

2021-1 운영체제 과제 #2

응용통계학과 2017122063 정재은

1. 사전 조사 보고서

A. 프로세스와 스레드

a. 프로세스와 스레드의 차이점

프로세스란 수행 중인 프로그램의 인스턴스이다. 프로그램은 하드디스크와 같은 저장소에 저장되는 정적인 객체인 반면 프로세스는 메인 메모리에 올라와서 CPU 스케줄링의 대상이 되는 동적인 객체이다. 프로세스는 `image`, `process context`로 구성된다. 모든 프로세스는 부모 프로세스로부터 생성된다. 리눅스 및 UNIX 계열의 운영체제에서는 `fork()` 시스템 콜을 이용하여 새로운 프로세스를 생성하는데, 부모 프로세스의 `image`, `program context`를 복사하여 자식 프로세스를 만든다.

스레드란 어떠한 프로그램 내에서, 특히 프로세스 내에서 실행되는 흐름의 단위를 뜻한다.¹ 따라서 프로세스는 여러 스레드가 실행되는 하나의 컨테이너로도 볼 수 있다. 스레드는 하나의 프로세스에 종속되지만 프로세스는 여러 개의 스레드를 가질 수 있다. 하나의 프로세스안에서 생성된 스레드들은 동일한 이미지 안에서 각자의 `logical control flow`와 `user stack`을 가지고 있으며 `code`, `data`, `kernel context`를 공유한다.

프로세스와 스레드 모두 각자의 `logical control flow`를 가지며 스케줄링의 대상이 된다는 점이 유사하다. (각자 독립적인 `pid/tid`를 가지고 있다.) 하지만 스레드는 하나의 이미지 안에서 코드, 데이터 등을 공유하지만 프로세스는 그렇지 않고 독립적인 메모리 공간 안에서 각자의 이미지를 가지므로 생성, (프로세스) 스위칭, 종료 모두 스레드에 비해서 OS의 작업량이 많다.

결국 스레드는 프로세스에 비해 경제적이며(응답 시간이 적게 걸리며 자원, 메모리가 낭비되지 않는다.) 단일 이미지 안에 존재하므로 커널 모드를 거치지 않고도 스레드간 자원 공유가 가능하다. 하지만 서로 다른 데이터가 동시에 전역 변수에 접근하게 된다면 전역 변수의 `integrity`가 보장되지 않는다는 동기화 이슈가 존재한다.

b. 프로세스와 스레드에서의 리눅스에서의 구조 및 구현

리눅스에서 `task_struct`라는 구조체를 이용하여 프로세스와 스레드를 구현한다.

`task_struct`는 PCB(Process Control Block) 또는 Process Descriptor로 불리는 자료구조의 일종이다.

¹ [https://ko.wikipedia.org/wiki/스레드_\(컴퓨팅\)](https://ko.wikipedia.org/wiki/스레드_(컴퓨팅))

`task_struct` 안에는 `image`, `process context`가 포함되어 있다. `image` 안에는 컴파일된 코드(`code/machine instructions`), 글로벌 변수(`global data`), `stack`, `heap`이 있다. `process context`는 `program context`, `kernel context`로 다시 한 번 나뉜다. 이 중 `program context`는 PC(Program Counter), SP(Stack Pointer), 레지스터들의 상태 등 특정 시점에서의 CPU 스냅샷을 의미한다. `Kernel context`는 커널이 프로세스를 관리하기 위한 여러 메타 정보를 가지고 있다. 스레드는 이 중에서 `user stack`, `program context`만을 담고 있는 프로세스의 구성요소이다.

리눅스에서는 `fork()` 시스템 콜을 호출하여 새로운 프로세스를 만든다. `fork()`를 호출할 때는 부모 프로세스의 `image`, `program context`를 복사하여 자식 프로세스를 만드는데 이때 포인터를 복사하는 `shallow-copy` 형식이 아니라 `deep-copy`를 통해 복사를 진행한다.

새로운 스레드를 만들기 위해 리눅스에서는 `clone()` 시스템 콜을 이용한다. `fork()`와 비슷하지만 `image`, `program context`를 모두 복사하는 `fork()`와 다르게 `clone()`은 공유할 데이터를 함수 인자로 넘겨주어 부모와 자식이 공유할 데이터를 지정한다. 이때 스레드는 단일 이미지 안에 존재하는 `code`, `data`, `heap`, `kernel context`를 공유하며 `program context`와 `user stack`만 따로 가진다. 생성된 스레드는 프로세스와 동일하게 CPU 스케줄링의 대상이 된다.

B. 멀티 프로세스와 멀티 스레딩

a. 멀티 프로세스와 멀티 스레딩의 개념 및 구현 방법

멀티 프로세스란 동시에 여러 프로세스를 병렬적으로 실행하여 작업을 하는 것을 뜻한다. 리눅스에서 멀티 프로세스는 앞서 언급한 `fork()` 시스템 콜을 통해 이루어진다. 부모 프로세스에서 `fork()`를 호출하면 자식 프로세스 생성이 성공했을 때는 자식 프로세서의 `pid`를, 실패했을 때는 `-1`을 반환한다. 자식 프로세스에서 `fork()`의 반환값은 `0`이 된다. 자식 프로세스들은 병렬적으로 작업을 처리하고 부모 프로세스는 `wait`, `waitpid` 같은 함수를 통해 자식 프로세스가 모두 끝나기를 기다렸다가 남은 일을 수행한다. 이때 부모와 자식 프로세스는 별개의 메모리 공간에 존재하므로 프로세스간 통신을 위해서는 `pipe`, `socket`과 같은 IPC(Interprocess communication)를 이용해야 한다.

멀티 스레딩이란 하나의 프로세스를 다수의 실행 단위로 구분하여 자원을 공유하고 불필요한 자원의 생성과 관리를 최소한으로 하여 수행 능력을 향상시키는 것이다.² 한 프로세스로부터 여러 개의 스레드가 생성되고, 생성된 스레드들은 병렬적으로 작업을 수행한다. 멀티 스레딩을 위해 POSIX 표준에 해당하는 `pthread` 라이브러리가 존재하는데, 구현은 각 OS마다 상이하다.

² <https://asfirstalways.tistory.com/340>

Pthread API에서는 `pthread_create` 함수를 통해 스레드를 생성하는데 리눅스에서는 내부적으로 앞서 언급한 `clone()` 시스템 콜을 이용한다. 스레드들은 하나의 이미지 안에 존재하고 동일한 코드를 가지므로 `pthread_create` 함수에는 작업을 수행할 함수 포인터를 넣어준다. 이때 전역 변수 동기화 이슈를 위해 `pthread_mutex_lock`, `pthread_mutex_unlock` 같은 함수를 사용한다. 스레드를 생성한 부모 프로세스에서는 `pthread_join` 함수를 통해 스레드들의 작업이 끝나기를 기다리고 각 스레드는 `pthread_exit` 함수를 이용하여 작업이 끝남을 알릴 수 있다.

C. exec 계열의 시스템 콜

a. exec 계열의 시스템 콜 종류³

- `int execl(const char *path, const char *arg0, ..., const char *argn, (char *)0)`: path에 지정한 경로명의 파일을 실행하며 arg0~argn을 인자로 전달한다. 관례적으로 arg0에는 실행 파일명을 지정한다.
- `int execv(const char *path, char *const argv[])`: execl과 유사하지만 배열 형태로 인자를 전달한다. argv 배열의 마지막에는 NULL을 저장해야 한다.
- `int execle(const char *path, const char *arg0, ..., const char *argn, (char *)0, char *const envp[])`: path에 지정한 경로명의 파일을 실행하며 arg0~argn을 인자로 전달한다. envp에는 새로운 환경 변수를 지정하며 배열의 마지막에는 NULL이 들어가야 한다.
- `int execve(const char *path, char *const argv[], char *const envp[])`: execle과 비슷하지만 인자들이 배열 형태로 들어간다. 마찬가지로 envp 배열의 마지막에는 NULL이 들어가야 한다.
- `int execlp(const char *file, const char *arg0, ..., const char *argn, (char *)0)`: file에 지정한 파일을 실행하며 arg0~argn을 인자로 전달한다. 파일은 환경 변수 PATH(ex: /usr/bin)에 정의된 경로에서 탐색한다.
- `int execvp(const char *file, char *const argv[])`: execlp와 유사하지만 인자들이 배열 형태로 들어간다. 마찬가지로 배열의 마지막에는 NULL이 들어가야 한다.

b. exec 계열의 시스템 콜이 리눅스에서 구현되고 동작하는 방법

`exec()` 계열의 시스템 콜은 `fork()`를 통해 생성된 새로운 프로세스의 실행을 위한 초기화를 담당한다. `exec()` 계열의 시스템 콜을 호출한 프로세스는 커널 모드로 가 아예 새로운 프로그램으로 바뀌고(`code, data, heap and stack`가 모두 변화한다.) 새로운 `main()` 함수를 호출한다. 다음과 같은 과정을 통해 `exec()` 계열의 시스템 콜은 내부적으로 모두 `execve()`를 호출한다.

