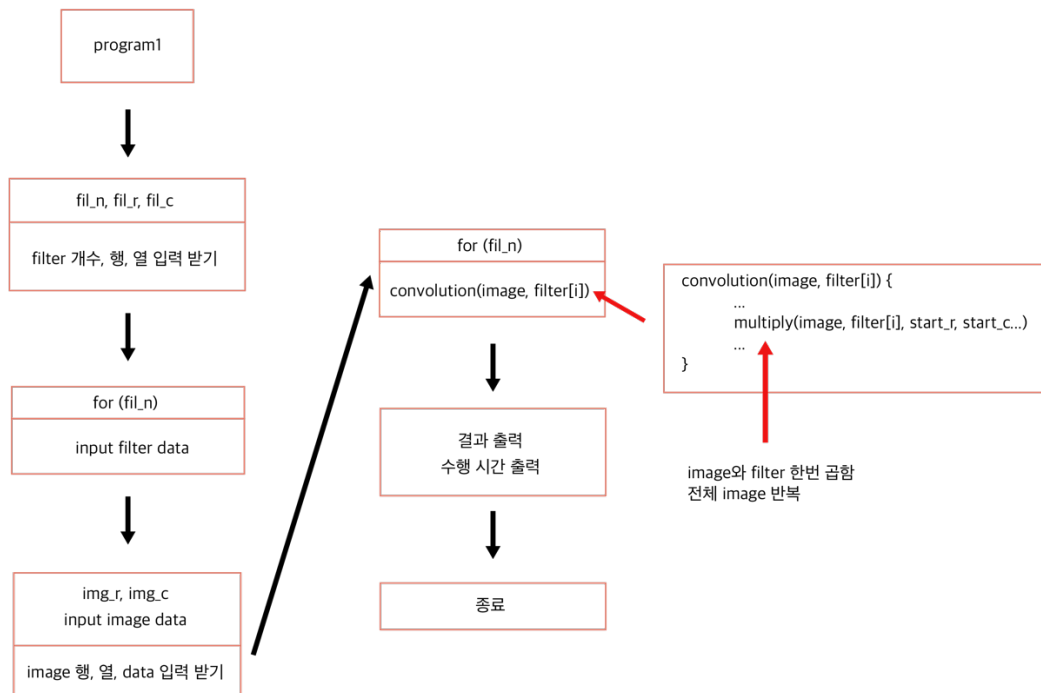
<https://bbolmin.tistory.com/35>

<http://ehpub.co.kr/tag/exec-%ED%95%A8%EC%88%98%EA%B5%B0/>

## 2. 프로그래밍 수행 결과 보고서

### 1) 작성한 프로그램의 동작 과정과 구현 방법

#### i) program1



우선 filter 와 image 의 데이터를 입력 받은 후 각 filter 마다 image 와 convolution 함수를 수행한다. 이때 convolution 함수는 convolution 과 ReLU 를 모두 구현하는 함수로서 내부에서 multiply 라는 또다른 함수를 콜 하는데, 이 함수는 특정 위치에서 image 와 filter 를 곱하여 더한 값을 리턴하는 함수이다. convolution() 함수를 통해 연산을 한 후 결과를 출력한다. 수행 시간을 출력하고 프로그램을 종료한다.

- Int multiply() 함수

```
int multiply(int** img, int** fil, int start_r, int start_c, int fil_r, int fil_c) { //multip
    int val = 0; //initialize value of multiplication
    int r = start_r; //row location of filter[0][0]
    int c = start_c; //column location of filter[0][0]
    int fil_idx_r = 0; //multiplying index of filter
    int fil_idx_c = 0;

    for(int i=0; i< fil_r; i++) {
        for(int j=0; j<fil_c; j++) {
            val += img[r][c] * fil[fil_idx_r][fil_idx_c]; //add result to val
            c++; //increase image column
            fil_idx_c++; //increase filter column
        }
        r++; //increase image row
        fil_idx_r++; //increase filter row
        c = start_c; //reset image column to start
        fil_idx_c = 0; //reset filter column to 0
    }
    return val;
}
```

이미지 채널과 필터 채널 하나씩, 곱이 이루어지는 이미지의 위치, 그리고 필터의 크기를 인자로 받아온다. 그 후 특정 시점에서의 곱셈을 하고 결과를 모두 더한 후 그 값을 리턴한다.

