Pintos Project 2 : User Program

(설계 프로젝트 수행 결과)

과목 명 : [CSE4070] 운영체제

담당 교수 : 소정민

조 / 조원 : 1조 / 이규연, 허남규

개발 기간 : 2018/10/1 ~ 2018/10/7

프로젝트 제목 : Pintos Project 2 : User Program

제출일 : 2018/11/6

참여 조원 : 20141552 이규연, 20161662 허남규

I. 개발 목표

- Pintos 운영체제의 기본 코드는 사용자의 프로그램을 메모리에 적재하고 실행시킬 수 있도록 구현되어 있습니다. 하지만 입출력이나 기타 상호작용을 하지 못하는 매우 제한적인 상태로 작동합니다. 이번 프로젝트는 사용자의 프로그램이 운영체제와 상호작용할 수 있도록 시스템 호출 (system call)을 구현하는 것이 목표입니다.
- 1차 프로젝트 User Program Basic에 이어서, Pintos의 파일 시스템에 관련된 시스템 호출과 동기화 이슈에 대한 해법을 구현하였습니다.

II. 개발 범위 및 내용

A. 개발 범위

- **File System Call Implementation**
  - ■ 지난 번 프로젝트에서 시스템 호출 중 가장 필수적인 항목들만 구현하였습니다. 이번 프로젝트에서는 파일 시스템을 다루는 시스템 호출들에 대한 추가적인 구현을 합니다.
  - ■ Pintos의 소스코드 중 syscall.c에 아직 구현되지 않은 시스템 호출의 함수들을 작성합니다.
- **Denying Writes to Executables**
  - ■ 프로세스는 필요한 파일을 메모리에 로드한 후 수행되기 때문에 디스크 상의 실행 파일이 수정되더라도 프로세스에 영향을 주지 않습니다. 하지만 Pintos의 경우, 안정성을 위하여 수행 중인 프로세스의 실행 파일을 지워지지 않도록 합니다.
  - ■ Pintos 소스코드 중 프로세스가 생성되는 부분을 수정합니다.
- **Memory Management**
  - ■ 프로세스가 수행 중에 파일을 연 채로 종료될 수 있습니다. 프로세스가 비정상적으로 종료하더라도 할당 받은 자원을 모두 반납하고 종료될 수 있도록 메모리 관리를 해야 합니다.
- **Synchronization**
  - ■ 이번 프로젝트에서 가장 중요한 두개의 항목은 다음과 같습니다.
    - ◆ 파일에 읽기/쓰기/제거에 대한 동기화.
    - ◆ 프로세스의 수행/종료에 대한 동기화, 비정상 종료 처리.
  - ■ 하나의 파일을 동시에 여러 프로세스가 조작하는 행위는 시스템의 안정성에 매우 부정적인 영향을 줍니다. 이에 대한 적절한 동기화를 위해 Pintos에 구현된 lock을 사용하였습니다.
  - ■ 프로세스의 수행/종료에 대한 동기화를 위하여 Pintos에 구현된 semaphore을 사용하였습니다.

B. 개발 내용

- **File System Call Implementation**

  - Pintos에는 제한적이지만 완전한 파일 시스템이 구현되어 있습니다. Pintos 위에서 수행중인 사용자 프로그램은 시스템 호출을 이용하여 파일 시스템 내의 파일에 읽고 쓰는 조작을 할 수 있습니다. 사용자 프로그램이 시스템 호출을 이용하여 파일을 조작할 수 있도록 구현합니다.

  - 다음의 시스템 호출을 구현하였습니다. 인자들은 Pintos.pdf에 제시된 함수 원형을 따랐습니다.

    - create() : 새로운 파일을 생성합니다.
    - remove(): 기존의 파일을 삭제합니다.
    - open(): 프로세스가 파일을 열도록 합니다.
    - close(): 열린 파일을 닫습니다.
    - write(): 파일에 새로운 내용을 씁니다.
    - read(): 파일에서 읽어옵니다.
    - seek(): 파일의 현재 위치를 새로운 위치로 바꿉니다.
    - tell(): 파일의 현재 위치를 알려줍니다.
    - filesize(): 파일의 크기를 반환합니다.

- **Denying Writes to Executables**

  - Pintos에서 수행중인 프로세스의 실행 파일은 보호되어야 합니다. 이를 위하여 file_deny_write()함수를 프로세스의 실행 파일에 사용하여 수정되는 것을 방지합니다.

