OS assignment2 - Report

2016112083 김연웅

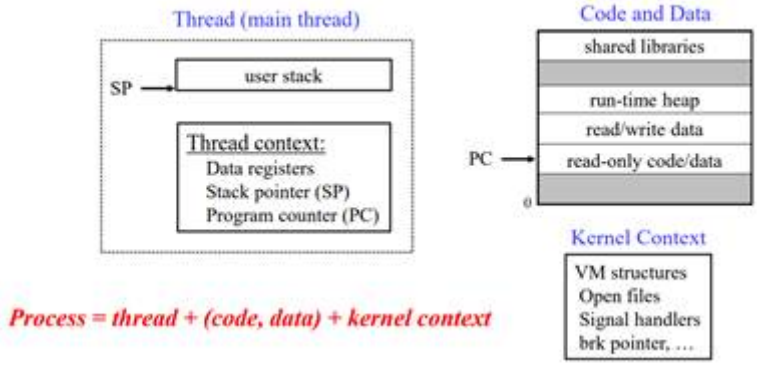
사전 조사 보고서

- **프로세스와 스레드의 차이점.**

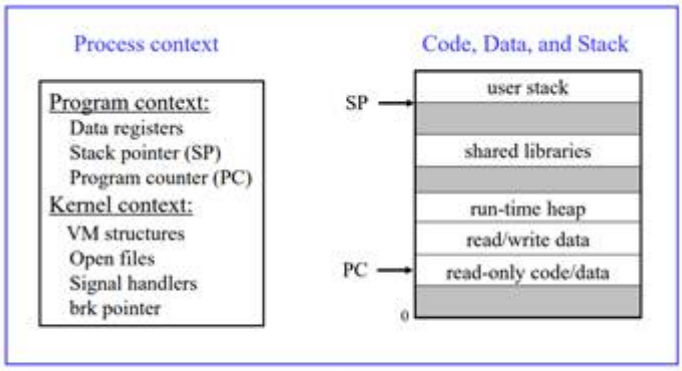
프로세스: 운영체제로부터 시스템 자원을 할당 받은 작업의 단위이며 메모리에 실행되고 있는 프로그램의 인스턴스이다. 프로세스는 크게 코드, 데이터, 스택, 힙 등 네 가지 영역로 구성되어 있는 “Image”와 program context, kernel context로 구성된 Process Context를 포함한다. 먼저 이미지의 Code 영역은 Instruction 코드 자체를 담고 있는 메모리 영역이며 Data는 전역 변수, 자료 구조, static 변수 등 초기화된 데이터를 지니고 있다. Stack 은 지역변수 매개 변수 리턴 값을 지닌 임시 메모리 영역이며, Heap은 동적 할당을 담당하는 영역이다. Program context는 프로세스 자체의 주소공간, CPU상에서의 수행 상태를 나타내는 Program counter 등의 Register 값을 포함한다. Kernel context는 PCB, kernel stack, process state, PID, scheduling information등과 같은 운영체제가 해당 프로세스를 관리할 때 필요한 여러 자료구조와 포인터 등을 저장한다.

스레드: 한 프로세스 내에 존재하는 여러 실행들의 흐름을 지칭한다. 프로세스 내의 자원을 공유하며 다수의 실행을 동시에 실행시키는 것이다. 프로세스가 생성됨과 동시에 프로세스는 최초의 스레드를 생성하며 이것이 메인 스레드이다. 메인 스레드에서 추가로 여러 개의 스레드를 생성해낼 수 있으며 이것을 멀티스레딩이라고 한다. 프로세스의 이미지 영역 중 code, data, heap 등의 자원을 공유하며 각 스레드별로 독립적인 스택을 지닌다.

정리하자면 프로세스는 시스템 자원을 할당 받은 작업의 한 단위로 그 안에 프로세스에 의해 정의된 절차에 따라 실행되는 작업의 수행 흐름이 바로 스레드라고 할 수 있다. 스레드는 동일한 프로세스 내에 존재하기에, 서로 code, data, heap 등의 자원을 공유하며 손쉽게 접근 가능하다. 반면 프로세스는 운영체제로부터 시스템 자원을 할당받고, 각 프로세스간 자원들을 공유되지 않는 보다 독립적인 단위이다. 서로 다른 프로세스간 정보를 공유하기 위해서는 IPC(inter process communication) 기법들(pipe, shared memory)등의 방법을 사용하는 등 절차가 스레드간의 정보 공유보다 복잡하다. 대신, 스레드는 프로세스 내의 자원을 공유하기 때문에, 필연적으로 동기화의 문제가 존재하며, 한 스레드의 오류는 프로세스 전체의 중지를 불러일으킨다. 반면에 한 프로세스의 오류는 해당 프로세스의 중지를 일으킨다 하더라도 독립적인 개체이기에 다른 프로세스, 혹은 이들이 구성하는 전체 시스템에 영향을 끼치지 않는다.



Process = thread + (code, data) + kernel context



Process = process context + (code, data, and stack)

- **프로세스와 스레드의 리눅스에서의 구조 및 구현.**

리눅스 운영체제는 통상적인 개념과 달리 프로세스와 스레드 모두 'task' 라는 개념으로 구현하고 관리한다. 리눅스 커널의 입장에서 스레드와 프로세스는 모두 같은 task_struct라는 구조체 자료구조로 생성하여 관리한다. 이 구조체를 태스크 디스크립터라고 말하며 프로세스의 속성과 상태정보, 그리고 프로세스간의 관계가 담겨있다. tasks라는 멤버 필드를 이용해 다른 프로세스들을 모두 연결리스트로 관리한다고 한다. 리눅스에서의 스레드는 struct thread_info라는 구조체로 생성 및 관리한다. 이 구조체에는 프로세스의 실행 흐름에 관한 정보들을 담겨있으며 스케줄링 실행 이전의 레지스터들과 문맥 정보 등이 있어 이들을 이용해 스레드를 관리한다. 스레드는 공유메모리가 존재하므로 스레드를 만들 때 clone()함수와 인자들을 이용, 새로 생성되는 태스크의 메모리의 공유를 정의 및 설정한다.

- **멀티 프로세스와 멀티 스레딩의 개념 및 구현 방법.**

멀티 프로세스와 멀티 스레딩은 공통적으로 일련의 실행을 동시에 병렬적으로 수행한다는 것이다. 반면 여러가지 차이점이 존재한다. 새로운 프로세스의 생성은, 시스템으로부터 독립적인 자원을 할당 받아 자신만의 메모리공간을 지니게 된다. Fork를 통해 프로세스를 복사한 후 exec계열의 시스템 콜을 활용, 새로운 프로세스들을 발생시킨다. 프로세스간 자원을 주고받거나 통신을 하기 위해서는 IPC기법을 활용해야 하는데 파이프, 공유메모리 등의 기법이 있다. 부모 프로세스는 자식 프로세스의 수행이 끝날 때까지 wati등의 system call을 이용해 기다리게 할 수 있다. 프로세스의 식별은 pid로 하며 getpid 등의 call을 활용한다.