- Int\*\* convolution() 함수

```
int **convolution(Data img, Data fil) { //perform convolution
    int res_r = img.dat_r - (fil.dat_r - 1); //result data row
    int res_c = img.dat_c - (fil.dat_c - 1); //result data column
    Data result = Data(res_r, res_c); //result data of convolution

    int*** results = new int**[3] {result.chan1, result.chan2, result.chan3}; //array containing re:
    int*** images = new int**[3] {img.chan1, img.chan2, img.chan3}; //array containing image chan
    int*** filters = new int**[3] {fil.chan1, fil.chan2, fil.chan3}; //array containing filter chan

    for(int i=0; i<3; i++) { //convolute image and filter
        for(int j=0; j<res_r; j++) {
            for(int k=0; k< res_c; k++)
                results[i][j][k] = multiply(images[i], filters[i], j, k, fil.dat_r, fil.dat_c);
        }
    }

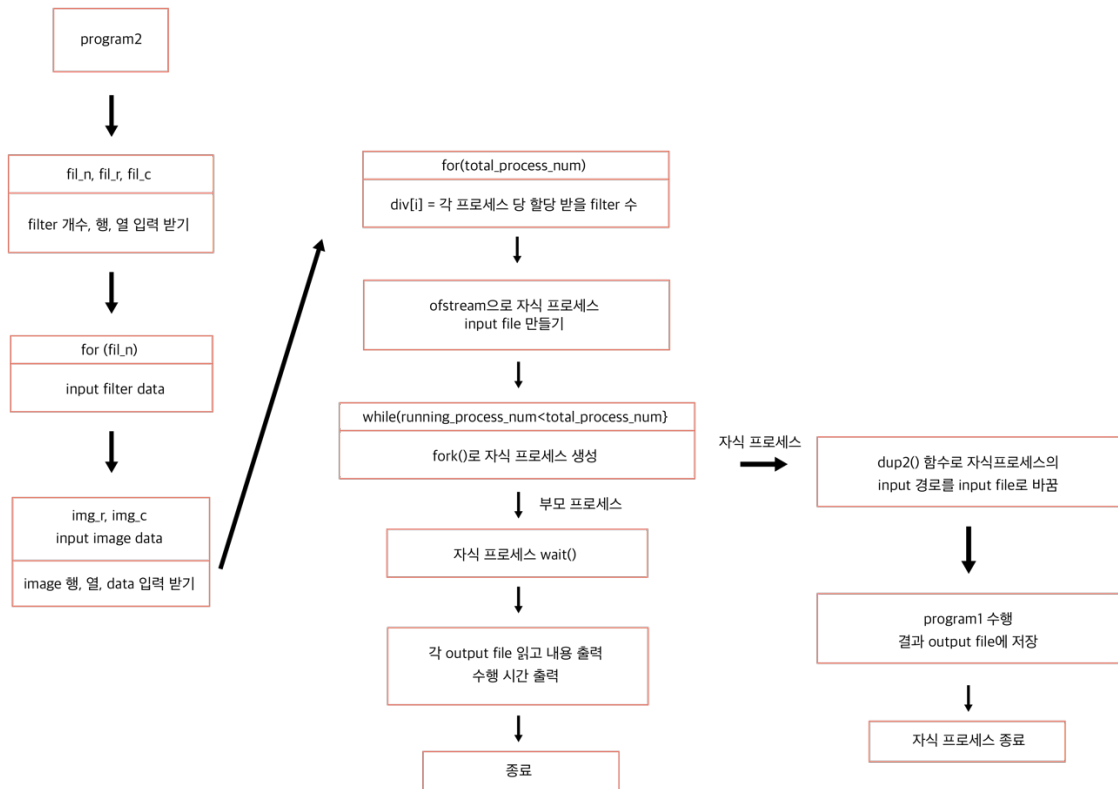
    int** convoluted; //final 2-array result
    convoluted = new int*[res_r];
    for(int i=0; i<res_r; i++) {
        convoluted[i] = new int[res_c];
        for(int j=0; j<res_c; j++) { //add channels
            int val = results[0][i][j] + results[1][i][j] + results[2][i][j];
            if(val < 0) {convoluted[i][j] = 0;} //perform ReLU function
            else {convoluted[i][j] = val;}
        }
    }

    return convoluted;
}
```