- **Memory Management**

  - 지난 프로젝트에서는 로딩에 실패한 프로세스 등에 대한 처리가 미흡했습니다. 이는 시스템의 안정성 및 자원 관리에 부정적인 영향을 주기 때문에 이에 대한 추가적인 기능을 구현하였습니다.

  - 프로세스가 파일을 열고 비정상적으로 종료를 하는 경우, 메모리 누수를 막

기 위하여 열린 파일에 대한 처리가 필요합니다.

- **Synchronization**

  - 파일 시스템을 사용하는 시스템 호출의 경우, 하나의 파일에 동시에 읽고 쓰고 지우는 경우가 발생할 수 있습니다. 이러한 경우에도 문제 없이 수행될 수 있도록 적절한 동기화 기법이 필요합니다.
  - 프로세스의 동기화 또한 추가적인 고려가 필요합니다. 프로세스가 비정상 종료를 하는 경우에 대해서도 적절한 예외 처리가 수행되어야 합니다.
  - 동기화를 위하여 다음의 기능들을 사용하였습니다.
    - semaphore: 프로세스간의 동기화를 위하여 사용합니다.
    - lock: 파일 시스템에 사용되는 시스템 호출의 임계 구역에 사용합니다.

III. 추진 일정 및 개발 방법

A. 추진 일정

- 2018.10.29 ~ 2018.10.30: Pintos 코드 분석, 문제 상황 파악, 전략 수립.
- 2018.10.31: File system call 구현.
- 2018.11.1 ~ 2018.11.2: 동기화 및 메모리 문제 해결.
- 2018.11.3: 보고서 작성.

B. 개발 방법

- **From Project 1: User Program Basic**
  - ■ 이전 프로젝트에서 기존에 구현된 바쁜 대기 방식에는 프로세서의 자원을 낭비하는 문제가 있었습니다. 이 부분은 Pintos에 구현된 semaphore을 사용하는 것으로 바꾸었습니다. 반복문에 걸려 CPU를 사용하면서 기다리는 방식이 아닌, 스레드를 차단시키는 방법으로 구현되어있기 때문에 이전 프로젝트에서 시간 초과 문제로 통과하지 못한 multi-recurse 테스트를 통과할 수 있게 되었습니다.
- **File System Call Implementation & Denying Writes to Executables**
  - ■ 파일 시스템에 대한 조작을 위한 시스템 호출에 필요한 함수들은 Pintos.pdf를 참고하여 작성하였습니다. 주어진 함수 원형을 최대한 이용하면서, filesys.c와 file.c에 구현되어 있는 함수들을 최대한 사용하여 효율적으로 코드를 작성할 수 있도록 하였습니다.
- **Memory Management & Synchronization**
  - ■ Pintos에서 동적 할당이 어떻게 이루어지는지, 프로세스에 할당된 자원이 언제 반납되는지에 대한 정확한 이해가 필요합니다. 이전 프로젝트에서 구현된 process_wait()와 process_exit()에 대한 추가적인 기능의 추가가 필요했습니다.

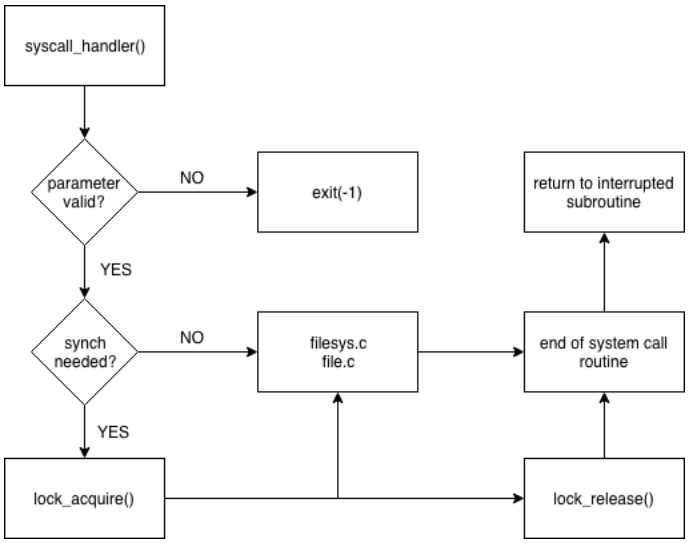
C. 연구원 역할 분담

- 이규연: 프로세스 동기화 구현, 파일 시스템 호출 구현, 보고서 작성.
- 허남규: 파일 시스템 동기화 구현.