반면 멀티스레딩의 경우, 새로운 스레드는 posix thread API의 pthread 계열 시스템 콜들을 이용해 생성되고 관리된다. 스택을 제외한 모든 자료구조들을 공유하기에 멀티 프로세스의 경우와 달리 생성하는데 드는 비용이 훨씬 적다. 새로운 스레드를 실행할 때 시작할 코드가 있는 void * 타입의 함수와 이 함수에 넘겨줄 인자를 담은 void * struct를 먼저 만든다. 이후 pthread_create을 통해 스레드를 생성하고 생성된 스레드의 실행이 끝날 때까지 기다리는 것은 pthread_join을 이용한다. 스레드의 식별은 pthred_t를 이용한다. 또한 detach 등을 이용해 공유하는 자원을 관리할 수 있다.

- **EXEC 계열의 시스템 콜의 종류와 리눅스에서의 구현, 동작 방법.**

```
int execl(const char *path, const char *arg0, ..., (char *)0);

int execlp(const char *file, const char *arg0, ..., (char *)0);

int execle(const char *path, const char *arg0, ..., (char *)0, char llconst
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[]);
```

exec계열의 시스템 콜은 exec뒤에 l,v,p,e등이 붙어있는데 이 알파벳의 의미에 따라 시스템 콜의 실행이 달리된다. 먼저 l 계열은 실행시키려는 명령어의 인자 (argv)를 문자열(char*)로 나열하여 하나씩 매개 변수에 작성해주어 전달함을 의미한다. 반면 v계열은 실행시키려는 명령의 인자를 문자열 배열(char[]) 안에 넣어 한 번에 전달한다. L과v 모두, 전달하는 마지막 인자로 NULL pointer을 주어야 한다.e계열은 명령어 실행에 환경변수를 전달한다는 의미이다. p계열은 실행시키고자 하는 명령어의 실행파일 및 프로그램을 상대경로로 입력 가능하게 해준다. 즉, 실행 시 시스템 환경변수인 PATH를 자동으로 참조하여 검색해 실행하기에 매개변수로 절대경로를 입력하지 않아도 된다는 의미이다.

exec계열의 시스템 콜은 새로운 프로세스를 실행시킨다. 이때, 실행된 프로세스는 독립적인 개체로 실행되는 것이 아니라, 기존의 프로세스를 대체해버리는 것이 큰 동작 흐름이다. 또한 현재 프로세스의 실행 흐름은 대체되는 프로세스로 넘어가며 이전의 프로세스로 다시 제어권이 돌아가지 않는다. 한편 exec에 의해 실행된 프로세스는 이전의 프로세스의 PID 와 사용중인 file descriptor을 상속받는다. 즉 원 프로세스에서 열려 있던 file descriptor는 여전히 새로이 생성된 프로세스에서도 열린 상태이다.

리눅스 시스템은 위 같은 exec계열의 시스템 콜과 더불어 fork() 시스템 콜을 통해 동작 및 구현된다. Exec계열과 달리 fork는 현재 프로세스의 모든 내용(이미지 program context, PCB)을 복사하기에 fork()가 실행된 이후 자식 프로세스는 부모가 fork를 콜한 그 위치에서 실행되게 된다.(실행흐름까지 복사) fork로 실행된 자식 프로세스는 부모 프로세스가 종료하거나, wait함수에 의해 불리면 더불어 종료되게 된다. 리눅스 시스템은 init이라는 프로그램을 시작으로 연쇄적인 fork()와 exec의 실행으로 모든 프로세스가 동작하게 되는 것이다.

<참조 문헌>

https://sosal.kr/37

https://noel-embedded.tistory.com/1198

https://blackinkgj.github.io/fork_and_exec/

https://www.guru99.com/difference-between-multiprocessing-and-multithreading.html

https://austindhkim.tistory.com/40?category=824072o

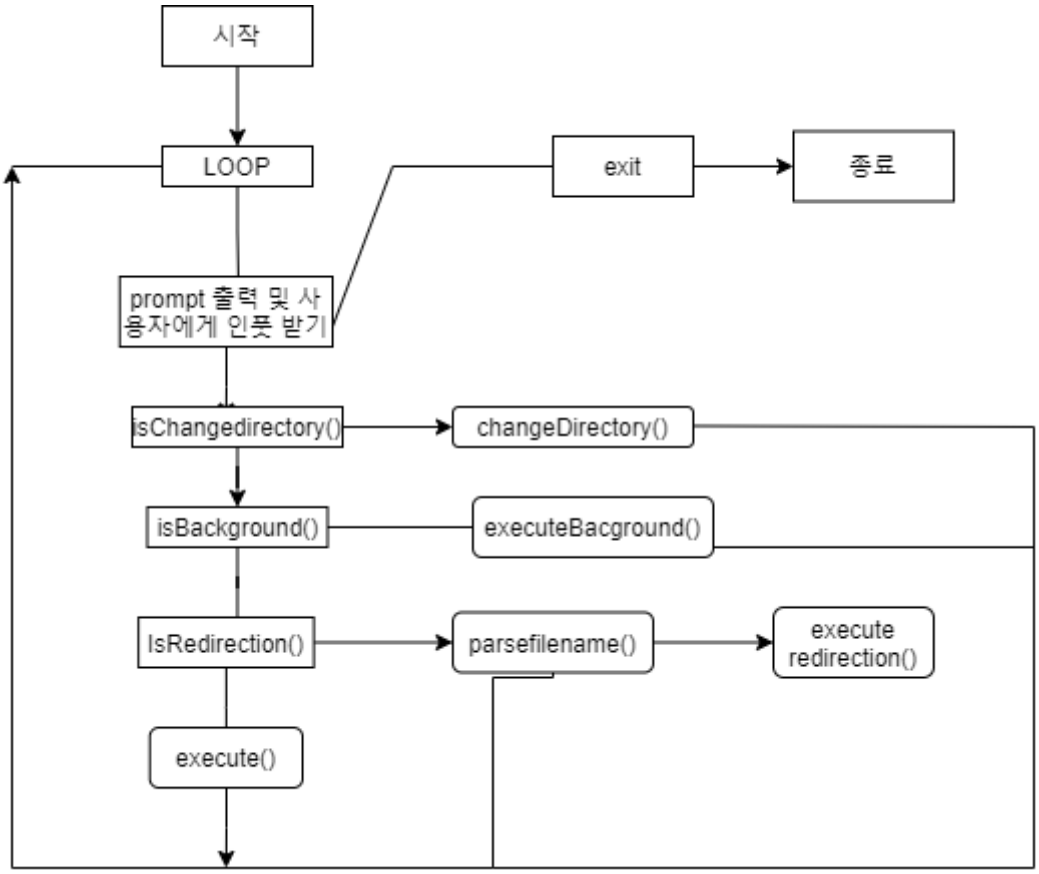
os 수업 피피티 (차호정 교수님)

