**Pintos Project 1: User Program (1)**

담당 교수 : 문의현

학번 / 이름 : 20181256/김도현

개발 기간 : 22.10.27~22.10.31

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

Pintos의 OS 기능들 중 일부 기능이 구현 되지 않아 이 것들을 구현하여 user program을 system call 명령을 통해 실행 시킬 수 있도록 환경을 만드는 것을 목표로 한다.

-system call 명령을 하기 전에 입력된 명령들을 parsing 한 이후 이 나뉜 명령들을 stack에 쌓아 argument passing 을 수행한다.

- user memory access 로 kernel 영역이나 잘못된 영역에 메모리가 침범하지 않도록 권한을 설정해준다.

- system call 명령어를 system call handler 를 구현하여 user mode 와 kernel mode 간의 이동을 가능하도록 해준다.

- 추가적인 additional implementation을 통해서 새로운 system call 환경을 만들어 Fibonacci, max\_of\_four\_int 연산 등을 수행 할 수 있는 api를 생성한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

사용자가 명령어를 입력하면 이를 띄어쓰기를 기준으로 잡고 parsing을 해 실행 프로그램과 인자를 구분해서 스택이라는 자료구조에 저장한다. 그리고 다시 프로그램이 제대로 인자를 받을 수 있도록 해준다.

1. User Memory Access

User 영역과 kernel 메모리 영역의 구분을 확실히 하여 user 가 kernel의 메모리 영역에 침범, process의 비 정상(unmapped) 메모리 접근, file load 실패 등의 경우 kernel 이나 다른 실행중인 process에 피해 없이 강제로 종료해야 한다. 해당 thread를 종료할 때 parent thread에게 return -1 을 한다.

1. System Calls

User mode 에서 kernel mode 에 접근을 하여 kernel이 제공하는 서비스를 사용 할 수 있도록 system call 명령어에 대한 system call handler를 작성하는 것이다. 이 때 명령어는 1번에 작성한 argument passing으로 stack 에 쌓여 있는데 이를 하나씩 불러와 이용한다. 해당 프로젝트에서는 halt, exit, exec, wait, read, write 을 구현하고 추가적으로 fibonacci, max\_of\_four\_int 를 구현하여 어떻게 새로운 system call을 등록하는지 알 수 있다.

* 1. **개발 내용**
* **아래 항목의 내용만 서술 (기타 내용은 서술하지 않아도 됨.)**
* Argument Passing
  + 커널 내 스택에 argument를 쌓는 과정 설명

우선 입력한 명령어를 strtok\_r 을 통해서 띄어쓰기 별로 parsing을 해준다. 이후 이 argv 들을 stack에 하나 씩 저장을 하고 esp(stack의 pointer라고 생각하면 된다) 역시 저장하는 argv 대로 순차적으로 감소 시킨다. Stack은 높은 주소에서 낮은 주소로 쌓이기 때문에 esp를 감소 시키는 것이다.

중요 요소 중 하나는 word alignment 를 해야 하는 것이다. 기본적으로 컴퓨터는 메모리를 32bit 기준 4byte 씩 나누어 읽어들인다. 이때 해당 stack에 저장된 데이터의 크기가 4byte를 넘지 않으면 null 값을 채워 해당 데이터의 메모리를 4byte로 맞춰야 제대로된 passing이 가능하다. 즉 argv의 데이터 크기를 구한다음 4로 나누어 4의 배수가 되도록 esp를 조절하면서 만들어 주면 된다.

이후 null pointer sentinel로 다음 명령어와 구별되도록 경계를 나누어 주고 parsing을 하여 stack에 저장한 argv 들의 주소를 stack에 다시 저장하고 argc 및 return address 역시 stack에 저장해주면 argument 가 정상적으로 쌓일 수 있다.