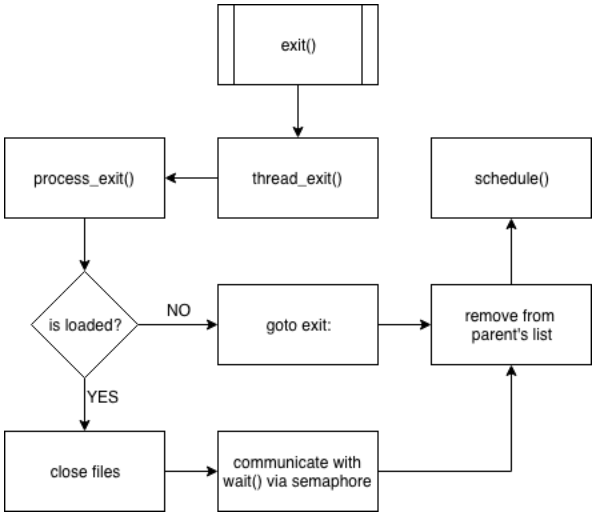
IV. 연구 결과

A. 합성 내용

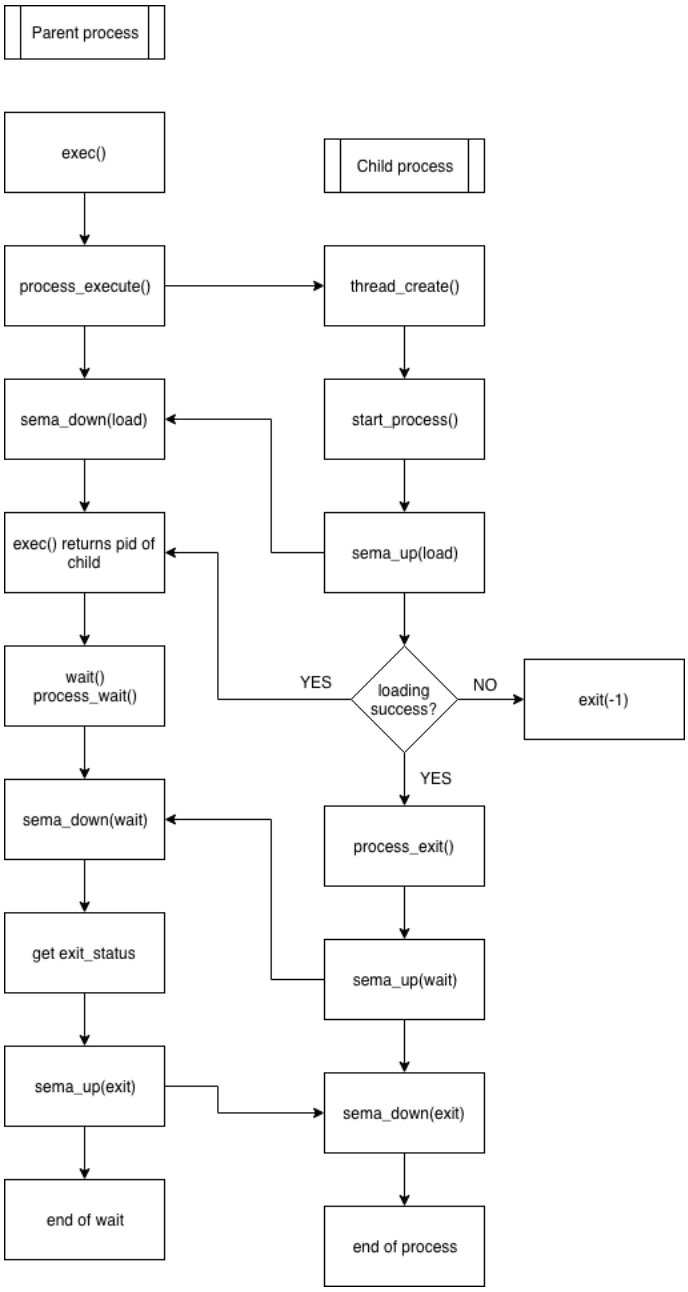
- 파일 시스템을 조작하는 시스템 호출은 다음과 같은 과정을 거칩니다.



- 프로세스의 자원 반납 과정은 다음과 같습니다.



- 부모-자녀 프로세스의 동기화 기법은 다음과 같습니다. 자녀 프로세스가 로딩에 실패할 경우 부모는 wait()을 호출하지 않습니다.