우선 결과를 담을 객체 result 를 선언하고 각 객체마다 갖고 있는 채널 3 개를 담을 array 를 선언한다. 그 후 이전에 선언한 multiply 함수를 이용하여 convolution 을 수행하고 결과 array 에 넣는다. 마지막으로 각 결과의 각 채널을 더한 후 음수인 값은 0, 아닌 값은 그대로 둔다. 최종 결과인 2 차원 배열을 리턴한다.



## ii) program2



program1 과 마찬가지로 filter 와 image 에 대한 데이터를 입력 받는다. 그 후 div 라는 array 에 각 프로세스 당 할당 받을 filter 수를 저장한다. 생성할 자식 프로세스 수 만큼 input file 을 만든 후 fork()를 통해 자식 프로세스를 생성한다.

자식 프로세스는 dup2()함수를 이용하여 인풋 경로를 input file 로 설정한 후 program1 을 수행한다. 그 결과는 다시 각 프로세스의 output file 에 저장한다.

부모 프로세스는 자식 프로세스가 종료될 때까지 기다린 후 각 output file 을 입력 받는다. 입력 받은 내용을 출력한 후 수행 시간을 출력한다. 그 후 프로그램을 종료한다.

- 자식 프로세스에 넘길 input file 생성 과정

```
for(int i=1; i<total_process_num + 1; i++) {    //making temporary input file
    string num = to_string(i);
    string dir = "ptoc.txt";
    string name = num + dir;
    const char* file_name = name.c_str();        // temp file name
    ofstream file;                               //output file stream
    file.open(file_name);                        //open stream

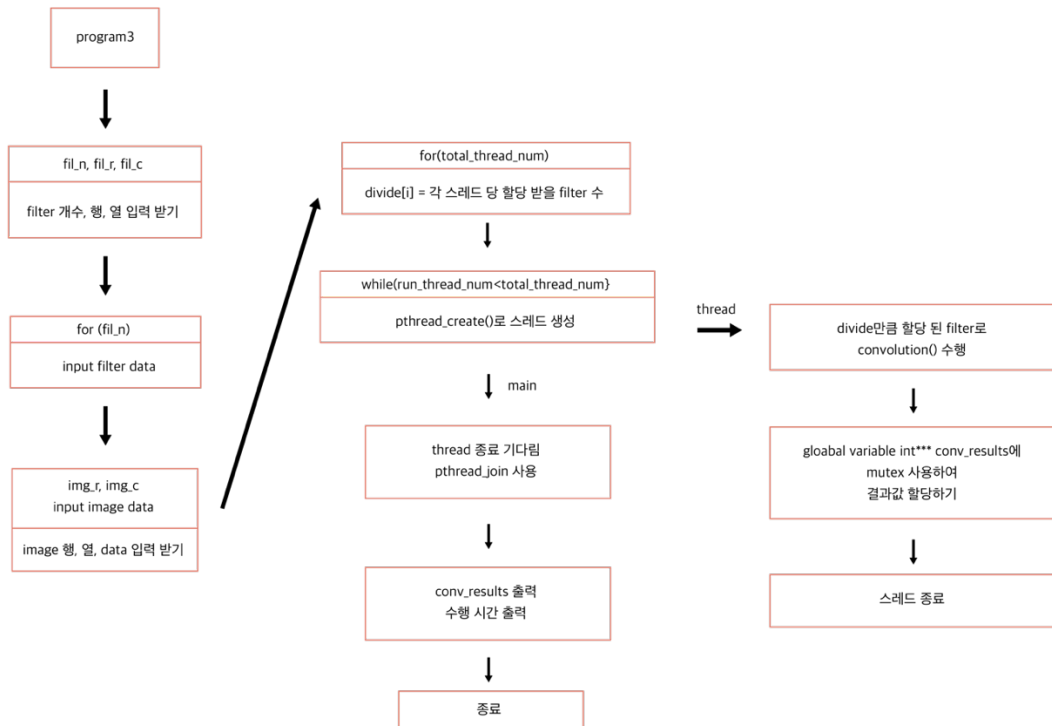
    if(div[i] != 0){                             //write file
        file<<div[i]<<" ";                       //write filter number
        file<<fil_r<<" "<<fil_c<<" ";          //write filter row, column
        int current = 0;                         //filter number initialize
        for(int j=0; j<i+1; j++) {               //current process filter num
            current += div[j];
        }
        int last = current - div[i];             //number of previous process's filter
        for(int j=last; j< current; j++) {       //write filters
            write_data(*fil_vec[j], & file);
        }
        file<<img_r<<" "<<img_c<<" ";          //image row, column