³ <https://bbolmin.tistory.com/35>

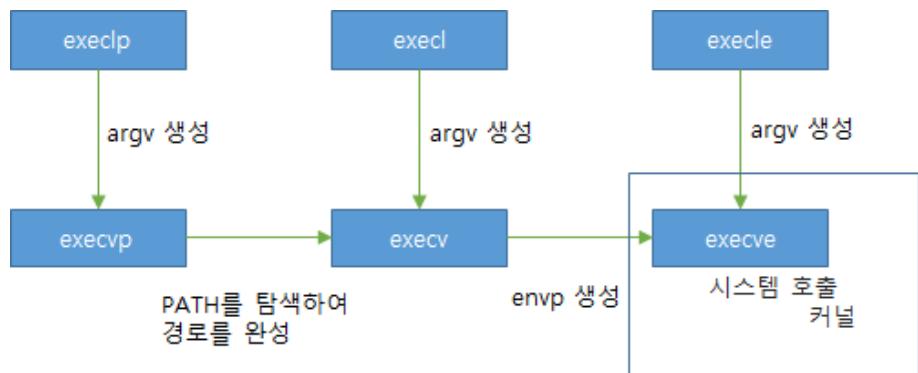


Figure 1: <https://ehpub.co.kr/tag/exec-함수군/>

D. 참조 문헌

- 연세대학교 2021년 1학기 차호정 교수님 운영체제 강의자료
- [https://ko.wikipedia.org/wiki/스레드_\(컴퓨팅\)](https://ko.wikipedia.org/wiki/스레드_(컴퓨팅))
- <https://asfirstalways.tistory.com/340>
- <https://bbolmin.tistory.com/35>
- <https://ehpub.co.kr/tag/exec-함수군/>

2. 프로그래밍 수행 결과 보고서

A. 작성한 프로그램의 동작 과정과 구현 방법

a. 실습 과제1 프로그램: MiniShell

<헤더 파일 설명>

실습 과제 1을 수행하기 위해 miniShell.h, miniShell.cpp 파일을 만들어주었다.

miniShell.h에는 miniShell.cpp을 위한 여러 상수와 구조체, 함수를 선언했다.

```
// maximum length of user command
#define MAX_LEN 200
// maximum number of args in user command
#define ARGS_LEN 64
```

먼저, 사용자로부터 입력 받을 한 줄의 명령어의 최대 길이를 200으로 저장해 MAX_LEN으로 명명했다. 또한, 사용자의 명령어를 파싱했을 때 인자의 최대 개수는 64로 설정하고 ARGS_LEN으로 지정해주었다.

```
// represents user command
typedef struct _Command {
    // commands as it is
    char msg[MAX_LEN];
    // number of args in parsed command
    int argc;
    // parsed args
    char *argv[ARGS_LEN];
} Command;
```

보다 편리한 처리를 위해 사용자의 명령어를 나타내는 구조체를 하나 선언했다. msg는 입력 받은 한 줄의 명령어 그 자체이고, argv는 명령어를 파싱한 문자열을 담고 있는 배열이다. 예를 들어, ls -alh를 입력하면 argv는 {"ls", "-alh"}가 된다. argc는 argv의 크기인 인자의 개수를 뜻한다.

```
void greeting();
void tokenize(char *msg, Command *command);
void cd_handler(Command *command, bool bg);
bool is_bg(Command *command);
int is_redirect(Command *command);
void redirect_handler(int redirect, Command *command);
```

다음은 miniShell.cpp 파일에서 사용할 함수의 원형을 선언했다. miniShell.cpp에서 사용할 함수는 6 가지이다.

<함수 설명>

- void greeting();

이 함수는 쉘을 실행했을 때 [HH:MM:SS] 'username'@'pwd'\$ 형식으로 출력하는 역할을 한다. 자세한 설명은 `miniShell.cpp`에 주석으로 달아놓았다.

- void tokenize(char *msg, Command *command);

사용자의 입력을 파싱하여 `command`의 `argv`를 만드는 함수이다. C++ 표준 라이브러리인 `<stack>`을 사용하여 문자열을 파싱하였는데, 문자열에 ‘ ’ 또는 “이 포함된 경우를 생각해주었다. 예를 들어, `cd 'd1 d2'`는 {"cd", "d1 d2"}로 파싱되고 `cd d1 d2`는 {"cd", "d1", "d2"}로 나눠진다. 나눠진 인자(토큰)들을 동적 배열을 통해 `command`의 `argv`에 저장해주었고 `argv`의 마지막인자는 항상 `NULL`이 되도록 했다. 구체적인 작동 방식은 주석으로 달아놓았다.

- void cd_handler(Command *command, bool bg);

사용자가 `cd`를 입력했을 때 핸들링하는 함수이다. 일반적인 다른 명령어처럼 `fork()`, `exec()`을 통해서 `cd`를 실행하면 자식 프로세스에서 작업 디렉토리가 바뀌지만 부모 프로세스인 `miniShell`에는 영향을 주지 못한다. 따라서 자식 프로세스를 만들지 않고 `cd`를 처리하는 함수를 따로 만들어주었다. C/C++ 표준 라이브러리 `<unistd.h>`의 `chdir()`함수를 이용하였다.

```
// if background process, you have to create a new process
if(bg) {
    pid_t pid = fork();
    if(pid < 0) {
        cout << "Fork failed.\n";
    } else if(pid == 0) {
        execvp(command->argv[0], command->argv);
        exit(0);
    }
    return;
}
```

그러나 명령어 끝에 백그라운드 명령을 나타내는 &가 붙으면 새로운 프로세스를 `fork()`를 통해서 새로운 프로세스를 만들어주었다. 백그라운드 실행이므로 부모 프로세스와 별개로 실행되고, 부모 프로세스는 자식 프로세스의 종료와 관계없이 동작한다.

```
// if user entered only "cd"
if(command->argc == 1) {
    if(getenv("HOME") != NULL) {
        chdir(getenv("HOME"));
    }
} else if(command->argc == 2) {
    if(chdir(command->argv[1]) < 0) {
        cout << "cd: no such file or directory: " << command->argv[1] << '\n';
    }
} else {
    cout << "cd: too many arguments" << '\n';
}
```

사용자의 명령어 개수가 하나라면 cd 만 입력했다는 뜻이므로 HOME 디렉토리로 이동해주고, 명령어 개수가 두 개라면 두 번째 인자에 알맞는 디렉토리로 이동하게 해주었다. 사용자의 명령어 개수가 3 개 이상일 때는 인자의 개수가 너무 많다는 의미로 “cd: too many arguments” 출력하게 했다. 또한 cd 작업이 실패했을 때는 “cd: no such file or directory: ”를 출력하여 올바르지 않은 파일/디렉토리임을 전달하도록 했다.

- `bool is_bg(Command *command);`

백그라운드로 실행되어야 하는 명령어인지를 알려주는 함수이다. Command 구조체 포인터를 인자로 받아 마지막 문자열이 ‘&’인지 확인한다. 만약 마지막 문자열이 ‘&’ 이라면 백그라운드 실행이라는 뜻이므로 해당 문자열의 메모리를 해제하고 인자 개수를 줄인 다음 마지막 문자열로 NULL을 넣어준다.

- `int is_redirect(Command *command);`

출력/입력 리다이렉션을 확인하는 함수이다. Command의 argv에 ‘<’나 ‘>’가 포함되어 있는지를 확인하고 포함되어 있다면 해당 인덱스를 반환한다. 만약 ‘<’ 또는 ‘>’가 포함되어 있지 않다면 출력/입력 리다이렉션을 사용하지 않는다는 뜻이므로 -1 을 반환한다.