* User Memory Access
  + Pintos 상에서의 invalid memory access 개념을 간략히 설명

크게 메모리는 kernel, user 두 영역으로 나뉘는데 각 프로세스들이 각 영역을 넘거나 혹은 kernel 영역에도 접근할 수 있다. User program이 이상한 pointer인 invalid pointer를 받을 시 원하는 영역이 아닌 이상한 메모리에 접근하게 되는 invalid memory access가 발생할 수 있다. 즉 user program이 invalid pointer 인 Null pointer, unmapped virtual memory, kernel address pointer 등을 받을 경우로 생각하면 된다.

* + Invalid memory access를 어떻게 막을 것인지 설명

그래서 해당 프로젝트에서는 exception handling을 하도록 해주어야 한다. 우선 user provided pointer 의 유효성 검사를 하고 dereference를 해준다. 이상한 pointer라면 page\_fault를 한다. 이는 userprog/exception.c 에서 page\_fault 함수 영역의 코드를 추가하면 된다. Null pointer 접근 여부는 단순히 null인지 아닌지 확인하고 Kernel address의 접근 여부는 is\_kernel\_vaddr 함수로 확인 가능하다.

그리고 system call handler 에서 is\_user\_vaddr 함수를 이용하여 반복적으로 해당 포인터가 원하는 주소를 가리키는지를 확인 해야 한다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명

만약 kernel 영역의 메모리를 user 영역에서 자유롭게 접근이 가능하다면 중요한 기능을 담고 있는 kernel 수정 등 컴퓨터에 좋지 않은 일들을 할 수 있어 일반적으로 os에서는 user mode와 kernel mode를 분리해 놓는다.

System call은 간단히 말해 user 영역과 kernel 영역이 구분 된 상태에서 kernel 에서 제공하는 중요한 서비스를 user에서도 사용할 수 있도록 하는 다리 역할을 하는 것이다.

* + 이번 프로젝트에서 개발할 시스템 콜에 대한 간략한 설명 (하나의 시스템 콜 당 최대 3문장으로 간략히 설명; 3문장을 넘길 정도로 길게 작성하지 말 것)

Halt(): shutdown\_power\_off 함수 만을 실행하여 pintos 자체를 종료시킨다.

Exit(): user\_program을 종료하고 이에 대한 상태(status) 를 kernel에 준다. Exit(0)은 정상 종료, 그 이외의 값들은 에러가 있음을 알 수 있다.

Exec(): child process를 만들고 새로운 process 의 pid를 return하는 함수이다. 이때 process\_execute를 실행하여 이를 수행한다.

Wait(): child process가 작업이 완료할 때 까지 기다리는 역할을 한다. 이때 semaphore 개념을 도입해 계속해서 child의 상태 여부를 체크해준다.

Read(): std input read 역할을 하는데 file descripter가 0일 때는 STDIN으로 input\_getc() 함수를 호출해서 글자를 읽어들인다.

Write(): read와 반대로 file descripter가 1일 때는 putbuf() 함수를 사용하여 write 기능을 수행해준다.

Fibonacci(): n을 input 값을 받아 해당 순서에 맞는 fibonacci 값을 던져준다.

Max\_of\_four\_int(): int 형 값인 4개의 숫자를 받아 이 중 가장 큰 값을 던져준다.

* + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명

예시로 thread/init.c 안의 Main() 에서 run\_action(argv)를 부른다.

여기서 다시 a->function(argv) 를 통해서 run\_task(char \*\*argv) 가 실행된다.

여기서는 다시 process\_wait 내에 실행하고 하는 process\_execute 부터 실행한다.

이 후 thread\_create으로 새로운 thread를 생성하여 tid를 만들어 start\_process로 해당 user\_program이 돌아간다.

Process scheduling 으로 프로세스가 실행되고 종료될 때 exit이 불리고 다시 process\_wait으로 return 된다.