프로그래밍 수행 결과 보고서

작성한 프로그램의 동작 과정과 구현 방법

실습과제 1 프로그램에 대한 서술

main() 함수



miniShell.cpp는 위의 도식처럼 작동한다. 우선 prompt 출력 및 사용자 인풋을 입력받는다. 이후 isChangedirectory(), isBackground(), isRedirection()과 같은 함수들을 이용해 명령어의 종류를 판단한다.(exit명령어/cd명령어/background실행/redirection실행/normal 실행) 판별이 끝나면 실행 흐름은 분기하여 해당하는 명령어의 수행을 changeDirectory(), executeBackground(), executeRedirection(),execute()를 통해 수행한다. 이후 다시 반복문인 loop로 돌아가게 하여 쉘의 기능을 구현하였다.

prompt 출력 및 입력받기 과정

printPrompt() 함수를 이용해 prompt를 출력한다.

```
void printPrompt(){
    /*
    print prompt message
    */

    // get PWD
    char pwd[250];
```

```
    getcwd(pwd,sizeof(pwd));

    // get name
    char *name;
    name = getlogin();

    // get current time
    time_t current_time = time(NULL);
    struct tm *pLocal = localtime(&current_time);

    // print prompt
    printf("[%02d:%02d:%02d]%s@%s$", pLocal->tm_hour,pLocal->tm_min, pLocal->tm_sec, name,pwd);
}
```

input()함수를 이용해 input을 받는다.

```
bool input(vector<string> &args){
    /*
    get user input int args vector

    */
    // clear the vector before get new inputs
    args.clear();

    // get input
    string input;
    getline(cin,input);

    // tokenizing begins here
    // first add spaces for convenience of further tokenizeing
    int prev=0;
    int curr =0;

    // if '<' found, add space before and after '<' position
    curr = input.find('<');
    while(curr != string::npos){
        input.insert(curr, " ");
        input.insert(curr + 2, " ");
        prev = curr + 3;
        curr = input.find('<', prev);

    }
    prev = 0;
    curr = 0;

    // if '>' found, add space before and after '>' position
    curr = input.find('>');
    while(curr != string::npos){
        input.insert(curr, " ");
        input.insert(curr + 2, " ");
        prev = curr + 3;
        curr = input.find('>', prev);

    }
    prev = 0;
    curr = 0;
```

```
    // if '&' found, add space before and after '&' position
    curr = input.find('&');
    while(curr != string::npos){
        input.insert(curr, " ");
        input.insert(curr + 2, " ");
        prev = curr + 3;
        curr = input.find('&', prev);

    }

    // string tokenizing using strtok

    // in order to use strtok, convert string type to char* type.
    char c_string[strlen(input.c_str())];
    strcpy(c_string,input.c_str());

    // tokenizing and push each tokens to args vector.
    char *ptr = strtok(c_string, " ");
    while( ptr != NULL){
        args.push_back(ptr);
        ptr = strtok(NULL, " ");
    }


    // if user dosen`t put any inputs, return false
    if(args.size() > 0) return true;
    else return false;
}
```

input() 함수는 사용자 입력을 받고 이를 적절히 tokenizing을 진행한다. 먼저 < > & 입력에 대해 띄어쓰기를 하지 않은 경우를 고려하여, 해당 입력을 찾은 후 앞뒤에 스페이스를 추가한다. 이후 띄어쓰기를 기준으로 tokenizing을 실시한 후 string 토큰들을 vector에 넣어 저장하고 리턴한다. 토큰들을 string으로 저장하였으므로 차후 exec시킬때에는 토큰들을 모두 char*타입으로 바꾸어 주는 과정이 필요하다.

cd 명령어

```
void changeDirectory(vector<string> args){
    /*
    command cd
    */

    // if number of argument tokens are larger than 3, return error
    if(args.size() >= 3)
    {
        cout << "cd: too many arguments\n";
    }

    // for valid input, change working directory.
    else if(args.size() <2 || args[1].compare("~") == 0){
        chdir("/");
```

```
    }
    // succeed
    else if(chdir(args[1].c_str()) == 0){
        return;
    }
    // if failed, print error message
    else{
        cout <<"cd: "<< args[1] << ": No such file or directory\n";
        return;
    }


}
```

cd 명령어의 구현은 다음과 같다. 우선 토크나이징된 인풋의 토큰 개수가 3개 이상이라면, 에러메시지를 출력한다. ~ 혹은 아무런 입력을 받지 않은 경우 home directory로 chdir("/") 하며 실패시 에러메시지를 출력한다.

redirection 명령어

redirection은 두가지 과정을 거친다. 먼저 redirection 연산의 input file, output file을 다시 tokenizing 하는 parse_filename()라는 함수와, fork() 이후 파일 디스크립터를 수정하고 exec 명령어를 하는 과정을 수행하는 execute_redirection()함수이다.

parse_filename()

parse_filename은 < 와 > 문자를 찾아내고 뒤에 입력된 file name을 vector 에 저장한다. 만일 사용자가 여러가지의 file input, file output 명령어를 입력한다면, 가장 마지막에 입력된 input file name과 output file name을 vector에 저장한다. 이때 벡터의 첫번째 인덱스는 input filename을 두 번째 인덱스는 output filename을 저장한다. 동시에 redirection 관련한 명령어 부분은 arg vector에서 처리와 동시에 pop out 시켜 차후 execvp를 수행할 수 있게끔 하였다.

(ex) ./program1 <inputs.txt >outputs.txt <inputs2.txt >outputs2.txt

벡터에 저장된는 파일이름들 -> inputs2.txt, outputs2.txt

execute_redirection()

execute_redirection은 input()의 결과물인 벡터와 parse_filename()의 결과물인 벡터를 인자로 받아 redirection 명령어를 수행한다. 먼저 fork()를 수행하여 새로운 프로세스를 발생시키고, file name에 해당하는 file을 open한 후 , dup2 시스템콜을 이용해 새로만든 프로세스의 파일 디스크립터를 변경한다 이후 execvp를 통해 남은 arg들을 전달하여 명령어를 실행시킨다. 이후 부모프로세스에서는 wait 시스템 콜을 통해 기다린다.

normal 명령어

execute()

cd, exit, background redirection 어느 타입에도 해당되지 않는 명령어이다. fork를 통해 새로운 프로세스를 만들고, exec으로 명령어를 실행한다. 이후 마찬가지로 wait으로 자식 프로세스가 종료되길 기다린다.

background명령어

&가 붙은 background명령을 실행한다. 실행하는 원리는 위의 execute_redirection과 execute함수들을 이용하는데 명령어가 수행되는 자식프로세스의 수행 완료를 wait함수를 사용하지 않고 기다리지 않는 것이다. background 명령어는 normal 명령어 타입일수도, redirection명령어 타입일수도 있으므로 두 가지 경우를 모두 고려하여 분기하게끔 구현했다.

실습과제2에서 구현한 3개 프로그램에 대한 서술

program1

일반적인 merge sort 알고리즘을 구현했다. mergeSort()로 분할을 진행하고 merge()로 합친다.