        write_data(img, & file);                //write image
    }
    else{                                         //process with no filters
        file<<<<" "<<<<" "<<<<" "<<<<" "<<<<";
    }
    file.close();
}
```

- 자식 프로세스 생성 코드

```
while(run_process_num<total_process_num) {    //repeat until total process number
    pids[run_process_num] = fork();            //create process
    if(pids[run_process_num] < 0) {            //if fail
        perror( s "fork error");
        exit(EXIT_FAILURE);
    }
    else if(pids[run_process_num] == 0) {      //if child process
        int file_in, file_out;
        char *par[] = { "./program1", NULL };
        string num = to_string( val: run_process_num + 1 );
        string dir_in = "ptoc.txt";
        string name_in = num + dir_in;
        string dir_out = "ctop.txt";
        string name_out = num + dir_out;
        const char* in_name = name_in.c_str(); //input file name
        const char* out_name = name_out.c_str(); //output file name

        file_in = open(in_name, O_RDONLY);
        file_out = open(out_name, oflag: O_WRONLY | O_TRUNC | O_CREAT, S_IRUSR | S_IRGRP | S_IWGRP | S_
        dup2(file_in, fd2: 0);                 //redirect input
        dup2(file_out, fd2: 1);               //redirect output
        execvp( file: "./program1", par);     //execute program1
        close(file_in);
        close(file_out);
        exit(EXIT_SUCCESS);
    }
}
```

### iii) program3



Filter 와 image 데이터를 입력 받은 후 divide array 에 각 스레드 당 받을 filter 수를 저장한다. 이후 입력 받은 수 만큼 스레드를 생성한 후 각 스레드는 할당 받은 filter 로 convolution() 함수를 수행한다. 수행 결과는 전역변수인 conv\_results 에 mutex 를 사용하여 저장한다. 그 후 스레드를 종료한다. 메인 함수는 모든 스레드가 종료될 때까지 기다린 후 conv\_results 에 저장된 결과를 출력한다. 마지막으로 수행 시간을 출력한 후 프로그램을 종료한다.