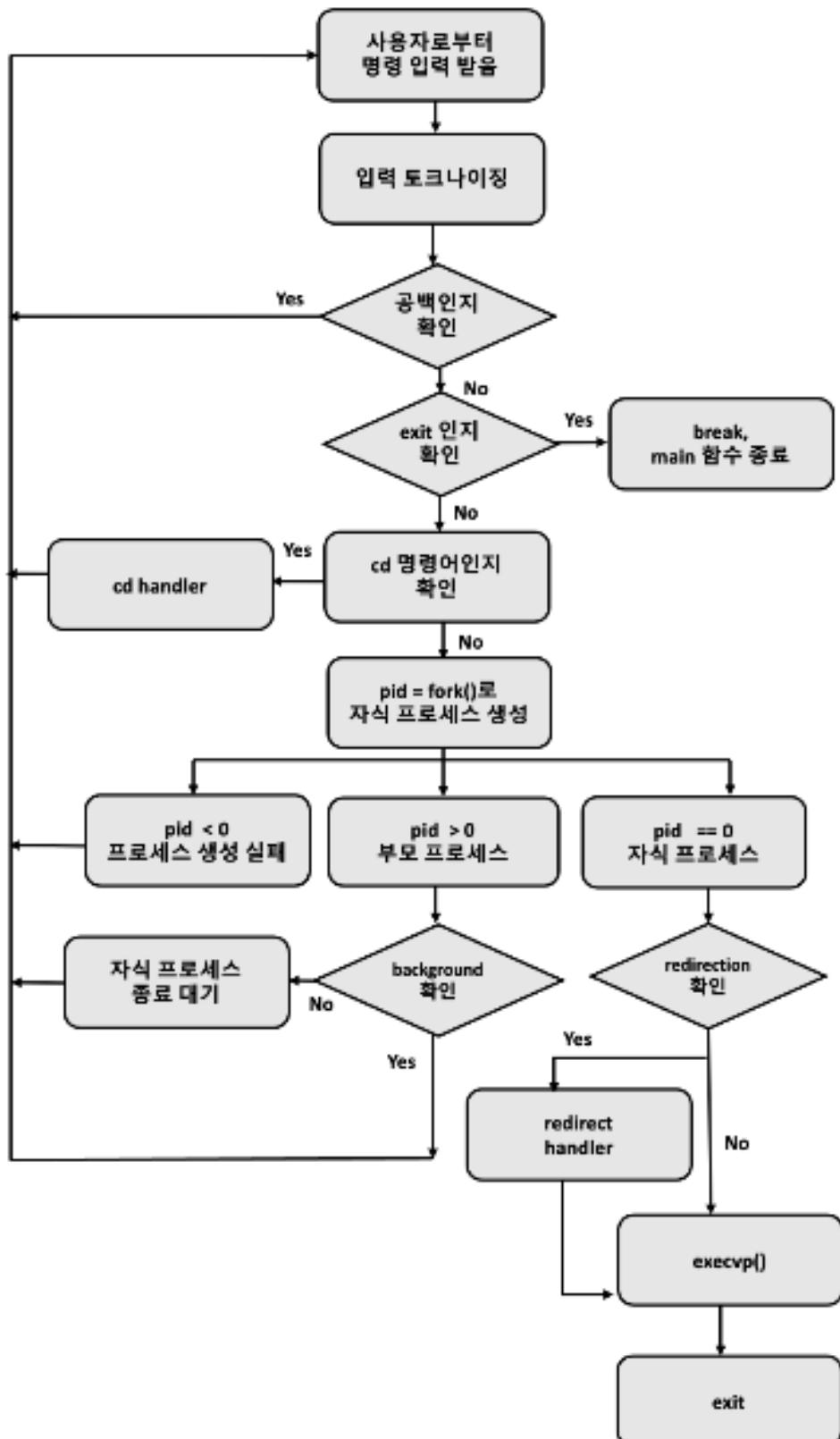
- `void redirect_handler(int redirect, Command *command);`

출력/입력 리다이렉션을 처리해주는 함수이다. 인자로는 command->argv에서 ‘<’, ‘>’의 인덱스를 의미하는 redirect와 Command 구조체 포인터 command를 넣어준다. ‘<’, ‘>’ 뒤에는 출력/입력 리다이렉션 파일 이름이 온다고 가정하였다. 따라서 open 함수를 통해 command->argv[redirect+1]를 열었고 dup2를 통해 stdin/stdout을 대체하였다. 출력/입력 리다이렉션 파일 이름 뒤에 오는 인자/문자열들은 불필요한 것으로 간주하여 동적 할당한 메모리를 해제하고 command->argc를 그만큼 감소시켜주었다. 아래는 출력 리다이렉션을 처리한 코드 부분이다.

```
// stdout redirection
if(ch == '>') {
    // open file after '>'
    fid = open(command->argv[redirect+1], O_WRONLY | O_TRUNC | O_CREAT,
               S_IRUSR | S_IWGRP | S_IWGRP | S_IWUSR);
    // exit if the file can't be opened
    if(fid < 0) {
        exit(1);
    }
    // replace stdout to the newly opened file
    dup2(fid, STDOUT_FILENO);
    // discard the args after '>'
    for(int i=redirect;i<command->argc;i++){
        free(command->argv[i]);
    }
    // reduce the number of args
    command->argc -= (command->argc - redirect);
    command->argv[command->argc] = NULL;
    // close unused file descriptor
    close(fid);
    // execute child process
    execvp(command->argv[0], command->argv);
```

<메인 코드 설명>

miniShell.cpp는 다음과 같은 흐름으로 진행된다.



b. 실습 과제2 프로그램

<헤더 파일 설명>

programs.h 에서는 program1, program2, program3에서 사용할 변수, 함수, 구조체 등을 선언해주었다.

```
/* global vars for program1, program2 and program3 */
// number of data
int n = 0;
// input data
vector<int> v;
// sorted data
vector<int> sorted;
```

먼저, program1, program2, program3에서 모두 공통으로 쓰일 n, v, sorted를 선언해주었다. n은 데이터의 개수, v는 입력 받은 배열, sorted는 v의 정렬된 결과를 담고 있는 배열이다. v, sorted는 모두 C++의 STL에 속하는 vector 라이브러리를 사용했다.

```
/* program2 */
// number of processes
int total_process_num = 0;
// input temporary file name
const string INPUT = "HW2_MYINPUT";
// output temporary file name
const string OUTPUT = "HW2_MYOUTPUT";
/* program2, program3: used for merging partially sorted array */
typedef struct _Index {
    int start;
    int mid;
    int end;
} Index;
```

다음은 program2에서 사용할 변수와 구조체를 선언했다. 프로세스의 개수를 나타내는 total_process_num과 임시 파일 생성을 위한 INPUT, OUTPUT 문자열 변수를 선언했고 병합 정렬의 인자로 쓰이는 start, mid, end를 담고 있는 Index 구조체 변수를 선언해주었다.

```
/* program3 */
// number of threads
int total_thread_num = 0;
// mutex variable for synchronization
pthread_mutex_t mutex_lock;
// (void *)arg for pthread_create
typedef struct _Args {
    int start;
    int end;
} Args;
// function for pthread_create()
void *merge_sort(void *args);
```

그리고 program3를 위한 변수, 구조체 및 함수를 선언해주었다. total_thread_num은 스레드 개수를 나타내며 mutex_lock은 전역 변수 동기화를 위한 뮤텍스 변수이다. pthread_create에 인자로 들어가는 함수는 void형 포인터를 인자로 받고 void형 포인트를 리턴하는 함수이므로 Args, (void *) merge_sort(void *args)를 선언해주었다. Args 포인터 변수는 (void*) merge_sort(void *args) 함수의 인자로 들어간다. (void*) merge_sort(void *args) 함수의 내용은는 program3.cpp에 있다. Args는 Index와 다르게 start, end 만을 멤버로 갖는다.

마지막으로는 program1, program2, program3에서 모두 쓰일 merge 함수와 merge_sort 함수를 선언해주었다. 해당 함수에 대한 설명은 주석으로 달아 놓았으며 기본적인 병합 정렬 알고리즘에 해당하므로 설명을 생략하겠다. 이때 일반적인 병합 정렬 알고리즘과 다르게 merge 함수에서는 bool mutex를 인자로 받는데, program3에서 전역 데이터를 공유하기 때문이다. 만약 mutex == true라면 merge 함수를 실행하기 전에 mutex를 lock 해주고 merge 함수가 종료될 때 lock을 풀어준다. program1, program2에서는 mutex를 사용할 일이 없으므로 false로 설정한다.

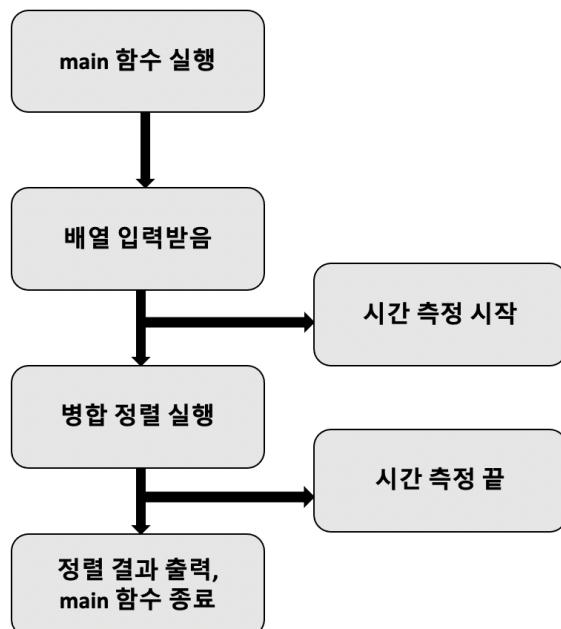
```
// mutex: true => locking
if(mutex) {
    pthread_mutex_lock(&mutex_lock);
}

if(start < end) {
    int mid = (start + end) / 2;
    // left merge sort
    merge_sort(start, mid);
    // right merge sort
    merge_sort(mid+1, end);
    // merge left and right
    // by default, mutex is false(program1, program2)
    merge(start, mid, end, false);
}
```

```
// unlock mutex
if(mutex) {
    pthread_mutex_unlock(&mutex_lock);
}
```

<Program 1>

프로그램 1은 기본적인 병합 정렬을 구현한 프로그램이며 다음과 같은 흐름으로 진행된다.