인풋 을 받아 sort를 진행하는 배열인 int* array에 저장했고 이 변수는 전역변수로 선언했다.

```
void mergeSort(int *array, int first, int last)
{
    int mid = (first + last) / 2;

    // if a splitted piece is bigger than 1 keep split and merge recursively
    if (first < last)
    {
        mergeSort(array, first, mid);
        mergeSort(array, mid + 1, last);
        merge(array, first, last);
    }
}
```

first와 last는 각각 sorting하려는 배열의 인덱스이다. first>last 이면(sort해야될 array크기가 1이라면) 재귀호출을 중지한다.

```
void merge(int *array, int first,  int last)
{
    // merge function
    int* temp = new int[last - first + 1];

    int mid = (first + last) / 2;
    int i, j, index;
    i = first;      // First array index
    j = mid + 1;    // Second array index
    index = 0;      // temporarily sorted array idx
```

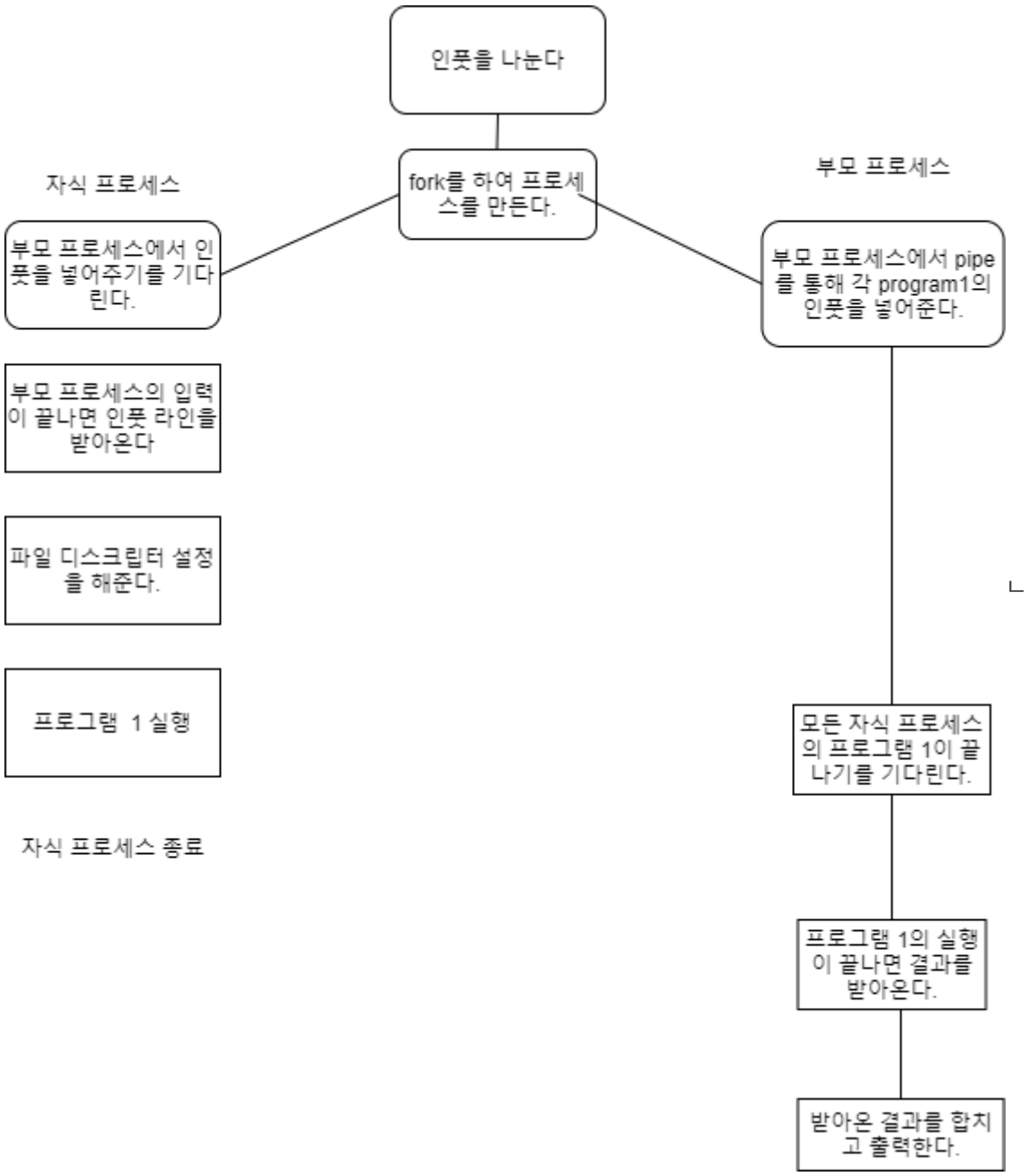
```
    // compare each one element of first array and second array, then sort it to
temp array
    // untill either one of them run out its all the elements.
    while (i <= mid && j <= last)
    {
        if (array[i] >= array[j]){
         temp[index++] = array[i++];
        }
        else{
            temp[index++] = array[j++];
        }
    }
    // if last array still has elements
    if (i > mid){
        for(int x = j; x <= last; x++){
            temp[index++] = array[x];
        }
    }
    // if first array still has elements.
    else{
        for(int x = i; x<=mid; x++ ){
            temp[index++] = array[x];
        }
    }

    // copy the sorted array 'temp', to original array
    for (i = first, index = 0; i <= last; i++, index++) {
        array[i] = temp[index];
    }

    delete[] temp;
}
```

합치는 함수이다. 왼쪽 배열과 오른쪽 배열을 합치는 과정이다. first는 합치려는 배열의 시작 인덱스 last는 합치려는 배열의 마지막 인덱스이다. 중간을 나누어 왼쪽 배열과 오른쪽 배열이라고 가정하고, 각각 element들을 비교하여 큰 쪽을 temp array넣어준다. 쭉 진행하다가 둘중 한 쪽 배열에 원소가 없어졌을 때, 남은 쪽 배열의 원소들을 모두 temp array에 넣어준다. 이후 sort 완료된 temp의 배열을 다시 원 배열에 넣어준다.

program2

program2는 받은 인풋을 분배하여 total_num_process만큼 프로세스를 생성 후, 앞서 구현된 program1을 이용해 sort를 진행 후 다시 합치는 프로그램이다. 다수의 프로세스를 만들고 program 1을 실행시키는 과정에서 IPC기법의 pipe를 사용하여 program1에 인풋을 보내고 아웃풋을 받아왔다.

프로그램의 기본적인 동작 흐름은 다음과 같다.