#### - thread 생성 코드

```

gettimeofday(&start, tz: NULL); //start time
while(run_thread_num < total_thread_num) { //create thread
    tid = pthread_create(&threads[run_thread_num], attr: NULL, thread_conv, (void *)thread_nums[run_thread_num]);
    if(tid != 0) {
        perror("create failed");
        exit(EXIT_FAILURE);
    }
    run_thread_num++; //increase thread num
}
  
```

thread\_conv 는 스레드에 넘겨줄 함수로서, convolution()함수 + conv\_results 에 결과 저장의 과제를 수행한다.

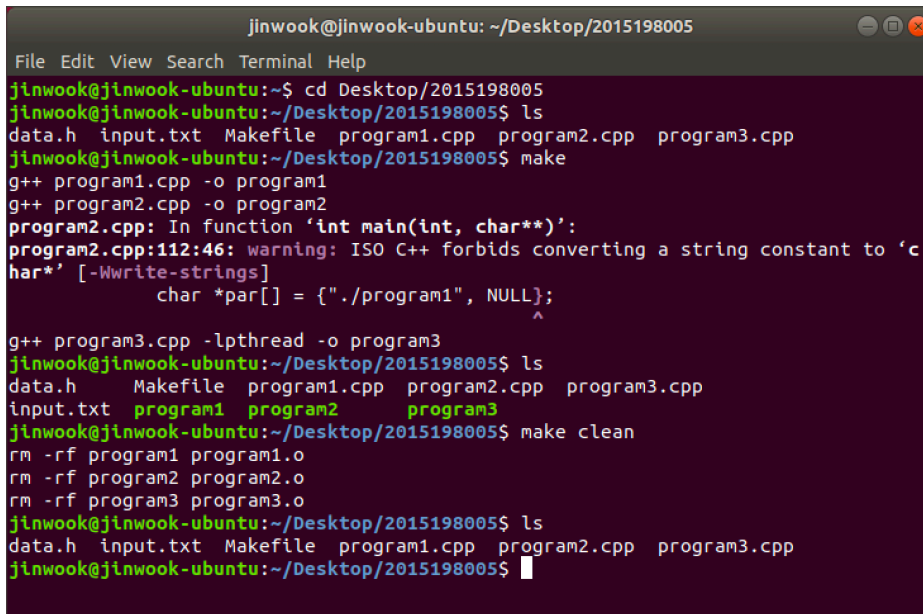
## 2) 작성한 Makefile 에 대한 설명

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

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

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

우선 all 설정을 통해 program1, 2, 3 의 실행 파일을 생성한다. 프로그램을 만들 때 사용한 언어가 C++이기 때문에 g++로 컴파일 하도록 설정하였다. 프로그램을 실행할 때 ./program#의 형식으로 하기 위해 -o program#의 옵션을 추가하였다. 또한 program3 은 pthread 함수들을 사용하기 위해 -lpthread 옵션을 추가하였다. clean 이라는 명령을 통해 만들어진 실행 파일들을 지울 수 있도록 하였다. rm 과 -r 옵션을 통해 프로그램을 완전히 삭제하고, -f 옵션을 추가하여 만약 삭제할 대상이 존재하지 않더라도 에러가 발생하지 않도록 하였다.