<Program 2>

프로그램 2는 멀티 프로세스를 이용하여 병합 정렬을 구현한 프로그램이다.

```
// vector that stores the size of divided data before calling merge sort
vector<int> prev;
prev.push_back(n);
// vector that stores the size of divided data after calling merge sort
/**
 * ex) prev: {8}, now: {4, 4} => prev: {4, 4}, now: {2, 2, 2, 2}
 */
vector<int> now(prev);
while(true) {
    // if the number of data segments is greater than or equal to process number
    if(now.size() >= total_process_num) {
        break;
    }
}
```

먼저, 과제 요구 사항에 따라 병합 정렬을 재귀적으로 진행하던 중에 쪼개진 데이터의 수가 `total_process_num` 이상이면 멀티 프로세스를 이용하여 병합 정렬을 진행해야하므로 병합 정렬 과정을 시뮬레이션 해보았다. `prev`, `now` 벡터를 통해 병합 정렬 과정을 시뮬레이션할 수 있었는데, `prev` 벡터는 처음에는 `n` 만을 가지고 있으며 `now` 벡터는 `prev` 벡터에서 병합 정렬 함수를 한 번 더 호출했을 때 나눠진 데이터의 개수들을 가지고 있다. `while`문을 통해 병합 정렬에서 데이터가 나뉘는 과정을 시뮬레이션 하다가 `now` 벡터의 수가 `total_process_num` 이상이면 `while`문을 빠져나온다.

과제 스펙에 나와있는 예시는 다음과 같은 흐름으로 진행될 것이다.

```
prev: {16}, now: {8, 8}
prev: {8, 8}, now: {4, 4, 4, 4} => while 문 빠져나온다.
```

```
// fix total_process_num to now.size()
total_process_num = now.size();

/* start indices of divided data
 * ex) now: {2, 2, 2, 2} => offset: {0, 2, 4, 6} */
vector<int> offset;
int tmp = 0;
for(int i=0;i<now.size();i++) {
    offset.push_back(tmp);
    tmp += now[i];
}
```

이제 앞에서 구한 `now` 벡터의 사이즈가 새로운 `total_process_num`이 된다. 앞서 언급한 예시에서는 `total_process_num`이 4가 된다. 이제 `offset` 벡터를 새로 생성하고 나눠진 데이터의 원래 배열에서의 시작 인덱스를 저장한다. 예시와 같은 경우에는 `{0, 4, 8, 12}`가 된다.

```

// save temporary files that store the divided data segment
for(int i=0;i<total_process_num;i++) {
    ofstream file;
    string file_name = INPUT + to_string(i);
    file.open(file_name.c_str());
    // number of data segment
    file << now[i] << '\n';
    // save data segment
    for(int j=0;j<now[i];j++) {
        file << v[offset[i]+j] << ' ';
    }
    file.close();
}

```

이제 now, offset 벡터를 통해 배열의 데이터를 나눠서 임시 파일로 저장한다.

```

// array for containing child processes
pid_t pids[total_process_num];
// start performance measurement, clock(): ms
clock_t start = clock();

```

새롭게 생성될 자식 프로세스들의 pid를 담은 배열을 생성하고, 시간 측정을 시작한다.

```

for(int i=0;i<total_process_num;i++) {
    pids[i] = fork();
    if(pids[i] < 0) {
        cout << "Fork failed.\n";
        exit(1);
    } else if(pids[i] == 0) {
        // temporary input file made just before
        string in_file = INPUT + to_string(i);
        // temporary output file to store sorted data segment
        string out_file = OUTPUT + to_string(i);
        int in, out;
        in = open(in_file.c_str(), O_RDONLY);
        out = open(out_file.c_str(), O_WRONLY | O_TRUNC | O_CREAT,
                   S_IRUSR | S_IRGRP | S_IWGRP | S_IWUSR);
        // replace stdin, stdout
        dup2(in, STDIN_FILENO);
        dup2(out, STDOUT_FILENO);
        // close unused files
        close(in);
        close(out);
        // remove temporary input file
        remove(in_file.c_str());
        char *args[] = {(char *) "./program1", NULL};
        // execute program1
        execvp("./program1", args);
        exit(0);
    }
}

```

total_process_num 만큼 fork() 함수를 이용하여 자식 프로세스를 생성한다. 자식 프로세스의 stdin은 방금 생성한 임시 파일이며 stdout을 위한 임시 파일도 만들어준다. dup2() 함수를 통해 stdin, stdout을 해당 파일들로 바꾸고 program1을 실행한다.

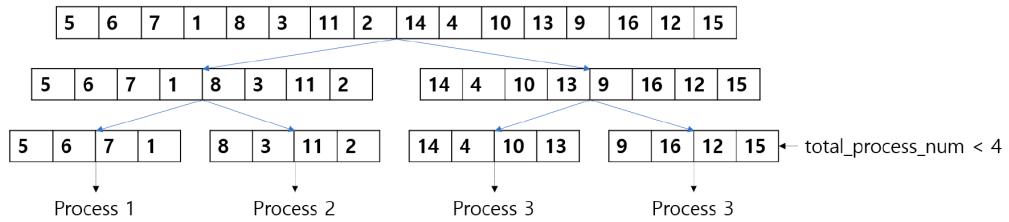
```

// wait all the child processes to be done
for(int i=0;i<total_process_num;i++) {
    waitpid(pids[i], NULL, 0);
}

// take partially sorted data
int idx = 0;
for(int i=0;i<total_process_num;i++) {
    // temporary output file that stores sorted data segment
    string out_file = OUTPUT + to_string(i);
    ifstream output;
    output.open(out_file);
    for(int j=0;j<now[i];j++) {
        int tmp;
        output >> tmp;
        v[idx] = tmp;
        sorted[idx] = tmp;
        idx++;
    }
    output.close();
    // remove temporary output file
    remove(out_file.c_str());
}

```

부모 프로세스는 `waitpid()`를 통해 자식 프로세스들이 모두 종료되길 기다린다. 자식 프로세스가 모두 종료된 후에는 자식 프로세스에서 생성한 아웃풋 파일을 `v`, `sorted` 벡터에 복사해준다. 따라서 `v`, `sorted` 벡터는 모두 부분적으로 정렬된 배열을 가지고 있다.



과제에 나온 예시에서는 `v`, `sorted`는 모두 {1, 5, 6, 7, 2, 3, 8, 11, 4, 10, 13, 14, 9, 12, 15, 16}로 저장되어 있을 것이다. (4개씩 부분 정렬됨)

부모 프로세스에서는 이제 자식 프로세스에서 수합한 데이터를 새로 `merge` 하는 과정을 거쳐야 한다. 해당 과정을 위해 헤더 파일에서 정의한 `Index` 구조체 포인터 변수를 가지고 있는 큐를 선언해주었다.

```

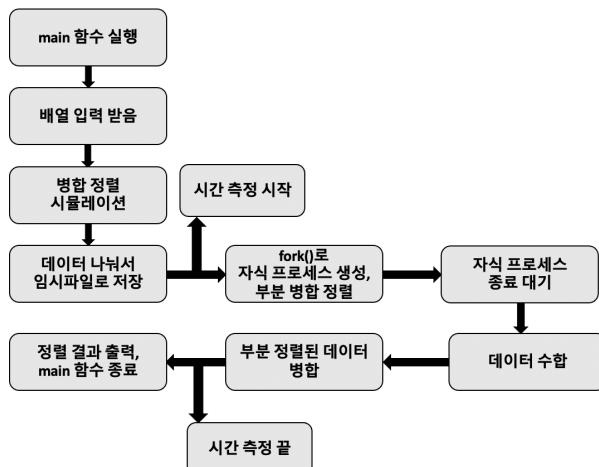
/* By using offset & now, prepare to combine partially sorted data.
   Pushing struct Index that has arguments of merge() function to queue. */
queue<Index *> q;
for(int i=0;i<total_process_num-1;i+=2) {
    int start = offset[i];
    int end = offset[i+1]+now[i+1]-1;
    int mid = start+now[i]-1;
    Index *idx = new Index();
    idx->start = start;
    idx->mid = mid;
    idx->end = end;
    q.push(idx);
}

```

자식 프로세스에서 수합된 데이터들을 앞에서 만들어준 `now`, `offset` 벡터를 활용하여 데이터들을 두 부분씩 합쳐준다. 예를 들어, 4개씩 부분 정렬된 배열 `{1, 5, 6, 7, 2, 3, 8, 11, 4, 10, 13, 14, 9, 12, 15, 16}`의 `offset`은 `{0, 4, 8, 12}`이고 `now`는 `{4, 4, 4, 4}`이다. 이때 `{1, 5, 6, 7}`과 `{2, 3, 8, 11}`을 merge 하기 위해서 `start=offset[i]=0`, `end=offset[i+1]+now[i+1]-1=7`, `mid=start+now[i]-1=3`이 된다. 이러한 정보를 새로 생성한 `Index` 구조체 포인터 변수에 저장하고 큐에 차례대로 삽입해준다.

```
// merge until size of queue == 1, which means start=0, end=n-1
while(!q.empty()) {
    Index *idxs1 = q.front();
    q.pop();
    merge(idxs1->start, idxs1->mid, idxs1->end, false);
    // merge complete: start=0, end=n-1
    if(q.empty()) {
        break;
    }
    Index *idxs2 = q.front();
    q.pop();
    merge(idxs2->start, idxs2->mid, idxs2->end, false);
    int start = idxs1->start;
    int end = idxs2->end;
    int mid = idxs1->end;
    // now push merged data to queue again => backward of merge sort
    Index *idx = new Index();
    idx->start = start;
    idx->mid = mid;
    idx->end = end;
    q.push(idx);
}
```

이제 큐에 있는 `Index` 구조체 포인터 변수의 정보를 이용하여 병합 과정을 진행한다. 여기서 주의할 점은 두 번씩 병합 과정을 진행한 후 먼저 병합한 데이터와 나중에 병합한 데이터가 나중에 병합될 수 있도록 하기 위해 큐에 새로운 `Index` 구조체 포인터 변수를 삽입한다는 것이다. 이때 큐가 비어 있다면 병합 과정이 완료된 것이므로 `while` 문을 빠져나오면 된다. 일련의 과정은 병합 정렬에서 재귀 함수 호출이 끝나고 `merge` 함수를 재귀적으로 수행되는 것을 구현한 것이다. 전체적인 흐름은 다음과 같다.