B. 제작 내용

- **Handling Files in Pintos**

  ■ 각각의 프로세스는 파일을 열고 닫을 수 있습니다. 해당 프로세스에서 열린 파일에 대한 정보를 저장하기 위하여 list file_list라는 연결 리스트를 thread 구조체에 추가하였습니다. 파일은 파일 디스크립터로 구분됩니다. 파일 디스

크립터는 프로세스마다 독립적으로 관리되며, 하나의 프로세스가 종료될 때까지 중복되지 않도록 구현하였습니다. 구조체 thread에서 다음 파일 디스크립터를 저장하는 변수를 저장하여 매번 업데이트 하는 방식으로 구현하였습니다. 파일을 제한 없이 추가할 수 있도록 구현하기 위하여 struct file_wrapper라는 리스트 원소 구조체를 만들어 파일에 대한 포인터 변수(f)와 그 파일을 나타내는 디스크립터(fd)를 원소로 갖도록 하였습니다. 각각의 file_wrapper은 file_list에 연결되어 프로세스에서 접근 가능합니다.

< struct file_wrapper >

```
struct file_wrapper
{
    int fd;
    struct file *f;
    struct list_elem file_elem;
};
```

■ 다음은 이전 프로젝트에 이어서 thread.h의 struct thread 구조체에 새로 추가된 항목들입니다.

< struct thread, thread.h >

```
/* New elements. */
struct thread *parent;              /* Pointer to parent. */
struct list children;               /* List of children. */
struct list_elem siblings;          /* list_elem for children. */

int exit_status;                    /* Exit status. */
bool needs_wait;                    /* If process is waited, set false. */
bool is_loaded;                     /* If load fails, set false. */

struct semaphore sema_load;         /* For start_process(). */
struct semaphore sema_wait;         /* For process_wait(). */
struct semaphore sema_exit;         /* For process_exit(). */

bool is_child_loaded;               /* To check whether child is loaded. */

int fd_next;                        /* Vairable to store the next fd. */
struct list file_list;              /* List of open files. */
struct file *executable;            /* Executable file of process.*/
```

■ 파일 시스템에 대한 시스템 호출을 구현하기 위하여 Pintos 소스코드 중 filesys.c와 file.c에 있는 함수들을 이용하였습니다.

- 다음의 경우에 대하여 동기화가 필요합니다.
  - remove() : 파일이 삭제되는 도중에 다른 프로세스가 접근하면 안됩니다.
  - read(): 파일을 읽는 도중에 다른 프로세스가 접근하면 안됩니다.
  - write(): 파일에 쓰는 도중에 다른 프로세스가 접근하면 안됩니다.
- 이는 코드 상의 임계 구역으로 생각할 수 있습니다. 임계 구역 내에서는 하나의 프로세스만이 수행될 수 있도록 적절한 동기화 기법이 필요합니다. 이를 위하여 Pintos에 구현된 lock을 사용하였습니다.
- 위에 언급된 시스템 호출에서 해당 조작을 수행하기 위해 filesys.c에 구현된 함수를 사용하였습니다. 이 함수들이 호출된 부분을 임계구역으로 정의하여 앞뒤로 syscall_lock이라는 lock 변수에 대하여 lock_acquire()와 lock_release()를 호출하도록 하였습니다. 이렇게 하면 동시에 두개의 프로세스가 동시에 파일 시스템에 해당 조작을 할 수 없어 정상적으로 수행될 수 있습니다.
- 파일 시스템 호출을 위한 다른 함수들은 filesys.c와 file.c의 함수들을 그대로 사용하였기 때문에 자세한 구현 내용은 보고서에 생략하였습니다.
- 시스템 호출 중 lock을 사용한 부분은 다음과 같습니다. 모두 해당 syscall.c에 구현된 시스템 호출 함수 내에 존재하는 코드입니다.
  - 다음 코드에서 fd2file() 함수는 파일 디스크립터에 해당하는 파일 포인터를 현재 수행중인 프로세스의 파일 리스트에서 찾아서 반환하는 함수입니다. 현재 프로세스의 파일 리스트에 존재 하지 않는 디스크립터의 경우 NULL 포인터를 반환합니다.