인풋 나누기 parseInput()

```
void parseInputs(int first, int last, int depth, vector<string> &input_lines,
vector<int>&len_input)
{
    int mid = (first + last) / 2;
    // if spliting reaches max_depth, save current splitted part as an input line
for a child process.
    if(depth == max_depth && remain_divide > 1){
        string input = "";
        input += to_string(last - first + 1);
        input += " ";
        len_input.push_back(last - first + 1);
        for(int i= first; i <= last; i++){
            input+= to_string(array[i]);
            input += " ";
        }
        input_lines.push_back(input);
        remain_divide -=1;
        return;
```

```
    }
    // if spliting reaches max_depth, save current splitted part "PLUS" rest of
picese part of array as an input line for a child process
    else if(remain_divide == 1){

        string input = "";
        input += to_string(N - first);
        input += " ";
        len_input.push_back(N - first);
        for(int i= first; i <= N-1; i++){
            input+= to_string(array[i]);
            input += " ";
        }
        input_lines.push_back(input);
        remain_divide -=1;
        return;

    }
    if (first < last )
    {
        // stop split when all the piceces reauired for total_num_process are
already exists.
        if(remain_divide >0){parseInputs(first, mid, depth+1,input_lines,
len_input);}
        if(remain_divide >0){parseInputs( mid + 1, last, depth+1, input_lines,
len_input);}

    }
}
```

program1의 합병정렬의 기본 함수를 변형해 만들었다. program1에 각각 들어갈 input line을 string형태로 vector에 넣어주어 반환한다.

우선 나누기는 max_depth만큼 split을 진행한다. total_process_number만큼의 생성할 조각들을 미리 max_depth 로 설정해 maxdepth에 도달하면 분할을 멈춘다. 마찬가지로 remain_divide라는 변수는 total_process_number와 같은 수를 가지고 시작해, 하나씩 조각이 정해지면 -1을 해주며 모든 프로세스를 위한 인풋이 저장되었다면 마지막 프로세스는 해당 인덱스부터 끝까지 모든 남은 인덱스를 저장하게 하였다.

IPC

부모 프로세스가 자식 프로세스에게 인풋을 보내고 결과를 받아오는 pipe_P2C, 자식 프로세스가 부모 프로세스에게 인풋을 받아오고 결과를 받아오는 pipe_C2P , 총 두개의 pipe를 사용했다. 자식 프로세스에서는 파이프들에 부모의 입력이 완료되었는지를 확인하기 위해 fd_set을 이용해 입력이 끝나기를 기다린 후 입력을 받아왔다.

program3

program3는 posix pthread api를 이용하여 멀티스레딩 기법을 구현한 merge sort이다. 멀티 프로세스를 위한 mergeSort_thread함수와 인자로 전달할수 있는 ARG structure을 추가로 구현했다. 이후 num_total_thread만큼 pthread_create()를 통해 mergeSort_thread, ARG structure등을 통해 스레드를 생성해낸다. main thread에서는 pthread_join을 통해서 생성된 스레드들이 끝나기를 기다린다. 이후 결과물들을 다시 기존의 mergeSort함수를 사용해 합친다.

인풋 나누기와 합치기는 program2와 동일한 방법을 사용했다.

복잡한 IPC 과정이 없어 보다 구현에 있어 편리했다.

작성한 Makefile에 대한 설명

```
.DEFAULT_GOAL:=all

all: miniShell program1 program2 program3


miniShell: miniShell.cpp
    g++ -std=c++11 -o miniShell miniShell.cpp
program1: program1.cpp
    g++ -std=c++11 -o program1 program1.cpp
program2: program2.cpp
    g++ -std=c++11 -o program2 program2.cpp
program3: program3.cpp
    g++ -std=c++11 -pthread -o program3 program3.cpp
clean:
    rm miniShell program1 program2 program
```

.DEFAULT_GOAL 변수로 Makefile의 기본 goal을 all변수로 설정했다. all 변수에는 miniShell, program1, program2 program3를 저장했고 각각의 명령어를 g++로 컴파일하게 했다. 컴파일 옵션은 c++11을 사용하게 했다.

또한 clean으로 만든 파일들을 삭제하게 해주었다.

개발 환경 명시

- **uname - a 실행결과**

```
eric@ubuntu:~$ uname -a
Linux ubuntu 5.10.19-2016112083 #1 SMP Thu Mar 11 09:06:07 PST 2021 x86_64 x86_6
4 x86_64 GNU/Linux
eric@ubuntu:~$
```

- **사용한 컴파일러 버젼**

```
eric@ubuntu:~$ g++ --version
g++ (Ubuntu 7.5.0-6ubuntu2) 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.

eric@ubuntu:~$
```

CPU 6코어, 메모리 4기가바이트

```
eric@ubuntu:~$ cat /proc/meminfo | grep 'MemTotal'
MemTotal:        3995220 kB
```

```
eric@ubuntu:~$ grep -c processor /proc/cpuinfo
6
```

과제 사용중 발생한 애로사항 및 해결방법

- 멀티 스레딩을 적용한 program3 의 수행 시간이 멀티프로세싱을 적용한 program2의 수행시간보다 적어야하는데 반대의 결과들이 발생했다. 수행하는 시간을 측정하는 함수를 clock_t clock()에서 clock_gettime()으로 교체에서 사용했고 이에도 수행시간의 결과가 예상과는 달랐다. 그래서 인풋 사이즈 크기가 문제라고 생각이 들어 인풋 사이즈를 2만5천개인 파일을 만들어 수행해본 결과 본래 예상했던 결과대로 나왔다.
- program 2 구현시 pipe기법을 사용할때 많은 어려움을 겪었다. 원하는 대로 동작하지 않아 많은 수정과 디버깅 과정을 해본 결과, 문제의 근원이 **자식 프로세스에서 부모프로세스로부터 받은 인풋 정보를 읽어올때 발생**한다는 것을 깨달았다. 많은 시도를 해 보았지만 도저히 고쳐지지 않았다. 알고보니, pipe의 변화를 감시, 동기화하는 fd_set이라는 구조체와 FD_SET()과 같은 기능이 존재했고 이를 사용하니 문제가 해결되었다.
- 구현이 완료된 후 결과 분석의 과정에서, 큰 사이즈의 인풋을 넣어야 함을 깨닫고, 큰 사이즈의 인풋 파일을 생성, 테스팅을 했다. 그 결과 약 만개에서 만 오천개의 인풋 사이즈에서 program2에서 무한대기상태에 빠져버리는 오류를 발견했다. 이는 string이나 char배열을 통해 IPC를 진행하는데 이때 메모리 사이즈를 초과해서 발생하는 에러로 생각된다. 이에 큰 사이즈의 인풋에 대해서는 프로세스, 스레드 수를 크게 늘려 string 메모리 초과를 하지 않게끔 충분히 잘게 쪼개지는 조합으로 인풋하여 테스팅했다.