<Program 3>

프로그램 3은 멀티 스레드를 이용하여 병합 정렬을 구현한 프로그램이다. 헤더 파일에서 정의한 `(void *) merge_sort(void *args)`의 코드는 다음과 같다.

```
void *merge_sort(void *args) {
    Args *myargs = (Args *) args;
    int start = myargs->start;
    int end = myargs->end;
    if(start < end) {
        int mid = (start + end) / 2;
        Args *newargs = new Args();
        // left merge sort
        newargs->start = start;
        newargs->end = mid;
        merge_sort(newargs);
        // right merge sort
        newargs->start = mid+1;
        newargs->end = end;
        merge_sort(newargs);
        /* merge left and right; set mutex = true since
         | threads share global data */
        merge(start, mid, end, true);
    }

    return NULL;
}
```

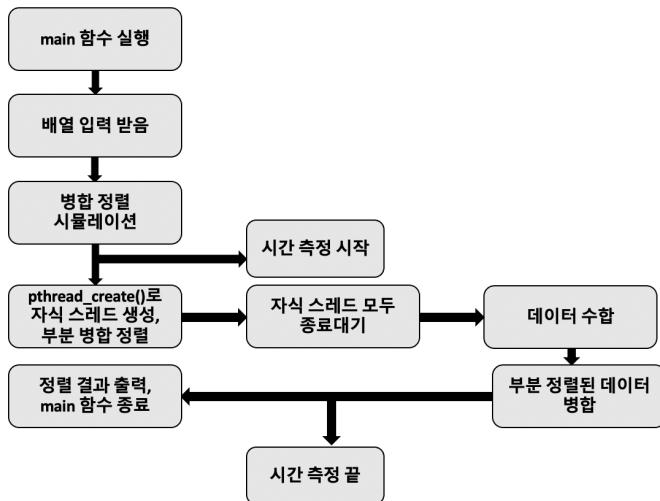
`programs.h`에 있는 `merge_sort` 함수와 다른 점은 `int start`, `int end`를 인자로 받지 않고 `start`, `end`를 저장하고 있는 `Args` 구조체 변수의 주소값을 받는다는 점이다. 나머지 부분은 기존의 `merge_sort` 함수와 같으나 `merge` 함수에 들어가는 `mutex`를 `true`로 설정한다는 점에서 상이하다. (물론 멀티 스레드를 생성해서 병합 정렬하는 과정을 생각한다면 스레드별로 정렬하는 데이터 부분이 다르므로 사실상 필요가 없지만 일반적인 멀티 스레드 방식에서는 동기화 이슈가 문제가 되므로 이번 과제에서는 이렇게 수행하였다.)

```
// array for containing threads
pthread_t tids[total_thread_num];
// init mutex to NULL
pthread_mutex_init(&mutex_lock, NULL);
// start performance measurement, clock(): ms
clock_t start = clock();
for(int i=0;i<total_thread_num;i++) {
    int start = offset[i];
    int end = start+now[i]-1;
    Args *args= new Args();
    args->start = start;
    args->end = end;
    pthread_create(&(tids[i]), NULL, merge_sort, args);
}

// wait all the threads to be done
for(int i=0;i<total_thread_num;i++) {
    pthread_join(tids[i], NULL);
}
```

프로그램 3은 프로그램 2와 매우 유사하지만 전역 데이터를 공유하므로 프로그램 2처럼 임시 파일을 생성할 필요가 없다. 프로그램 2에서와 같은 방법으로 `now`, `offset` 벡터를 사용하여 데이터를 나눠주고 `pthread_create` 함수를 이용하여 스레드를 생성하여 각 스레드가 `merge_sort`를 함수를 수행하도록 하였다. 부모 프로세스에서는 `pthread_join` 함수를 통해 모든 자식 스레드들이 끝나기를 기다린다.

모든 자식 스레드가 종료되었다면 글로벌 변수인 `v`, `sorted`는 부분적으로 정렬된 데이터를 가지고 있을 것이다. 이후엔 프로그램 2에서 한 것과 같은 방식으로 부분 정렬된 데이터를 `merge` 해주고 출력한다. 전체적인 흐름은 다음과 같다.



B. Makefile

다음은 Makefile의 캡처본이다.

```

CC = g++
CXXFLAGS = -std=c++11

all: miniShell program1 program2 program3
miniShell: miniShell.h miniShell.cpp
        $(CC) $(CXXFLAGS) miniShell.cpp -o miniShell
program1: programs.h program1.cpp
        $(CC) $(CXXFLAGS) program1.cpp -o program1
program2: programs.h program2.cpp
        $(CC) $(CXXFLAGS) program2.cpp -o program2
program3: programs.h program3.cpp
        $(CC) $(CXXFLAGS) program3.cpp -o program3 -lpthread
clean:
        rm -rf miniShell
        rm -rf program1
        rm -rf program2
        rm -rf program3
  
```

`g++` 컴파일러를 사용해서 C++11 버전으로 작성한 파일들을 컴파일하게 해주었다. 또한 이때 리눅스에서 `<pthread.h>`의 함수를 사용하기 위해서는 옵션에 `-lpthread`를 추가해주어야 하므로 `program3`는 `-lpthread`를 추가해주었다.

```

jaeunjung@ubuntu:~/hw2$ ls
Makefile helloworld.c input100.txt input1000.txt input10000.txt input100000.txt miniShell.cpp program1.cpp program3.cpp test.txt
data.py input.txt input1000.txt input10000.txt ls.txt miniShell.h program2.cpp programs.h
jaeunjung@ubuntu:~/hw2$ make
g++ -std=c++11 miniShell.cpp -o miniShell
g++ -std=c++11 program1.cpp -o program1
g++ -std=c++11 program2.cpp -o program2
g++ -std=c++11 program3.cpp -o program3 -lpthread
jaeunjung@ubuntu:~/hw2$ ls
Makefile      input.txt      input1000.txt      ls.txt      miniShell.h  program2      program3.cpp
data.py      input100.txt      input10000.txt      miniShell  program1      program2.cpp  programs.h
helloworld.c  input1000.txt      input100000.txt     ls.txt      miniShell.cpp  program1.cpp  program3

```

`make` 명령을 사용하면 `miniShell`, `program1`, `program2`, `program3` 실행 파일이 모두 생성되는 모습을 볼 수 있다.

```

jaeunjung@ubuntu:~/hw2$ make clean
rm -rf miniShell
rm -rf program1
rm -rf program2
rm -rf program3
jaeunjung@ubuntu:~/hw2$ ls
Makefile helloworld.c input100.txt input1000.txt input10000.txt input100000.txt miniShell.cpp program1.cpp program3.cpp test.txt
data.py input.txt input1000.txt input10000.txt ls.txt miniShell.h program2.cpp programs.h

```

`make clean` 명령을 실행하면 `make`을 통해 생성된 실행 파일들이 모두 삭제되는 것을 확인할 수 있다.

C. 개발 환경

a. `uname -a` 실행결과

```

jaeunjung@ubuntu:~/hw2$ uname -a
Linux ubuntu 5.10.19-2017122003 #1 SMP Thu Apr 8 20:35:05 PDT 2021 x86_64 x86_64 x86_64 GNU/Linux

```

과제 1에서 컴파일한 커널을 사용했다.

b. 컴파일러 정보, CPU, 메모리 정보

```

jaeunjung@ubuntu:~/hw2$ 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.

jaeunjung@ubuntu:~/hw2$ 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.