< remove() >

```
/* Synchronization needed. */
lock_acquire (&syscall_lock);
result = filesys_remove (file);
lock_release (&syscall_lock);
```

< read() >

```
/* Read from file */
else if (fd > 1)
{
    f = fd2file (fd);
    if (!f)
        nread = -1;

    else
    {
        /* Synchronization needed. */
        lock_acquire (&syscall_lock);
        nread = file_read (f, buffer, size);
        lock_release (&syscall_lock);
    }
}
```

< write() >

```
/* Write to file. */
else
{
    f = fd2file (fd);
    if (!f)
        i = -1;

    /* Synchronization needed. */
    lock_acquire (&syscall_lock);
    i = file_write (f, buffer, size);
    lock_release (&syscall_lock);
}
```

- **Process Control & Memory Management**

  - 지난 프로젝트와는 달리. 더 많은 테스트 케이스를 수행하기 위해서는 추가적으로 고려할 사항들이 생겼습니다.

  - 첫째는 프로세스가 로딩에 실패하는 경우입니다. 이에 대한 경우는 크게 두 가지로 다음과 같습니다.

    - 파일이 존재하지 않는 경우.

    - 파일이 존재하지만 로딩 도중에 실패한 경우.

  - 두 경우 정상적인 프로세스의 수행이 불가능합니다. 이러한 경우를 제대로 처리하지 못하면 메모리상에 비정상적인 프로세스가 계속해서 적재되기 때문에 메모리 누수가 발생합니다. 이러한 프로세스를 구분하기 위해 시스템 호출 exec()에서 생성되는 자녀 프로세스가 로딩을 성공적으로 끝냈는지 여부를 확인하도록 합니다. 만약 자녀가 로딩을 실패한다면, 자녀 프로세스의 pid (혹은 tid)를 반환하지 않고 -1을 반환하도록 합니다. 이러한 프로세스는 부모가 기다려주지 않기 때문에 스스로 exit(-1)을 호출하여 정리되도록 하였습니다. 이는 다음과 같이 구현하였습니다. (start_process() 내의 코드)

```
/* Check whether loading is successful. In both cases,
 * we have to UP the parent's SEMA_LOAD. */
if (!success)
{
    /* The child failed loading */
    t->is_loaded = false;

    /* Notice parent that this process failed loading. */
    t->parent->is_child_loaded = true;
    sema_up (&t->parent->sema_load);

    /* Exits by itself since parent does not wait. */
    exit(-1);
}

else
{
    /* The child successfuly loaded. */
    t->is_loaded = true;
    sema_up (&t->parent->sema_load);

    /* Deny writes to the file which is loaded on the process. */
    t->executable = filesys_open (t->name);
    file_deny_write (t->executable);
}
```

■ 둘째는 로딩에 성공한 프로세스의 종료 시 자원 반납 여부입니다. 이 경우 프로세스는 수행 중에 파일을 열 수 있습니다. 이러한 프로세스가 종료 할 경우 열린 파일을 모두 닫고 동적으로 할당된 메모리를 모두 반납해야 메모리 누수를 막을 수 있습니다. 수행에 성공한 모든 프로세스에 process_exit() 함수에서 닫지 않은 파일이 있는지, 반납되지 않은 동적 할당 메모리가 있는지 확인하여 모두 처리 하도록 하였습니다. 이는 다음과 같이 구현하였습니다.

```
  /* Close the executable file of this thread to allow writes. */
  if (cur->is_loaded)
    file_close (cur->executable);

  /* The case of crashed process; parent does not wait because
   * exec() did not return a valid pid when failed load. It has
   * to deallocate resources itself, if necessary.
   * Crashed process cannot open any files or spawn children. */
  else
    goto exit;

  /* Close all open files */
  while (!list_empty (&cur->file_list))
  {
    e = list_pop_front (&cur->file_list);
    fw = list_entry (e, struct file_wrapper, file_elem);

    /* Why not close(fw->fd)? */
    file_close (fw->f);
    free(fw);
  }

  /* In this part of code, parent is still suspended, which cannot
   * retrieve the status. Hence, notice the parent. */
  sema_up(&cur->sema_wait);

  /* Waits for the parent to retrieve the exit status. */
  sema_down(&cur->sema_exit);

exit:
  /* Remove current process from parent's list */
  e = list_head (&parent->children);
  while ((e = list_next (e)) != list_end (&parent->children))
  {
    child = list_entry (e, struct thread, siblings);
    if (cur->tid == child->tid)
      list_remove (e);
  }
```