Process\_wait에서는 부모 프로세스가 자식 프로세스가 끝날 때까지 status 값을 지속적으로 확인하여 대기하는 역할을 한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* **II. A.의 개발 범위를 포함하여 구현 내용에 대한 일정 작성**

10.27~10.28 : 사이버캠퍼스 과제 영상 확인 및 pintos 매뉴얼 필요한 부분 정독, pintos 실행 방법 숙지, argument passing 구현

10.28~10.29 : user\_memoery\_Access, system call 구현 및 시간 여유 시 additional system call 구현

10.29~10.31 : 전체적으로 필요한 부분 코드 수정 및 보고서 작성

* 1. **개발 방법**
* **II. B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**

1. Argument passing

* Userprog/process.c load 함수에 argument passing 코드를 추가한다.
* Userprog/process.c load 함수 윗부분에 File\_name 을 parsing 하고 해당 문자들을 setup\_stack()이 끝난 후 argv 들을 스택에 쌓는 코드를 추가한다.
* Userprog/process.c argv들을 쌓을 때 높은 주소에서 낮은 주소로 쌓고 esp를 이동하면서 해당 esp가 가리키는 stack의 메모리 주소에 argv들을 저장해야 한다.

1. User memory access

* Userprog/exception.c 에서 page\_fault() 부분에서 is\_kerne\_vaddr 함수로 kernel 접근 유무를 확인하여 맞을 시 exit(-1)를 return 하는 부분을 추가한다.
* Userprog/process.c process\_execute 에서 파일 이름이 null 인지 혹은 파일을 열었을 때(filesys\_open) null 인지 체크를 해준다.
* Userprog/syscall.c syscall 호출을 하기 전에 가장 앞에 조건문을 두어 우선적으로 is\_user\_vaddr 함수를 호출하여 user\_memory 유무를 계속해서 체크를 해준다.