```

컴파일러는 `g++ 7.5.0` 버전을 사용하였다.

```

jaeunjung@ubuntu:~/hw2$ grep -c processor /proc/cpuinfo
2
jaeunjung@ubuntu:~/hw2$ head -n 15 /proc/cpuinfo
processor       : 0
vendor_id      : GenuineIntel
cpu family     : 6
model          : 126
model name     : Intel(R) Core(TM) i5-1038NG7 CPU @ 2.00GHz
stepping        : 5
microcode       : 0x96
cpu MHz         : 1996.800
cache size      : 6144 KB
physical id    : 0
siblings        : 1
core id         : 0
cpu cores       : 1
apicid          : 0
initial apicid : 0

```

CPU 개수는 2개이며 CPU 정보는 위와 같다. 다만 실습 과제2 실험에서는 CPU 개수가 1개 일때와 2개일때를 비교하였다.

jaeunjung@ubuntu:~/hw2\$ cat /proc/meminfo	VmallocTotal: 34359738367 kB
MemTotal: 8117940 kB	VmallocUsed: 31628 kB
MemFree: 4403588 kB	VmallocChunk: 0 kB
MemAvailable: 5535288 kB	Percpu: 45056 kB
Buffers: 51188 kB	HardwareCorrupted: 0 kB
Cached: 1313424 kB	AnonHugePages: 0 kB
SwapCached: 0 kB	ShmemHugePages: 0 kB
Active: 464056 kB	ShmemPmdMapped: 0 kB
Inactive: 2692628 kB	FileHugePages: 0 kB
Active(anon): 2328 kB	FilePmdMapped: 0 kB
Inactive(anon): 1843960 kB	CmaTotal: 0 kB
Active(file): 461728 kB	CmaFree: 0 kB
Inactive(file): 848668 kB	HugePages_Total: 0
Unevictable: 32 kB	HugePages_Free: 0
Mlocked: 32 kB	HugePages_Rsvd: 0
SwapTotal: 2097148 kB	HugePages_Surp: 0
SwapFree: 2097148 kB	Hugepagesize: 2048 kB
Dirty: 0 kB	Hugetlb: 0 kB
Writeback: 0 kB	DirectMap4k: 177984 kB
AnonPages: 1792112 kB	DirectMap2M: 5064704 kB
Mapped: 564860 kB	DirectMap1G: 5242880 kB
Shmem: 54220 kB	
KReclaimable: 84968 kB	
Slab: 166420 kB	
SReclaimable: 84968 kB	
SUnreclaim: 81452 kB	
KernelStack: 15392 kB	
PageTables: 55116 kB	
NFS_Unstable: 0 kB	
Bounce: 0 kB	
WritebackTmp: 0 kB	
CommitLimit: 6156116 kB	

메모리 정보는 위와 같다.

D. 과제 수행 중 발생한 애로사항 및 해결방법

a. C/C++에 대한 부족한 배경지식

지금까지 들은 컴퓨터과학과 수업에서 Python, Java만을 사용했기 때문에 C/C++ 이용하여 긴 코드를 짜본 적이 거의 없었다. 이번 겨울방학 때 운영체제 수업을 수강하는데 어려움이 없기 위해 C++에 대한 기초적인 공부를 했지만 execvp() 등을 사용하기 위해서는 C 언어에 대한 이해가 필수적이었다. 그래서 이번 과제를 수행하며 수많은 구글링과 디버깅 과정을 거치면서 C 언어에 대한 지식을 키울 수 있었다.

b. Program2에서의 merge 과정

처음 program2 코드를 작성할 때 부분적으로 수합된 배열을 다시 병합하고 정렬하는 과정에서 큰 어려움을 겪었다. 원래는 merge_sort 함수처럼 재귀적으로 병합을 수행하는 코드를 작성하려고 했으나 너무 복잡해져서 포기했다. 그러다가 앞에서 만든 offset, now 벡터와 큐 자료구조를 이용하면 merge_sort의 재귀적인 병합 과정을 구현할 수 있을 것이라는 것을 깨달았다. 저번 학기에 수강한 자료구조 수업의 중요성을 알 수 있었다.

c. Program3에서 pthread_exit()을 사용하려는 시도

원래는 program3 코드의 merge_sort 함수의 끝이 pthread_exit(NULL)로 종료되도록 코드를 작성하였으나 병합 정렬의 재귀적인 특성 때문에 정렬이 제대로 되지 않았다. pthread_exit(NULL)을 사용하는 것을 포기하고 return NULL;로 바꿔주니 올바르게 정렬이 되어 return NULL;을 사용하였다.

E. 결과 화면과 결과에 대한 토의 내용

a. 실습 과제1의 동작 검증

```
jaeunjung@ubuntu:~/hw2$ make
g++ -std=c++11 miniShell.cpp -o miniShell
g++ -std=c++11 program1.cpp -o program1
g++ -std=c++11 program2.cpp -o program2
g++ -std=c++11 program3.cpp -o program3 -lpthread
jaeunjung@ubuntu:~/hw2$ ./miniShell
[22:33:01] jaeunjung@home/jaeunjung/hw2$
```

위와 같이 터미널에서 make 명령을 입력해 miniShell.h, miniShell.cpp을 컴파일한 후 ./miniShell로 miniShell을 실행하였다.

<표준 UNIX 프로그램>

- cd

먼저, cd 명령을 하나만 입력했을때 HOME 디렉토리로 잘 이동하는지 보았다.

```
[22:34:59] jaeunjung@home/jaeunjung/hw2$cd
[22:35:02] jaeunjung@home/jaeunjung$pwd
/home/jaeunjung
jaeunjung@ubuntu:~/hw2$ echo $HOME
/home/jaeunjung
```

원래의 HOME 디렉토리인 /home/jaeunjung으로 잘 이동하고 있음을 알 수 있다.

```
[22:38:46] jaeunjung@home/jaeunjung$cd /home/jaeunjung/Desktop/
[22:38:54] jaeunjung@home/jaeunjung/Desktop$pwd
/home/jaeunjung/Desktop
[22:38:56] jaeunjung@home/jaeunjung/Desktop$cd ..
[22:38:59] jaeunjung@home/jaeunjung$cd ./Desktop
```

절대경로/상대경로 모두 알맞은 디렉토리로 이동하고 있다.

```
[22:43:32] jaeunjung@home/jaeunjung/hw2$cd /home/jaeunjung/ &
[22:43:38] jaeunjung@home/jaeunjung/hw2$cd ..
[22:43:48] jaeunjung@home/jaeunjung$ls
Desktop Documents Downloads Music Pictures Public Templates Videos examples.desktop hw2 snap
[22:43:49] jaeunjung@home/jaeunjung$cd Desktop Documents
cd: too many arguments
```

cd 를 백그라운드로 실행했을 때는 자식 프로세스에서 작업 디렉토리가 바뀌므로 miniShell이 실행되고 있는 디렉토리가 바뀌지 않는다. 또한 디렉토리를 여러 개 입력했을 때에는 LINUX의 bash 쉘과 유사한 메시지를 출력한다.

```
[22:46:35] jaeunjung@home/jaeunjung/hw2$ls
Makefile 'd1 d2' miniShell miniShell.cpp miniShell.h program1 program
m1.cpp program2 program2.cpp program3 program3.cpp programs.h
[22:46:36] jaeunjung@home/jaeunjung/hw2$cd 'd1 d2'
[22:46:38] jaeunjung@home/jaeunjung/hw2/d1 d2$pwd
/home/jaeunjung/hw2/d1 d2
```

마지막으로, cd 'd1 d2'와 같이 디렉토리명에 공백이 포함된 경우도 잘 작동하고 있음을 확인했다.