- **Synchronization Strategy**

  ■ 부모 프로세스가 exec() 시스템 호출을 실행하면 우선 process_execute()함수에서 필요한 실행 파일을 찾은 후 start_process()에서 로딩을 시작합니다. 여기서 exec() 함수는 새로 생성되는 자녀 프로세스가 성공적으로 로딩을 마쳤는지 알아야 합니다. 따라서 process_execute()함수가 끝난 후, 현재 프로세스는 세마포어 변수를 이용하여 새로 생성된 프로세스에 대한 정보를 받아야합니다. 현재 프로세스는 process_execute() 다음에 sema_down()을 하여 자녀 프로세스의 로딩을 기다립니다. 자녀 프로세스가 로딩을 마친 후, 로딩이 성공했는지 부모 프로세스에게 알리면서 sema_up()을 하여 부모가 실행되도록 합니다. 만약 로딩에 실패한 경우 exec()은 -1을 반환하고 성공한 경우 새로운 프로세스의 pid를 반환합니다.

```
/* run process_execute() */
tid = process_execute(file);

/* Starts loading after returning from process_execute()
 * wait for child to load. */
sema_down (&t->sema_load);

/* Case 1) & 2): Thread exits; it needs to be freed. */
if (t->is_child_loaded)
{
    /* Reset for re-use. */
    t->is_child_loaded = false;
    tid = -1;
}

/* Case 3): Return the tid of child. */
return tid;
```

  ■ 로딩에 실패한 프로세스의 pid는 생성되지만 exec() 시스템 호출에 의해 반환되지 않습니다. 그렇기 때문에 로딩에 실패한 프로세스는 부모가 wait()을 호출하지 못할 뿐더러 할 필요가 없습니다. 이러한 프로세스는 exit(-1)을 통해서 스스로 종료하도록 합니다. 이 과정에서 process_exit() 함수를 거치게 되는데, is_loaded 변수를 통해 이러한 프로세스를 구분할 수 있습니다. 로딩에 실패한 프로세스는 부모와의 동기화를 고려할 필요가 없으니 곧바로 리스트에서 제거되고 종료되도록 합니다.

  ■ 로딩에 성공한 프로세스에 대한 동기화는 이전 프로젝트의 방식을 따릅니다.

■ 단, busy_waits() 함수와 busy_wait_end() 함수를 이용한 바쁜 대기 방식이 아닌 sema_down() 함수와 sema_up() 함수를 이용하여 구현하였습니다.

■ 자녀 프로세스가 부모보다 먼저 종료되는 경우, 자녀의 sema_wait의 값은 1이 됩니다. 이러한 경우에는 부모가 sema_down()을 하더라도 기다리지 않고 바로 수행이 되기 때문에 문제되는 상황이 발생하지 않습니다.

C. 시험 및 평가 내용

- 다음은 make grade의 결과입니다.

■ 모든 테스트 케이스에 대하여 성공적으로 수행하였습니다.

< make grade >

```
TOTAL TESTING SCORE: 100.0%
ALL TESTED PASSED -- PERFECT SCORE

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

SUMMARY BY TEST SET

Test Set                                        Pts Max  % Ttl  % Max
----------------------------------------------- --- --- ------ ------
tests/userprog/Rubric.functionality           108/108  35.0%/ 35.0%
tests/userprog/Rubric.robustness               88/ 88  25.0%/ 25.0%
tests/userprog/no-vm/Rubric                     1/  1  10.0%/ 10.0%
tests/filesys/base/Rubric                      30/ 30  30.0%/ 30.0%
----------------------------------------------- --- --- ------ ------
Total                                                  100.0%/100.0%

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

SUMMARY OF INDIVIDUAL TESTS
```

V. 기타

A. 연구 조원 기여도

- 이규연 50%, 허남규 50%

B. 소감

- 운영체제 내에서 파일 시스템이 어떻게 작용하는지, 또한 어떤 부분에서 주의를 해야 하는지 이해할 수 있는 기회였습니다.
- 프로세스에 대한 이해를 확고히 할 수 있는 좋은 기회였으며, multi-oom 테스트 케이스를 작업하면서 메모리 관리의 중요성을 깨달았습니다. 어떤 부분에서 어떻게 문제가 생길 수 있는지 다시 생각해볼 수 있는 기회였습니다.
- 자신의 코드를 충분히 이해하는 것 조차 매우 어려운 일이라는 것을 깨달았습니다. 디버깅 하는 과정이 매우 어려웠는데, Kernighan's law가 어떤 의미인지 몸소 알게 되었습니다.
- 하나의 큰 프로그램을 완성할 수 있는 좋은 기회였습니다. 단순히 강의만으로 이론적인 내용을 배우는 것 보다 직접 구현하는 것이 공부에 큰 도움이 된다는 것을 느꼈습니다.