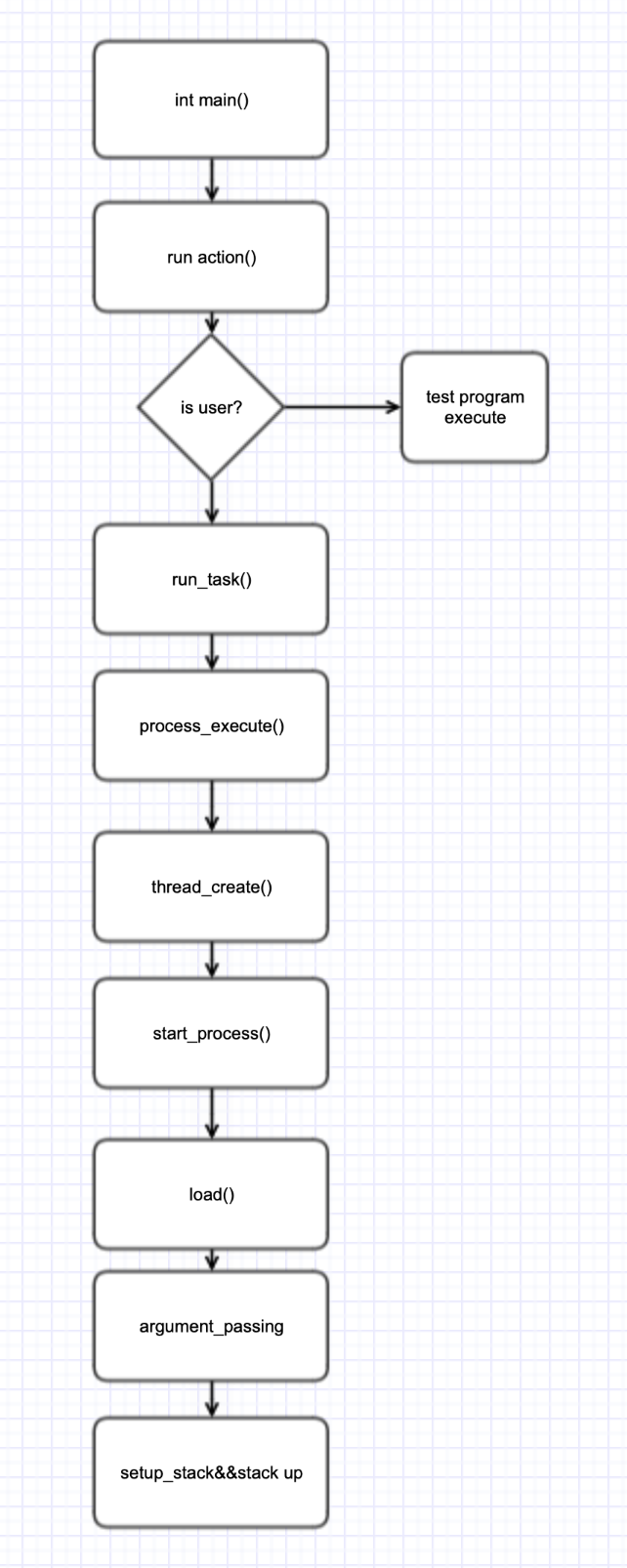
3. system call 및 additional system call

* Userprog/syscall.c syscall\_handler() 에 switch 문을 통해 system call number 에 맞는 system call 함수를 호출한다. (lib/syscall-nr.h 에서 해당 system call number 확인 가능하다.)
* Userprog/syscall.c exit(), exec(), wait(), read(), write() 함수 등을 구현하여 system call number에 따른 적절한 함수를 호출 해준다.
* Userprog/syscall.c 함수 수행 후 return 값 존재 시 eax에 저장한다.
* Userprog/process.c process\_wait 구현이 빠져 있어 semaphore 를 사용하여 부모 process가 자식 process 가 끝날 때 까지 기다리는 wait을 구현해야 한다.
* Src/thread/thread.h struct list, semaphore(child, memory), list\_elem 을 추가해준다.
* Src/thread/thread.c 에서는 thread.h 에서 구현한 것들을 초기화 시키는 과정이 필요하다.
* Userprog/syscall.c 위와 같은 방법으로 addititonal system call 인 Fibonacci, max\_of\_four\_int를 구현한다. Switch 문 및 해당 호출 함수 등을 구현하면 된다.
* Userprog/syscall.h 에서 추가된 함수에 대해 기입한다.
* Lib/syscall-nr.h system call number에 additional system call 역시 추가로 넣어준다.
* Lib/user/syscall.h 에 새로운 system call prototype 기입한다.
* Lib/user/syscall.c 에 syscall4()를 위의 syscall() 들을 기반으로 작성하고 system call api 역시 새로 작성을 해준다.
* Src/example additional.c 파일을 새로 만들어 기존 c 파일을 토대로 작성하고 해당 내용을 새롭게 makefile 에 additional.c 에 대한 부분 역시 추가해 새롭게 make 해준다.

1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

1. Argument Passing



1. User Memory Access

텍스트, 전자기기이(가) 표시된 사진

자동 생성된 설명

1. System Calls

텍스트, 컴퓨터, 전자기기이(가) 표시된 사진

자동 생성된 설명

* 1. **제작 내용**
* **II. B. 개발 내용의 실질적인 구현에 대해 코드 관점에서 작성.**
* **구현에 있어 Pintos에 내장된 라이브러리나 자체 제작한 함수를 사용한 경우 이에 대해서도 설명.**
* **개발상 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결책에 대해 설명.**

**코드에 대한 부분은 캡쳐 사진으로 대체하겠다.**

1. Argument Passing

Userprog/process.c

Load 함수에서 argument passing을 구현한다. Strtok\_r로 file\_name을 띄어쓰기로 구분을 하여 while문을 통해 parsing 하여 argv에 하나씩 저장한다. 가장 마지막 while문에서 argv[argc]에 null을 만들어주고 t.name 에 argv[0] 인 값을 복사해준다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Userprog/process.s

Load 함수의 setup stack 뒤 stack 구현이 끝난 후 stack을 보고서에 작성한 대로 구현한다. Stack에 쌓을 때마다 esp를 조절해주고 4 바이트마다 끊어지도록 Word alignment도 해준다. 이때 world alignment를 한 이후 Null pointer sentinel을 추가 한 다음 해당 argv 들의 주소를 넣어주고 argv[0] 의 시작 주소를 넣은 다음 argc 즉 문자열들의 개수, 마지막으로 return address를 넣어준다.