A terminal window titled 'jinwook@jinwook-ubuntu: ~/Desktop/2015198005' showing the execution of the Makefile. The user runs 'cd Desktop/2015198005', 'ls', and 'make'. The 'make' command compiles program1.o, program2.o, and program3.o. A warning is shown for program2.o: 'warning: ISO C++ forbids converting a string constant to 'char\*' [-Wwrite-strings]'. The user then runs 'ls' and 'make clean', which removes the compiled files. Finally, 'ls' is run again to show the remaining source files.

```
jinoovk@jinwoovk-ubuntu: ~/Desktop/2015198005
File Edit View Search Terminal Help
jinwoovk@jinwoovk-ubuntu:~$ cd Desktop/2015198005
jinwoovk@jinwoovk-ubuntu:~/Desktop/2015198005$ ls
data.h input.txt Makefile program1.cpp program2.cpp program3.cpp
jinwoovk@jinwoovk-ubuntu:~/Desktop/2015198005$ make
g++ program1.cpp -o program1
g++ program2.cpp -o program2
program2.cpp: In function 'int main(int, char**)':
program2.cpp:112:46: warning: ISO C++ forbids converting a string constant to 'c
har*' [-Wwrite-strings]
    char *par[] = {"/.program1", NULL};
                                ^
g++ program3.cpp -lpthread -o program3
jinwoovk@jinwoovk-ubuntu:~/Desktop/2015198005$ ls
data.h      Makefile  program1.cpp program2.cpp program3.cpp
input.txt  program1  program2      program3
jinwoovk@jinwoovk-ubuntu:~/Desktop/2015198005$ make clean
rm -rf program1 program1.o
rm -rf program2 program2.o
rm -rf program3 program3.o
jinwoovk@jinwoovk-ubuntu:~/Desktop/2015198005$ ls
data.h input.txt Makefile program1.cpp program2.cpp program3.cpp
jinwoovk@jinwoovk-ubuntu:~/Desktop/2015198005$
```

### 3) 개발 환경 명시

#### i) uname -a 실행 결과

```
jlnwook@jlnwook-ubuntu:~$ uname -a
Linux jlnwook-ubuntu 5.4.33-2015198005 #1 SMP Tue Apr 21 03:29:54 KST 2020 x86_64
x86_64 x86_64 GNU/Linux
```

#### ii) 사용한 컴파일러 버전

```
gcc version 7.5.0 (Ubuntu 7.5.0-3ubuntu1~18.04)
```

#### iii) CPU 코어 수

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

#### iv) 운영체제 버전

```
jlnwook@jlnwook-ubuntu:~$ cat /etc/issue
Ubuntu 18.04.4 LTS \n \l
```

#### v) 메모리 정보

```
jlnwook@jlnwook-ubuntu:~$ cat /proc/meminfo
MemTotal:        2034964 kB
MemFree:         126004 kB
MemAvailable:    836692 kB
Buffers:         152484 kB
Cached:         635900 kB
SwapCached:      1212 kB
Active:          925428 kB
Inactive:        706352 kB
Active(anon):    461028 kB
Inactive(anon):  397584 kB
Active(file):    464400 kB
Inactive(file):  308768 kB
Unevictable:     16 kB
Mlocked:         16 kB
SwapTotal:       1942896 kB
SwapFree:        1918400 kB
Dirty:           0 kB
Writeback:       0 kB
AnonPages:       842864 kB
Mapped:          149988 kB
Shmem:           15216 kB
KReclaimable:    125416 kB
Slab:            171024 kB
SReclaimable:    125416 kB
SUnreclaim:      45608 kB
KernelStack:     9056 kB
PageTables:      38736 kB
NFS_Unstable:    0 kB
Bounce:          0 kB
WritebackTmp:    0 kB
CommitLimit:     2960376 kB
Committed_AS:    4529480 kB
VmallocTotal:    34359738367 kB
VmallocUsed:      45980 kB
VmallocChunk:    0 kB
Percpu:          1296 kB
HardwareCorrupted: 0 kB
```

```
AnonHugePages:      0 kB
ShmemHugePages:     0 kB
ShmemPmdMapped:     0 kB
FileHugePages:      0 kB
FilePmdMapped:      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:        163776 kB
DirectMap2M:        1933312 kB
```

#### 4) 과제 수행 중 발생한 애로사항 및 해결방법

##### i) 동적 할당 및 segmentation error

```
int** pad1;           //padded 2d-array
int** pad2;
int** pad3;
pads.push_back(pad1);
pads.push_back(pad2);
pads.push_back(pad3);
channels.push_back(chan1);
channels.push_back(chan2);
channels.push_back(chan3);

for(int i=0; i<pads.size(); i++) {    //copy image to padded array
    for(int j=0; j<dat_r; j++) {
        for(int k=0; k<dat_c; k++) {
            pads[i][j+1][k+1] = channels[i][j][k];
        }
    }
}
```