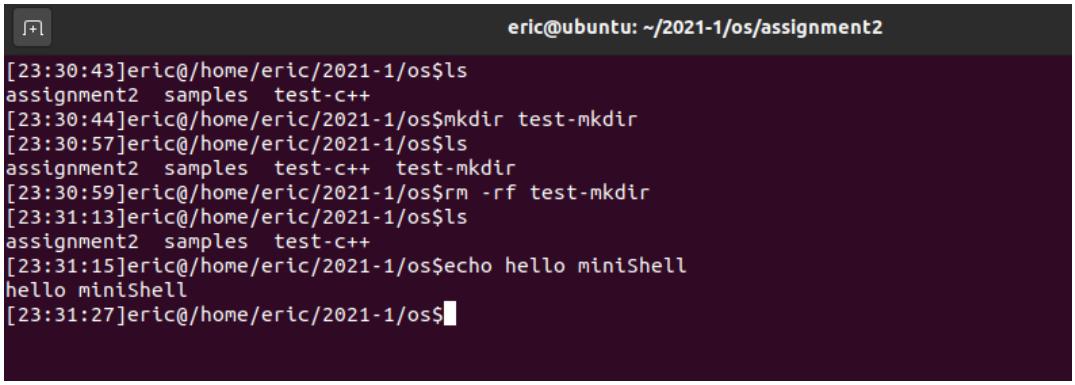
결과 화면과 결과에 대한 토의 내용

- **실습 과제 1의 동작 검증**
  - cd **명령어**

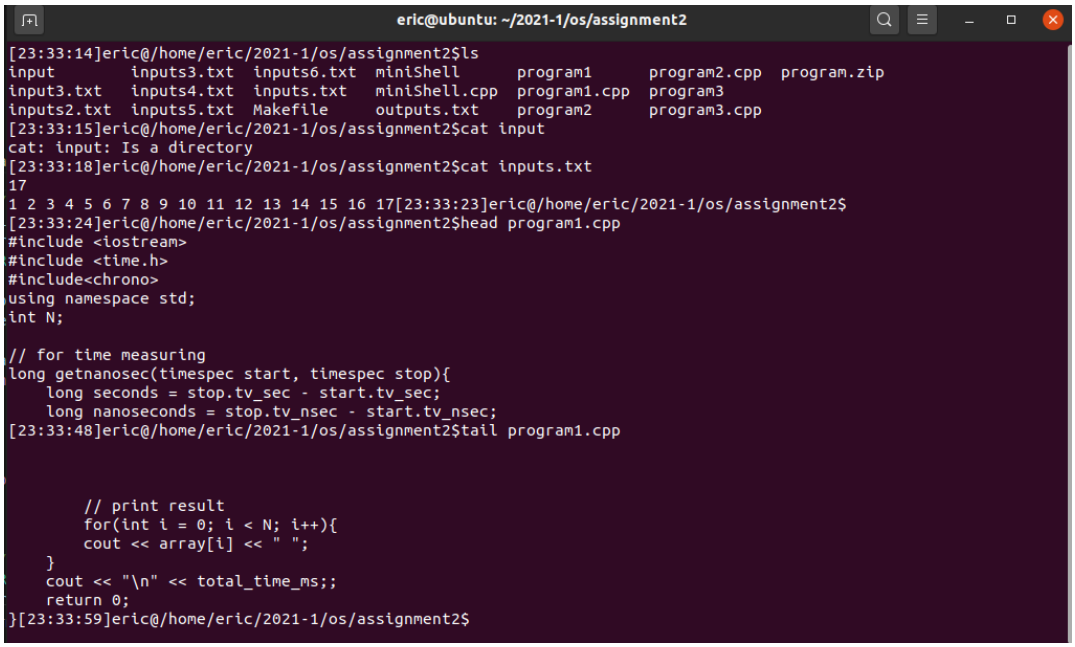
```
eric@ubuntu: ~/2021-1/os/assignment2
[23:27:49]eric@/home/eric/2021-1/os/assignment2$
[23:27:55]eric@/home/eric/2021-1/os/assignment2$cd ../
[23:27:58]eric@/home/eric/2021-1/os$cd
[23:28:01]eric@/$cd /home/eric/2021-1
[23:28:09]eric@/home/eric/2021-1$cd /os/assignment2
cd: /os/assignment2: No such file or directory
[23:28:18]eric@/home/eric/2021-1$cd os/assignment2/
[23:28:44]eric@/home/eric/2021-1/os/assignment2$cd asdfasdf
cd: asdfasdf: No such file or directory
[23:28:59]eric@/home/eric/2021-1/os/assignment2$
```

  - **표준 유닉스 프로그램 (ls, mkdir, rm, echo, cat, head, tail, time 등 환경변수에 이름이 있는 프로그램)**