텍스트이(가) 표시된 사진

자동 생성된 설명

1. User Memory Access

Userprog/process.c

Process\_Execute() 함수에서 파일 이름을 strlcpy로 복사한다음 이를 strtok\_r 로 나눠 해당 파일을 filesys\_open으로 열어 해당 파일이 null 이거나 파일을 열었을 때의 null check를 해준다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Userprog/exception.c

Page\_fault() 함수 내에 is\_kernel\_vaddr 을 사용해 현재 kernel 메모리를 접근하고 있는지 혹은 user가 0 인지 아닌지 판별을 해주는 코드를 구현해준다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Userprog/syscall.c

가장 처음 f->esp 에 대해 is\_user\_vaddr를 실행한다. Is\_user\_vaddr로 현재 메모리가 user\_memory에 접근하고 있는지 확인할 수 있다. 만약 0이면 syscall\_exit(-1)를 실행한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

이후 case문 실행하고자 하는 system\_call 별로 해당 argv에 맞게 조건문을 추가하여 is\_user\_vaddr 함수를 통해 sys\_exit(-1) 실행여부를 확인해준다.

1. System Calls

* **이번 프로젝트에서 개발한 시스템 콜을 구현 관점에서 상세히 서술.**

텍스트이(가) 표시된 사진

자동 생성된 설명

Userprog/syscall.c

Switch 문을 통해 위에서 설명한 것처럼 is\_user\_Vaddr 로 유저 부분 메모리 접근 유무를 판별하고 이상 없으면 case number에 맞는 함수를 실행한다. 이때 lib/syscall-nr.h 에서 case number 확인이 가능하다.

Case가 SYS\_HALT 일 시 바로 shutdown\_power\_off() 함수를 실행하여 종료시킨다.

Case 가 SYS\_EXIT 일 시 printf 문을 통해 해당 이름과, status(exit\_or)를 출력해준다. 이후 thread\_exit으로 thread 를 종료한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Case 가 SYS\_EXEC 일 경우 process\_execute 함수를 실행한다.

Userprpog/process.c

텍스트이(가) 표시된 사진

자동 생성된 설명

Case 가 SYS\_WAIT 일 경우 process\_wait 함수를 실행한다. 여기서는 child\_tid와thread\_tid 가 같은지 다른지 지속적으로 while문을 돌며 확인한다. 만약 같다면 sema\_down으로 child\_sem를 내려 child 가 끝날때 까지 계속해서 기다리고 반면 메모리는 sema\_up으로 올려 메모리를 계속 남겨두어야 한다. 만약 child 가 죽으면 exit\_or 즉 status를 return 한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Process\_exit 에서는 process를 종료할 때 와 달리 sempahore를 child는 이용 가능하게 sema up, memory는 sema down 을 해줘야 한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Case 가 SYS\_READ 일 경우 sys\_Read 함수를 실행한다. 여기서 fd==0 이면 input\_getc 를 호출해 ‘\0’이 나올 때까지 stdin을 수행하고 ‘\0’ 나오기 전까지의 개수를 return 해준다. 만약 fd가 0이 아니면 -1을 return 한다.

텍스트, 화면, 스크린샷이(가) 표시된 사진

자동 생성된 설명

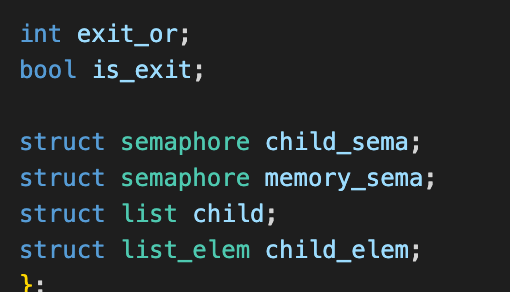
Case 가 SYS\_WRITE 일 경우 sys\_write 함수를 실행한다. 만약 fd==1 이면 putbuf 함수를 통해 stdout를 구현한다. 이때 size를 return 해준다. fd가 1이 아니면 return -1을 한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

threads/thread.h

현재 wait, signal을 하기 위해서는 process\_exit, process\_wait 에서 semaphore 를 사용하여 이를 수행하고 있다. 여기서 semaphore 를 child, memory 2개로 두어 각각 이름 그대로 child 에 대한 lock, memory에 대한 lock 이다. 이때 child\_elem 의 값과 exit\_or 은 exit 했을 때의 상태(status)를 의미한다.