Process finished with exit code 139 (interrupted by signal 11: SIGSEGV)

입력 받은 image 를 padding 하기 위해 함수 내에서 패딩 된 2 차원 배열을 위한 int\*\*를 선언하였고 여기에 기존 image 를 복사하였다. 이 과정에서 계속해서 segmentation error 가 발생하였다.

```
for(int i=0; i<pads.size(); i++){    //initialize padded 2d-array
    pads[i] = new int*[pad_r];
    for(int j=0; j<pad_r; j++) {
        pads[i][j] = new int[pad_c]();
    }
}
```

조사를 통해 이 에러는 동적 할당이 제대로 이루어지지 않아서 생기는 것을 알게 되었고 new 를 통해 int\*\* 변수를 미리 heap 메모리에 동적 할당 한 후 image 를 복사하니 문제가 해결 되었다.

## ii) program3 수행 시간

program3 를 완성한 후 program2 와 동일한 인풋과 스레드 생성 개수를 넘겨 주었는데도 불구하고 program2 보다 수행 시간이 약 1.5 배 길었다.

```
m.lock();  
conv_results[i] = convolution(*image, *fil_vec[i]);  
m.unlock();
```

이후 조사를 해보니 기존에 코드에서는 global 변수인 conv\_results 에 연산하는 함수의 리턴값을 할당하는 방식으로 되어있었다. 여기서의 문제점은 conv\_results 에 할당하는 코드 전에 mutex lock 을 수행하기 때문에 스레드가 필터 하나로 연산하는 전체 과정 동안 다른 스레드들이 conv\_results 에 접근을 못하는 구조로 되어있다는 것이다.

```
convolution(*image, *fil_vec[i], i);
```

이를 해결하기 위해 우선 convolution 함수를 void 로 바꾼 후

```
m.lock();    //mutex lock  
conv_results[filter_num] = convoluted;  
m.unlock();  //mutex unlock
```

convolution 함수 내부에서 결과 값을 직접 전역 변수인 conv\_results 에 할당하도록 수정하였다. 그 결과 program3 의 수행 시간이 월등히 빨라졌다.

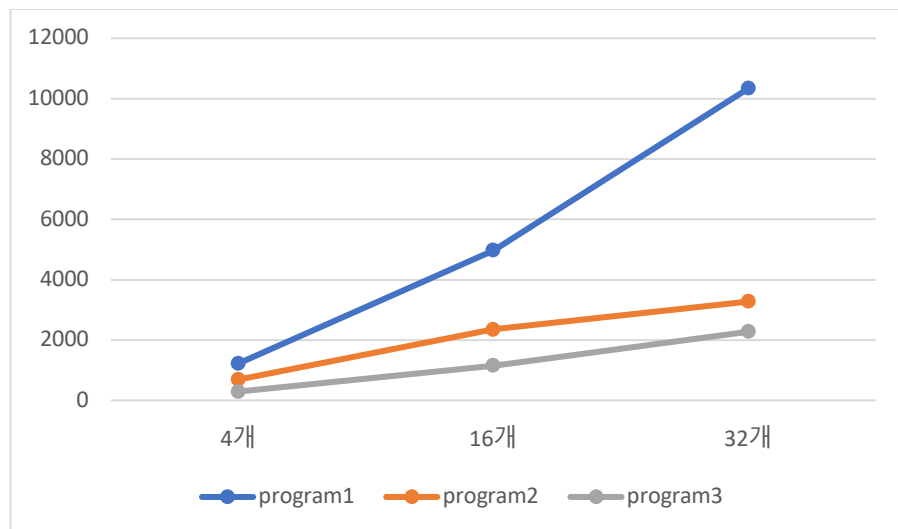
## 5) 결과 화면과 결과에 대한 토의 내용

### i) filter 개수에 따른 수행시간 차이

program2 와 program3 는 각각 프로세스/스레드 생성 개수를 4 개로 하였다.  
또한 필터의 크기는 모두 5\*5, 이미지의 크기는 모두 1000\*1000 이다.