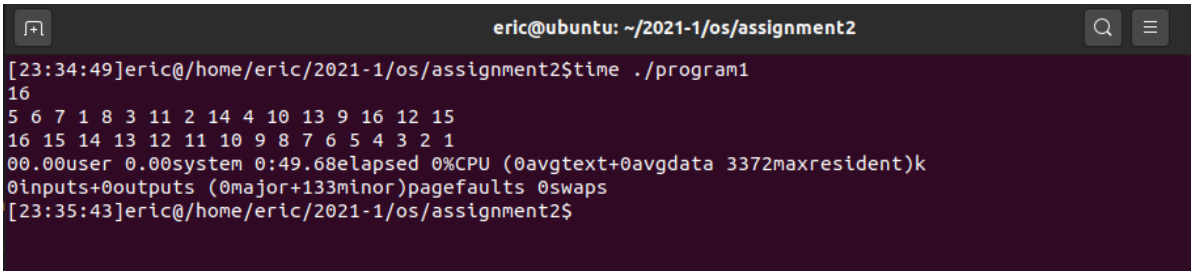
    ls, mkdir, rm, echo



cat head tail

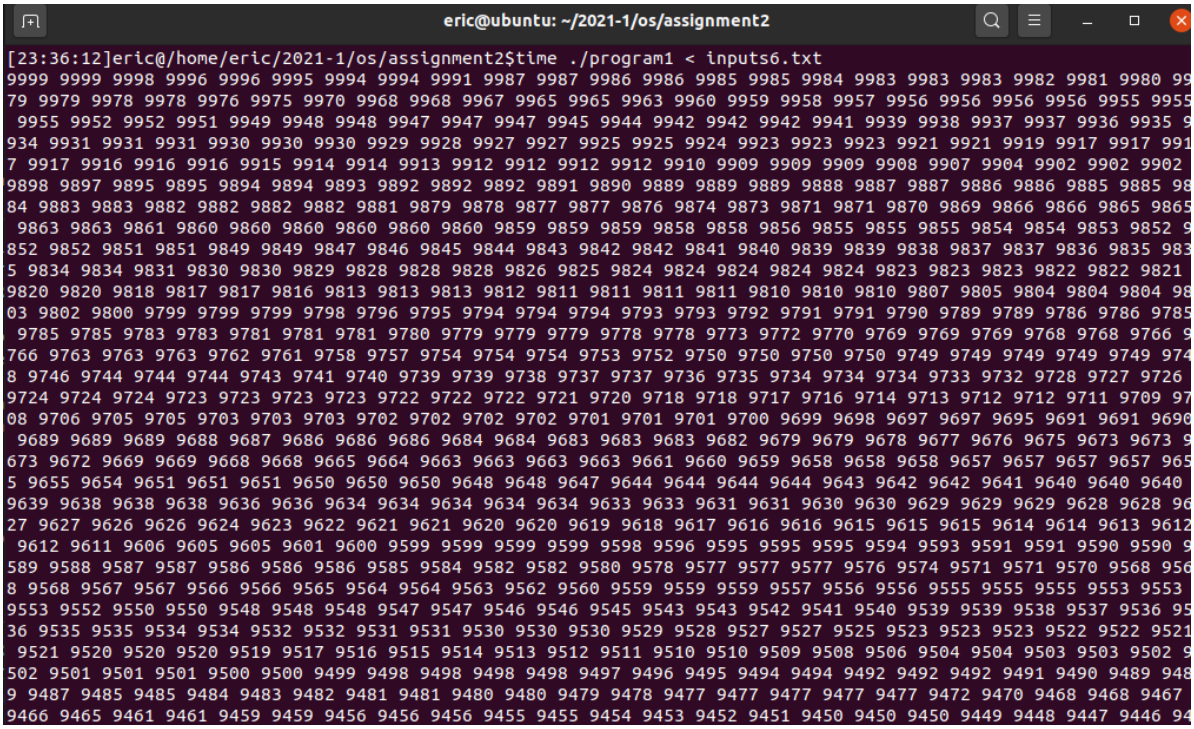


time (user input for program1)

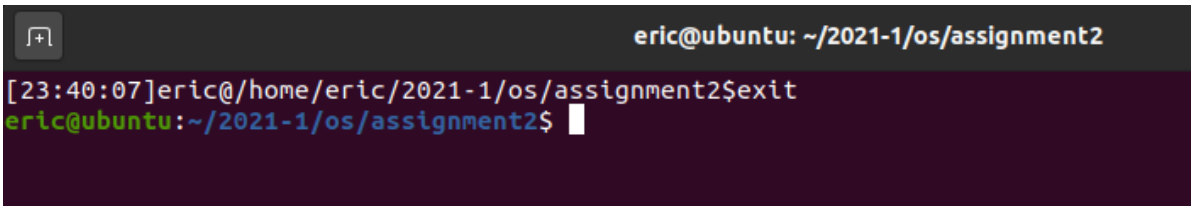


time(input redirection for program 1)

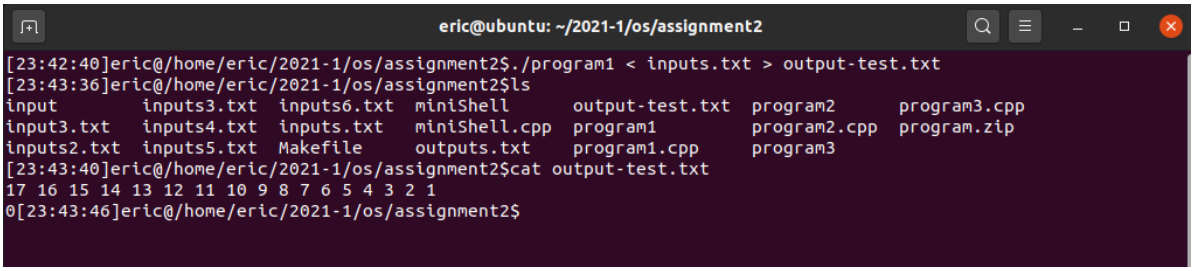
-> inputs6.txt는 25000개의 size이며, -9999~9999까지의 랜덤한 숫자들이 입력된 파일이다.



exit



입출력 리다이렉션과 프로그램 실행



백그라운드 명령

백그라운드에서 생성된 ls 명령어 프로세스의 pid가 출력됨. ps를 통해 ls 확인

- **실습과제 2의 동작검증**

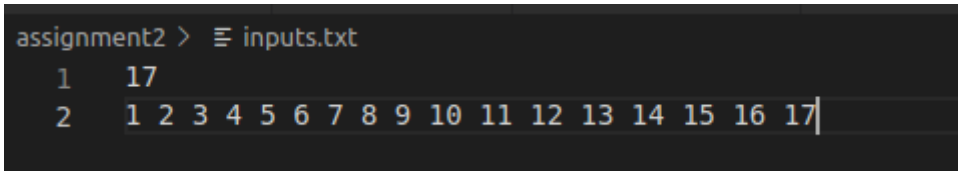
테스트한 인풋 파일 정보

파일 이름	인풋 사이즈	설명
inputs.txt	16	과제 스펙의 예시와 동일
inputs4.txt	5000	-9999~9999사이의 랜덤한 숫자
inputs6.txt	25000	-9999~9999사이의 랜덤한 숫자

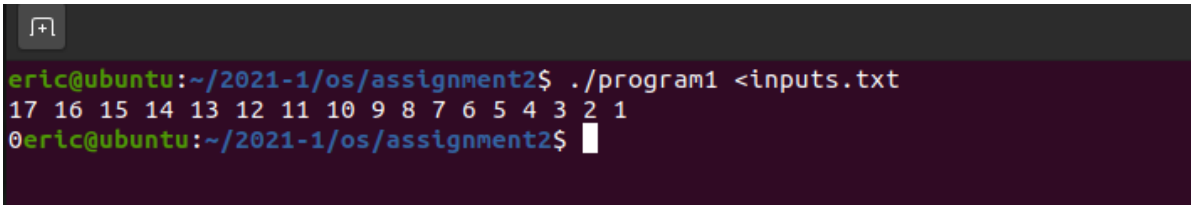
프로그램1, 프로그램2, 프로그램3의 정렬 검증

보고서에 담기 위해 가장 작은 사이즈의 인풋 파일인 inputs.txt로 검증

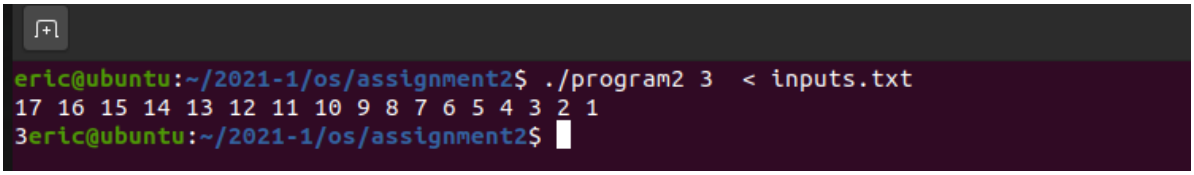
inputs.txt:



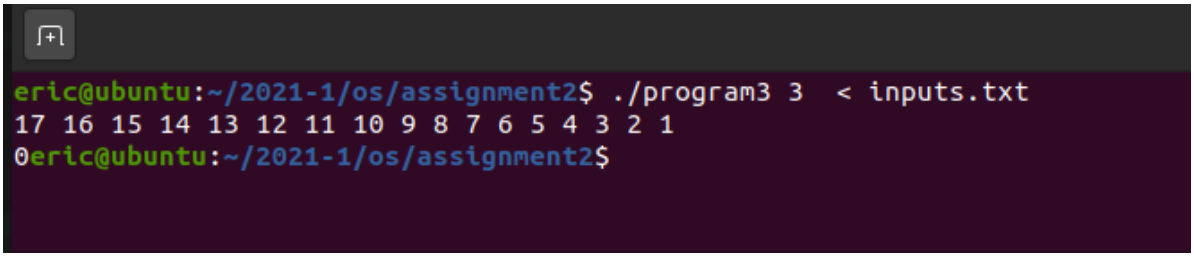
program1 정렬 결과



program2 정렬 결과



program3 정렬 결과



프로그램들의 성능 분석

가설1: program2, program3의 경우 total num process/ total num thread 가 증가할 수록 수행 시간이 늘어날 것.
가설2: program2의 멀티 프로세싱 때문에 program2가 멀티 스레드 기법을 사용한 program3보다 수행 소요시간이 더 걸릴 것.
가설3: 인풋 사이즈가 작을때에는, 멀티 프로세스, 멀티 스레딩에 의한 분할 처리 효과보다 생성하고 수행하는데 소요되는 cost의 효과가 더 커서 program1이 더 빠를 것이다.(인풋 사이즈가 커지면 어느 순간에는 program2/program3가 program1보다 빨라질 것.)