Threads/thread.c

Thread\_create 함수에서 list\_push\_bac으로 running\_thread에 child\_elem을 집어 넣고 semaphore 인 child\_sema, memory\_sema를 초기화 한는 과정을 진행한다.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

Init\_thread 함수에서는 list\_init 함수로 thread child를 초기화 시켜준다.

**텍스트이(가) 표시된 사진

자동 생성된 설명**

Userprog/syscall.h

함수를 추가 해준 것들의 프로토타입을 기입한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

1. Additional System calls

* **새로운 시스템 콜(fibonacci, max\_of\_four\_int)을 구현하기 위해 수정하거나 작성한 코드에 대해 서술**

Userprog/syscall.c

syscall\_handler 함수에 위에서 구현한 것과 같이 case 문에 Fibonacci, max\_of\_four\_int 를 추가 number로 기입한다. 이후 is\_user\_vaddr에서 메모리에 대해 문제가 없을 경우 Fibonacci, max\_of\_four\_int 함수를 구현해준다. Fibonacci 함수는 배열 크기를 3으로 지정하여 % 연산을 사용해 배열 크기를 최소화 하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Lib/syscall-nr.h

위의 코드들을 참고하여 additional(Fibonacci, max\_of\_four\_int) system call number를 추가한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Lib/user/syscall.c

Syscall1(), …syscall3()을 참고하여 syscall4() 를 새롭게 구현한다. 이후 system call 을 수행하는 api 함수를 구현하다. 이떄 Fibonacci 는 인자가 1개만 받기 떄문에 syscall1() 를 사용하고 max\_of\_four\_int 는 인자를 4개를 받기 때문에 새로 구현한 syscall4() 를 사용한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

Lib/user/syscall.h

헤더파일에 additional System call 의 api 프로토타입을 적어준다.

텍스트이(가) 표시된 사진

자동 생성된 설명

Userprog/syscall.h

Src/userpog/syscall.c 에서 작성한 Fibonacci, max\_of\_four\_int 에 대한 함수 역시 헤더 파일에 프로토타입을 추가해준다.

(위에 코드 추가하였다.)

Src/examples/additional.c

Vi additional.c 로 새롭게 c 파일을 작성해야 한다. 여기서는 input 인자에 맞게 Fibonacci, max\_of\_four\_int 를 계산하여 print 해주는 c 파일이다. 이때 argc를 통해 만약 인자가 원하는 개수대로 들어오지 않았다면 exit\_false 정상 출력되었다면 EXIT\_SUCCESS 를 return 한다.(이때 계산 시 정수현으로 바꿔야 하는데 atoi 로 char을 int 형을 변환한다.)

텍스트이(가) 표시된 사진

자동 생성된 설명

Src/examples/makefile

Example에 추가된 파일은 additional.c 를 새롭게 생성 할 수 있도록 다른 파일들처럼PROGS 에 추가로 적어준다. 이후 다시 make를 하면 된다.

텍스트이(가) 표시된 사진

자동 생성된 설명

* 1. **시험 및 평가 내용**
* **fibonacci 및 max\_of\_four\_int 시스템 콜 수행 결과를 캡처하여 첨부.**

**해당 캡쳐본은 Fibonacci, max\_of\_four\_int를 실행한 것이다.**

**55,62 모두 정상 출력 됨을 알 수 있다.**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

**Make grade 결과를 보면 해당 프로젝트의 요구사항인 functionality 13개, robustness 8개 모두 충족 함을 볼 수 있다.**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

****

**텍스트이(가) 표시된 사진

자동 생성된 설명**