	4 개	16 개	32 개
program1	1224	4966	10336
program2	689	2355	3274
program3	292	1151	2274

수행 결과를 살펴보면 program3, program2, program1 순서로 수행시간이 빠른 것을 볼 수 있다. Filter 의 개수가 늘어나면서 program2 가 program1 보다 빠른 비율도 함께 늘어나고 있는데 이는 프로세스의 생성에 필요한 시간이 고정값으로 존재하기 때문이다. 또한 program2 에 비해 program3 의 수행 속도가 더 빠른 이유는 스레드를 생성하 것이 프로세스를 생성하는 것보다 시간이 덜 소모되기 때문이다.



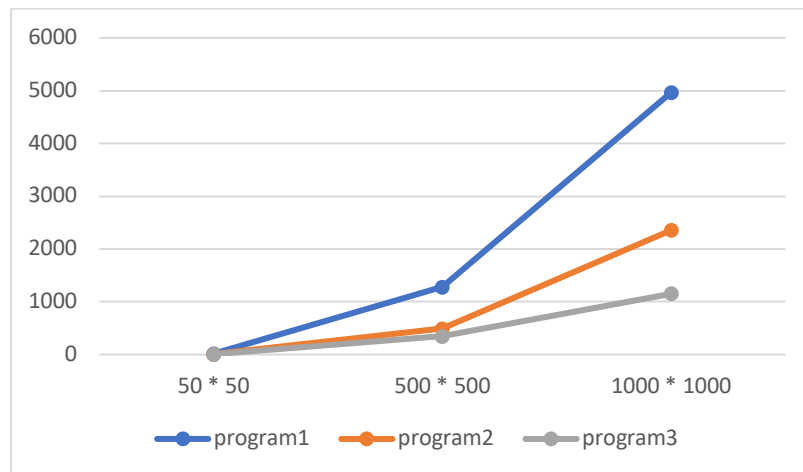


## ii) image 크기에 따른 수행시간 차이

프로세스/스레드 생성 개수는 4 로 하였고, filter 는 모두 16 개의 5\*5 로 주어졌다.

	50 * 50	500 * 500	1000 * 1000
program1	17	1281	4966
program2	9	492	2355
program3	7	348	1151

Filter 개수와 마찬가지로 이미지의 크기를 늘릴 수록 각 프로그램의 수행시간은 길어진다.  
프로그램의 수행시간 또한 program3 – program2 – program1 순서로 빠르다.



## iii) 프로세스/스레드 생성 개수에 따른 수행시간 차이

수행과정에서 filter 의 개수와 크기는 모두 16 \* 5 \* 5, image 의 크기는 모두 1000 \* 1000 로 하였다.

	4 개	10 개	100 개
program2	2355	2143	2561
program3	1151	1148	1174

프로세스/스레드의 개수를 4 개에서 10 개로 늘리면 수행시간도 짧아진다는 것을 확인할 수 있었다. 그러나 개수를 100 으로 하니 오히려 수행시간이 늘어났다. 그 이유는 필터의 개수가 총 16 개이므로 16 개 이상의 프로세스/스레드를 만들기 시작하면 아무런 작업을 하지 않는 프로세스/스레드가 생기는 것이다. 그러나 작업을 하지 않더라도 생성하는 데에는 시간이 소모되기 때문에 전체 프로그램의 수행시간이 늘어나는 것이다. 이를 통해 알 수 있는 중요한

점은 프로세스/스레드의 개수를 무분별하게 늘리는 것은 오히려 프로그램의 성능을 저하시키고, 작업량에 적합한 개수를 설정해야한다는 것이다.