- ls

```
[22:49:07]jaeunjung@/home/jaeunjung/hw2$ls
Makefile    miniShell.cpp    program1.cpp    program3
'd1 d2'      miniShell.h     program2        program3.cpp
miniShell   program1        program2.cpp    programs.h
[22:49:07]jaeunjung@/home/jaeunjung/hw2$ls -a
.           Makefile    miniShell.cpp    program1.cpp    program3
..          'd1 d2'      miniShell.h     program2        program3.cpp
.vscode     miniShell   program1        program2.cpp    programs.h
[22:49:08]jaeunjung@/home/jaeunjung/hw2$ls -S
program2    program1        program3.cpp    miniShell.h
program3    miniShell.cpp  'd1 d2'        program1.cpp
miniShell   program2.cpp  programs.h    Makefile
[22:49:10]jaeunjung@/home/jaeunjung/hw2$ls -l
total 228
-rw-rw-r-- 1 jaeunjung jaeunjung    484 Apr 12 08:49 Makefile
drwxr-xr-x  2 jaeunjung jaeunjung  4096 Apr 12 22:46 'd1 d2'
-rwxr-xr-x  1 jaeunjung jaeunjung 32232 Apr 12 22:41 miniShell
-rw-rw-r--  1 jaeunjung jaeunjung  8900 Apr 12 22:41 miniShell.cpp
-rw-rw-r--  1 jaeunjung jaeunjung   716 Apr 12 09:04 miniShell.h
-rwxr-xr-x  1 jaeunjung jaeunjung 27456 Apr 12 22:34 program1
-rw-rw-r--  1 jaeunjung jaeunjung   684 Apr 12 22:27 program1.cpp
-rwxr-xr-x  1 jaeunjung jaeunjung 63456 Apr 12 22:34 program2
-rw-rw-r--  1 jaeunjung jaeunjung  5833 Apr 12 22:29 program2.cpp
-rwxr-xr-x  1 jaeunjung jaeunjung 55320 Apr 12 22:34 program3
-rw-rw-r--  1 jaeunjung jaeunjung 4730 Apr 12 22:27 program3.cpp
-rw-rw-r--  1 jaeunjung jaeunjung  2385 Apr 12 22:29 programs.h
[22:49:18]jaeunjung@/home/jaeunjung/hw2$ls -alh
total 240K
drwxrwxr-x  4 jaeunjung jaeunjung 4.0K Apr 12 22:46 .
drwxr-xr-x  2 jaeunjung jaeunjung 4.0K Apr 12 22:46 'd1 d2'
-rwxr-xr-x  1 jaeunjung jaeunjung 32K Apr 12 22:41 miniShell
-rw-rw-r--  1 jaeunjung jaeunjung 8.7K Apr 12 22:41 miniShell.cpp
-rwxr-xr-x  1 jaeunjung jaeunjung 55K Apr 12 22:34 program3
-rwxr-xr-x  1 jaeunjung jaeunjung 62K Apr 12 22:34 program2
-rwxr-xr-x  1 jaeunjung jaeunjung 27K Apr 12 22:34 program1
-rw-rw-r--  1 jaeunjung jaeunjung 5.7K Apr 12 22:29 program2.cpp
-rw-rw-r--  1 jaeunjung jaeunjung 2.4K Apr 12 22:29 programs.h
-rw-rw-r--  1 jaeunjung jaeunjung 4.7K Apr 12 22:27 program3.cpp
-rw-rw-r--  1 jaeunjung jaeunjung 684 Apr 12 22:27 program1.cpp
-rw-rw-r--  1 jaeunjung jaeunjung 716 Apr 12 09:04 miniShell.h
-rw-rw-r--  1 jaeunjung jaeunjung 484 Apr 12 08:49 Makefile
drwxr-xr-x 20 jaeunjung jaeunjung 4.0K Apr 12 01:18 ..
drwxrwxr-x  2 jaeunjung jaeunjung 4.0K Apr  8 22:34 .vscode
[22:50:32]jaeunjung@/home/jaeunjung$ls hw2
Makefile    miniShell.cpp    program1.cpp    program3
'd1 d2'      miniShell.h     program2        program3.cpp
miniShell   program1        program2.cpp    programs.h
```

ls 명령어도 -a, -S, -l, -alh, ls \${폴더} 등 여러 옵션에서 알맞게 작동함을 확인할 수 있었다.

```

- mkdir
[22:58:26]jaeunjung@/home/jaeunjung/hw2$ls
Makefile minishell.cpp program1.cpp program3
'd1 d2' minishell.h program2 program3.cpp
minishell program1 program2.cpp programs.h
[22:58:27]jaeunjung@/home/jaeunjung/hw2$mkdir test1
[22:58:29]jaeunjung@/home/jaeunjung/hw2$mkdir -m 777 test2
[22:58:33]jaeunjung@/home/jaeunjung/hw2$ls -l
total 236
-rw-rw-r-- 1 jaeunjung jaeunjung 484 Apr 12 08:49 Makefile
drwxr-xr-x 2 jaeunjung jaeunjung 4096 Apr 12 22:46 'd1 d2'
-rwxr-xr-x 1 jaeunjung jaeunjung 32232 Apr 12 22:41 minishell
-rw-rw-r-- 1 jaeunjung jaeunjung 8900 Apr 12 22:41 minishell.cpp
-rw-rw-r-- 1 jaeunjung jaeunjung 716 Apr 12 09:04 minishell.h
-rwxr-xr-x 1 jaeunjung jaeunjung 27456 Apr 12 22:34 program1
-rw-rw-r-- 1 jaeunjung jaeunjung 684 Apr 12 22:27 program1.cpp
-rwxr-xr-x 1 jaeunjung jaeunjung 63456 Apr 12 22:34 program2
-rw-rw-r-- 1 jaeunjung jaeunjung 5833 Apr 12 22:29 program2.cpp
-rwxr-xr-x 1 jaeunjung jaeunjung 55320 Apr 12 22:34 program3
-rw-rw-r-- 1 jaeunjung jaeunjung 4730 Apr 12 22:27 program3.cpp
-rw-rw-r-- 1 jaeunjung jaeunjung 2385 Apr 12 22:29 programs.h
drwxr-xr-x 2 jaeunjung jaeunjung 4096 Apr 12 22:58 test1
drwxrwxrwx 2 jaeunjung jaeunjung 4096 Apr 12 22:58 test2
[22:58:34]jaeunjung@/home/jaeunjung/hw2$mkdir -p ./test3/test4
[22:58:41]jaeunjung@/home/jaeunjung/hw2$ls
Makefile minishell.cpp program1.cpp program3 test1
'd1 d2' minishell.h program2 program3.cpp test2
minishell program1 program2.cpp programs.h test3
[22:58:42]jaeunjung@/home/jaeunjung/hw2$ls test3
test4
[22:58:47]jaeunjung@/home/jaeunjung/hw2$mkdir -v test5
mkdir: created directory 'test5'

```

기본적인 `mkdir` 명령어가 잘 작동하고, 디렉토리를 생성할 때 퍼미션을 주는 옵션 `-m ${숫자}` 옵션도 알맞게 수행되었다. `ls -l` 명령어를 입력해서 777 권한으로 만든 `test2`의 권한이 다른 것을 확인해보았다. 디렉토리를 재귀적으로(`recursive`) 만드는 옵션인 `-p`를 통해 `./test3/test4` 디렉토리를 한 번에 만들 수 있었으며 `ls test3` 명령을 통해 `test3` 안에 `test4` 디렉토리가 형성된 것을 확인했다. `verbose`를 나타내는 `-v` 옵션을 통해 `test5`를 만들었고 `verbose`가 작동함을 알 수 있었다.

```

[23:02:28]jaeunjung@/home/jaeunjung/hw2$mkdir /test6
mkdir: cannot create directory '/test6': Permission denied
[23:02:34]jaeunjung@/home/jaeunjung/hw2$sudo mkdir -v /test6
mkdir: created directory '/test6'

```

다만 `root` 디렉토리 바로 하단에 새로운 폴더를 생성하려고 하자 `permission` 에러가 뜨는 것을 확인했고, `sudo` 명령을 통해 `root` 디렉토리에 `test6` 폴더를 만들 수 있었다.

```

[04:24:34]jaeunjung@/home/jaeunjung/hw2$mkdir -v 'test blank"
[04:25:21]jaeunjung@/home/jaeunjung/hw2$mkdir -v "test blank"
mkdir: created directory 'test blank'

```

마지막으로, 공백이 포함된 디렉토리를 만들라는 명령을 알맞은 형식으로 입력했을 때 작동하는 것을 볼 수 있었다.

```
- rm  
[23:08:36]jaeunjung@/home/jaeunjung/hw2$cat test.txt  
Hello world!  
[23:09:08]jaeunjung@/home/jaeunjung/hw2$rm test.txt  
[23:09:14]jaeunjung@/home/jaeunjung/hw2$cat test.txt  
cat: test.txt: No such file or directory  
[23:09:17]jaeunjung@/home/jaeunjung/hw2$ls  
Makefile      minishell.cpp  program2      programs.h  test5  
'd1 d2'      minishell.h   program2.cpp  test1  
helloworld.c  program1     program3     test2  
minishell     program1.cpp  program3.cpp  test3
```

먼저 기본적인 파일 삭제가 되는지 확인하기 위해 `test.txt` 파일을 하나 만들어주었다. `test.txt` 파일 안에는 `Hello world!`가 적혀있다. `rm test.txt` 명령을 수행하고 나니 `cat test.txt`가 작동하지 않았고 `ls`에도 `test.txt` 파일이 사라졌다.

```
[19:44:36]jaeunjung@/home/jaeunjung/hw2$rm -R -v test1 test2  
removed directory 'test1'  
removed directory 'test2'  
[23:12:03]jaeunjung@/home/jaeunjung/hw2$rm -R test3  
[23:12:24]jaeunjung@/home/jaeunjung/hw2$ls  
Makefile      minishell      program1     program2.cpp  programs.h  
'd1 d2'      minishell.cpp  program1.cpp  program3  
helloworld.c  minishell.h   program2     program3.cpp
```

재귀적으로 삭제를 수행하는 `-R` 옵션을 통해 방금 생성한 `test1`, `test2`, `test3/test4`, `test5`를 모두 삭제해주었다.

```
[23:16:47]jaeunjung@/home/jaeunjung/hw2$rm -rf -v /test6  
rm: cannot remove '/test6': Permission denied  
[23:16:52]jaeunjung@/home/jaeunjung/hw2$sudo rm -rf -v /test6  
removed directory '/test6'
```

마지막으로, 경고 없이 파일과 폴더를 강제로 삭제하는 `-rf` 옵션을 통해 `root` 디렉토리에 만든 `test6` 폴더를 지우려고 했으나 permission 에러가 나서 `sudo`로 수행하였고 삭제가 잘 되는 것을 볼 수 있었다.

```
- echo  
[04:28:10]jaeunjung@/home/jaeunjung/hw2$echo Operating System  
Operating System  
[04:28:17]jaeunjung@/home/jaeunjung/hw2$echo "Operating System"  
Operating System  
[04:28:30]jaeunjung@/home/jaeunjung/hw2$echo -e "Operating \bSystem"  
OperatingSystem
```

`echo` 명령어도 따옴표가 존재하지 않는 문자열, 존재하는 문자열 모두 정상적으로 작동함을 확인했다. `-e` 옵션에도 마찬가지였다.

```

- cat, head, tail
[04:33:13]jaeunjung@/home/jaeunjung/hw2$cat test.txt
1 Hello world!
2 Hello world!
3 Hello world!
[04:33:20]jaeunjung@/home/jaeunjung/hw2$head -n 2 test.txt
1 Hello world!
2 Hello world!
[04:33:25]jaeunjung@/home/jaeunjung/hw2$tail -n 2 test.txt
2 Hello world!
3 Hello world!