- **inputs.txt의 결과. (input size 16)**

  (input size는 16이며 첫 행의 각 열은 시험한 total process num(program2) 혹은 total thread num(program3) 를 의미함)

  각 수행을 한 후 충분한 시간적 간격을 두고 시행하여 혹시 모를 지연을 최소화했으며 각 조건마다 5회 시행, 보고서에는 5회중 가장 작은 결과를 기록했다.

	2	4	8	16
program1	0ms	-	-	-
program2	1ms	2ms	5ms	9ms
program3	0ms	0ms	0ms	1ms

** program1은 arg가 없으므로 수행 결과를 첫 열에 작성함.
**inputs size가 16이므로 생성할 수 있는 최대 프로세스/스레드 숫자인 16까지 넣어봄.(이상의 수를 넣으면 동작 불가)

인풋 사이즈가 가장 작은 테스트 케이스이다. 멀티 스레딩과 멀티 프로세싱을 사용한 program2, program3의 overhead효과 때문에 수행 시간이 가장 빠를 것이라 생각된다.

program 1은 0ms 소요되었다.

total process num 이 커질 수록 program2 는 점점 소요시간이 늘어났다. 이는 멀티 프로세스를 생성하는 과정과, 수행시 발생하는 context switching에서 오는 cost, pipe를 통한 입출력 IPC에서 발생하는 cost등을 원인으로 생각했다. **만일 그렇다면, 후에 더 큰 사이즈의 인풋파일에서 더 많은 수의 total process를 생성한다면, 프로그램3와 비교했을 때 각 total num process(thread)에 따른 시행간의 수행 시간 차이가 더욱 벌어질 것으로 예상된다.**

program 3은 total thread num을 증가시킬수록 늘어나는 소요시간의 정도가 program2에 비해 적었다. 거의 변화가 없을 정도로 수행속도에 차이가 보이지 않았다.

program 3의 오버헤드 효과가 예상보다 적었다. 아무래도 스레드의 오버헤드는 생성하는 스레드의 수가 훨씬 많아져야 확인할 수 있을것이라 생각했다.

program2는 program1과 비교했을때 미약하지만 가설 3에 부합하는 결과라고 생각된다.

- **inputs4.txt의 결과(input size 5000)**

	2	50	150	255	2000	4096
program1	0ms	-	-	-	-	-
program2	24ms	83ms	159ms	317ms	-	-
program3	7ms	22ms	37ms	69ms	1982ms	2187ms

  인풋 파일이 커져서 좀 더 많은 수의 total num process/ total num thread를 시험할 수 있었다.

  먼저 시험을 해보며 알아낸 사실로는 multiprocess는 총 255개까지의 process를 실행 가능하다는 점과 multithreading은 총 4096개의 thread를 생성 가능하다는 점이다. 더 큰 숫자는 모두 segmentation fault(core dumped)가 발생했다.

  input 사이즈가 커졌음에도 불구하고 program1의 소요시간에는 유의미한 차이가 발생하지 않았다.

반면 total num process/ thread의 **전체적인 소요시간의 증가**를 보아 가설 1에 부합되는 결과라고 생각된다.

반면 program2와 program3를 비교해 보면 가설 2에 부합하는 결과라고 생각된다. **멀티 프로세싱의 cost등에 의해 멀티 스레딩에 비해 같은 조건에서 더 느린 결과를 기록**했으며, num process/ thread가 늘어날 수록 소요시간의 증가 폭은 멀티프로세싱이 멀티스레드에 비해 컸다.

인풋 사이즈가 5000이나 커졌음에도 불구하고, 가설 1에는 전혀 부합하지 않는 결과가 발생했다.

- **inputs6.txt의 결과(input size 25000)**

	10	50	150	255	2000	4096
program1	17ms	-	-	-	-	-
program2	50ms	190ms	291ms	733ms	-	-
program3	11ms	115ms	163ms	323ms	2273ms	2769ms

여전히 program1의 수행속도가 가장 빨라 가설 3에 반하는 결과였다. 아무래도 멀티스레딩/프로세싱의 병렬 효과를 실감하기 위해서는 더더욱 **큰 input size, 혹은 더욱 복잡한 연산의 수행**이 필요하다고 생각된다.

proram2와 program3의 경우 모두 이전의 결과들과 같은 흐름의 결과였다. 다만 인풋 사이즈가 커지니 전체적인 소요시간이 늘어났으며 소요시간의 증가 폭도 이전 inputs4.txt에 비해 늘어났음을 확인했다.

- **더 큰 사이즈의 inputs들 결과**

  가설 1에 부합하는 결과를 보기 위해 input size를 백만까지 늘려서 실험했다. 또한 process/thread의 생성되는 갯수를 2개로 고정하여 multi processing, multi threading 기법 사용시 발생하는 cost를 최대한 줄여 실험했다.

  <발생한 문제>

  inputsize가 큰 테스트를 수행할때 program2의 동작이 수행되지 않고 무한 루프에 빠지는 결과가 발생했다.

  program2는 구현 과정 중, multiprocessing을 수행 할 때, IPC pipe통신을 할때 입출력을 'string' 자료형, char*자료형을 이용하도록 구현했다. 때문에서인지, 큰 input size의 input을 사용하면, 문자열에 너무 많은 크기의 정보를 담게 되어 시스템 메모리 제한을 초과하게 되는 문제가 발생했다고 예측된다. 혹은, pipe통신에서 메모리 초과가 발생해 pipe에 입출력이 전달이 안되어 무한히 대기하는 상태가 된 것이라 생각된다. 때문에 아쉽지만 program2, 멀티프로세싱은 큰 인풋사이즈와 total_num_process 2의 테스트 결과를 측정할 수 없었다.

결과는 가설 1과 부합하는 결과였다.

(input size 100만, total_num_thread 2)

program1	425ms
program3	241ms

인풋 사이즈가 충분히 크면 **멀티스레드의 병렬 연산의 효과가 멀티스레드의 오버헤드 cost 에 의한 성능 저하를 초과하는 것**을 확인할 수 있었다.