```

test.txt를 만들어준 다음에 cat, head, tail 명령어를 실행시켜보았다.
cat은 test.txt 안에 든 내용을 정상적으로 출력하고 있으며 head, tail 모두 앞/뒤에서 n 번째 줄까지 출력함을 확인했다.

```

- time
[04:35:39]jaeunjung@/home/jaeunjung/hw2$time sleep 100
0.00user 0.00system 1:40.00elapsed 0%CPU (0avgtext+0avgdata 2112maxresident)k
0inputs+0outputs (0major+89minor)pagefaults 0swaps
[04:37:24]jaeunjung@/home/jaeunjung/hw2$ls
Makefile helloworld.c miniShell.cpp program1      program2      program
3           programs.h
'd1 d2'     miniShell.h    miniShell.h    program1.cpp   program2.cpp   program
3.cpp      test.txt
0.00user 0.00system 0:00.00elapsed 100%CPU (0avgtext+0avgdata 2628maxresident)k
0inputs+0outputs (0major+126minor)pagefaults 0swaps
jaeunjung@ubuntu:~/hw2$ time sleep 100

real    1m40.005s
user    0m0.000s
sys     0m0.004s
jaeunjung@ubuntu:~/hw2$ time ls
Makefile      miniShell      program1      program2.cpp  programs.h
'd1 d2'        miniShell.cpp  program1.cpp  program3      test.txt
helloworld.c  miniShell.h    program2      program3.cpp

real    0m0.004s
user    0m0.004s
sys     0m0.000s

```

아래는 miniShell에서 time을 실행한 것이고 아래는 LINUX 쉘에서 time을 실행한 것인데, 정보는 알맞게 출력하고 있지만 출력 양식이 LINUX 쉘과는 약간 상이했다.

```

- exit
jaeunjung@ubuntu:~/hw2$ ./miniShell
[04:43:12]jaeunjung@/home/jaeunjung/hw2$exit
jaeunjung@ubuntu:~/hw2$ ./miniShell
[04:43:14]jaeunjung@/home/jaeunjung/hw2$ exit
jaeunjung@ubuntu:~/hw2$ ./miniShell

```

사용자가 exit 명령어를 사용하면 miniShell이 종료된다.

<백그라운드 실행>

```
[04:47:48]jaeunjung@/home/jaeunjung/hw2$sleep 100 &
[04:47:55]jaeunjung@/home/jaeunjung/hw2$ls &
[04:47:57]jaeunjung@/home/jaeunjung/hw2$ Makefile      minishell      program1
          program2.cpp  programs.h
'd1 d2'      minishell.cpp  program1.cpp  program3      test.txt
helloworld.c  minishell.h   program2     program3.cpp

[04:47:58]jaeunjung@/home/jaeunjung/hw2$
```

백그라운드 실행은 `stdout`을 쓰는 명령어, 쓰지 않는 명령어 하나씩 실행해보았다. `sleep`은 프로그램 실행을 유예시키는 UNIX 명령어인데 백그라운드로 실행하니 바로 부모 프로세스로 돌아왔다. 하지만 `ls` &와 같이 `stdout`을 쓰는 명령어는 엔터(또는 다른 명령어)를 쳐야 다시 `minishell`이 실행되는 것을 알 수 있었는데 LINUX 쉘도 이와 동일하게 동작한다.

<리다이렉션>

- 출력 리다이렉션

```
[04:47:58]jaeunjung@/home/jaeunjung/hw2$ls > ls.txt
[04:50:56]jaeunjung@/home/jaeunjung/hw2$cat ls.txt
Makefile
d1 d2
helloworld.c
ls.txt
minishell
minishell.cpp
minishell.h
program1
program1.cpp
program2
program2.cpp
program3
program3.cpp
programs.h
test.txt
```

`ls`의 출력을 `ls.txt`로 바꿔주었고 그에 따라 `ls`의 출력 내용을 담은 `ls.txt`가 생성되었다.

- 입력 리다이렉션

```
[04:58:01]jaeunjung@/home/jaeunjung/hw2$grep program < ls.txt
program1
program1.cpp
program2
program2.cpp
program3
program3.cpp
programs.h
```

방금 생성한 `ls.txt` 파일을 `grep` 프로그램의 표준 입력으로 리다이렉션 해주었다. `program`이 포함된 파일들만 잘 출력하고 있음을 확인했다.

<프로그램 실행>

```
[05:01:52]jaeunjung@home/jaeunjung/hw2$./program1 < input.txt
523490 10000 231 213 123 10 3 -768 -12312
5
```

실습 과제 2에서 만든 `program1`에 `input.txt`를 표준 입력으로 넣어주었고 `program1`이 정상적으로 실행되고 있음을 알 수 있다.

b. 실습 과제2의 동작 검증

1) CPU 개수에 따른 결과 차이 및 분석

CPU 개수를 1 개로 설정했을 때와 CPU 개수를 2 개로 설정했을 때 차이점이 있는지 보고자 했다. 길이가 100000 인 배열을 만들어서 프로그램 1, 2, 3를 5 번씩 실행시켜준 뒤 평균을 내었다. 프로세스와 스레드 개수는 5로 설정하였다.

CPU 개수 \ 프로그램	프로그램 1	프로그램 2	프로그램 3
1	26030ms	14671ms	33886ms
2	25411ms	12314ms	30258ms

CPU 개수가 1 개에서 2 개로 증가함에 따라 프로그램 2, 프로그램 3의 실행시간은 10%씩 단축된 것에 비해 프로그램 1의 실행시간 감소율은 미미했다. 다만 특이한 점은 멀티 스레딩을 사용하는 프로그램 3의 실행 시간이 제일 길다는 것인데, 이는 `mutex`를 사용하는 데서 오는 한계와 프로그램 3의 `merge_sort`는 함수의 인자들을 재귀적으로 메모리를 동적 할당 해주기 때문에 그렇다고 생각할 수 있다. 또한 병렬 프로그래밍이 항상 정답이 되지 않는다는 것을 실감할 수 있었다.

2) 프로세스 수, 스레드 수에 따른 결과 차이 및 분석

다음으로는 프로세스 수, 스레드 수에 따른 성능차이를 보고자 했다. CPU 개수는 2로 고정하였으며 각 측정값은 5 개의 결과값들의 평균이다. 데이터는 1)에서 사용한 길이가 100000 인 데이터를 사용하였다.

프로세스/스레드 수 \ 프로그램	프로그램 2	프로그램 3
1	5916ms	32582ms
10	12811ms	41271ms
100	27998ms	81520ms

위의 결과로 생각할 수 있는 것은 프로세스 수/스레드 수를 무조건 크게 하는 것이 효율적이지는 않다는 것이다. 왜냐하면 새로운 프로세스와 스레드를 만드는데 드는 오버헤드가 존재하기 때문이다. 특히 이는 프로그램 2에서 두드러지는데, 파일 입출력에 따른 오버헤드로 인해 프로세스 수가 증가함에 따라 실행 시간이 길어지는 것을 볼 수 있다.

3) 데이터의 크기에 따른 결과 차이 및 분석

마지막으로 데이터의 크기에 따라 달라지는 결과를 보고자 했다. CPU 개수는 2 개이고 프로세스 수와 스레드 수는 5로 고정하였다.

데이터 크기	프로그램 1	프로그램 2	프로그램 3
100	27ms	1159ms	703ms
1000	328ms	1202ms	1383ms
10000	3832ms	1954ms	8016ms
100000	24602ms	10969ms	35631ms



전반적인 실행시간은 프로그램 3 > 프로그램 1 > 프로그램 2 순이었다. 수업 시간에 배운 것과 다르게 멀티 스레딩을 이용했을 때 속도가 제일 느리게 나왔는데, 이 이유로는 mutex의 사용과 재귀적 동적 메모리 할당, merge sort 함수의 재귀적 구조, 스레드 생성의 오버헤드 등이 있다.

F. 참조 문헌

- <http://www.cplusplus.com/reference/ctime/strftime/>