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

담당 교수 : 김영재 교수님

조 / 조원 : 20201654 최호진

개발 기간 : 2024. 09. 14 – 2024. 09. 22

1. **개발 목표**

* PintOS에서 유저 프로그램을 실행할 수 있도록 시스템 콜과 관련한 부분을 구현해야 한다. 먼저 Argument를 적절히 Parsing하여 x86-64 Calling Convention에 맞게 스택을 쌓고, 이와 함께 시스템 콜 함수를 구현하고, 관련된 기타 함수들을 같이 고쳐주어 프로그램이 시스템 콜을 통해 OS와 상호작용할 수 있도록 만드는 것이 이번 프로젝트의 목표이다.

1. **개발 범위 및 내용**
   1. **개발 범위**
2. Argument Passing

입력받은 Argument들을 띄어쓰기 단위로 token으로 만들어 나누고, 총 argument 개수(argc)를 센다. 그 다음 이 argument들을 x86-64 calling convention에 맞춰 스택을 쌓아준다.

1. User Memory Access

비정상적으로 kernel address에 접근 또는 핀토스에서 허용하는 메모리 범위가 아닌 주소로 접근하려고 할 때, 취약점 및 오류가 발생할 수 있으므로, 이를 해결하기 위해 비정상적인 주소에 접근하지 않는지 확인하는 코드를 삽입해주었다.

1. System Calls

Pintos Manual 3.3.4 System Call에 정의된 각 System call의 역할에 따라 알맞게 구현해 준다. 이번 프로젝트에서는 halt, exit, exec, wait, read, write 시스템 콜과 system call handler를 구현하였고, 추가적으로 프로젝트에서 요구하는 fibonacci, max\_of\_four\_int 시스템 콜도 구현하였다.

* 1. **개발 내용**
* Argument Passing
  + **커널 내 스택에 argument를 쌓는 과정 설명**

1. userprog/process.c의 process\_execute 함수에서 첫번째 공백 위치를 찾아 file name과 나머지 부분으로 나눈다.
2. load 함수에서 strtok\_r 함수를 사용하여 나머지 부분을 공백(“ “) 단위로 나누고, argument의 개수를 센다. 그 다음 x86-64 calling convention에 따라 stack을 쌓기 위해 새롭게 선언한 함수(stack\_constructor)를 호출하여 stack에 쌓아 준다. 먼저, argument들을 역순으로 순서대로 쌓아주고, 필요하다면 마지막에 4 word alignment를 위해 필요한만큼 채워준다. 그 다음, 0으로 4만큼 채워주고, argument들이 저장된 주소를 마찬가지로 역순으로 쌓아준다. 마지막으로 argv[0]의 주소, rgc, return address를 쌓아준다. 이때 stack pointer (esp)는 return address를 가리키도록 한다.

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

PintOS에서 허용하는 메모리 범위(PHYS\_BASE~0xc0000000) 밖의 주소로 접근 하려고 할 때 이는 invalid memory access가 된다. 또한, User program이 User가 접근 가능한 메모리 영역을 넘어서 Kernel memory 주소로 들어가게 되면 Kernel Panic과 같은 오류가 발생한다.

* + **Invalid memory access를 어떻게 막을 것인지 설명**

PintOS 프로젝트에 비정상적인 메모리 접근을 확인하는 함수가 미리 선언되어 있다. (threads/vaddr.h에 is\_user\_\_vaddr, is\_kernel\_vaddr) 이를 이용해서 접근하는 주소가 올바른 주소인지 확인하여, 만약 올바르지 않은 주소로 접근하려고 할때 Exit(-1)로 비정상적인 종료를 시켜, 취약점 발생 및 오류를 막는다.

* System Calls
  + **시스템 콜의 필요성에 대한 간략한 설명**

운영체제는 상호불가침의 영역, User Program과 Kernel으로 나뉜다. File read, write와 같이 user program 단독으로 실행할 수 없는 기능을 user program에서 수행하기 위해 kernel 영역의 기능을 요청해야 하는데, 이를 System Call을 이용해 호출하게 된다.

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

1. Halt: 미리 정의되어 있는 shutdown\_power\_off() 함수를 호출해 PintOS를 종료시키는 요청을 보낸다.
2. Exit: 실행중인 User Program을 종료하고 해당 thread의 status를 커널에 알린다. 정상적인 종료일 때는 status가 0이고, 이외의 경우에는 0이 아닌 값으로 나타낸다.
3. Exec: process\_execute 함수를 실행하고 해당 함수의 반환 값을 반환한다. Process\_execute 함수에서는 argument로 입력받은 file name에 해당하는 프로세스(thread)를 자식 프로세스로 새로 fork하여 실행한다.
4. Wait: pid를 argumnet로 받아 process\_wait 함수를 실행하고 해당 함수의 반환 값을 반환한다. Process\_wait 함수에서는 전달받은 pid의 종료를 기다리고, exit status 값을 반환한다.
5. Read: 이번 프로젝트에서는 file descriptor 중 STDIN의 입력만을 받아 buffer에 작성한다.
6. Write: 이번 프로젝트에서는 file descriptor 중 STOUDT에 내용을 출력한다. 출력한 byte 수를 반환한다.
7. fibonacci: 입력 받은 정수(0 이상의 정수)에 해당하는 피보나치 수열의 값을 반환한다.
8. max\_of\_four\_int: 4개의 정수를 입력 받아 그 중 가장 큰 값을 반환한다.
   * **유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명**

유저 레벨에서 시스템 콜 API를 호출하면 mode bit에 이를 표시하여 kernel 모드로 진입하고 syscall handler로 이동한다. stack에 System call number와 주어진 인자들이 쌓이게 되고 stack pinter(esp)가 가리키는 address에는 시스템 콜 number가 저장되어 있는데, system call handler에서 해당 number를 확인해 적절한 system call을 호출할 수 있도록 테이블에서 찾아 해당 함수를 호출한다. 시스템 콜에서 작업이 끝나면 다시 유저 레벨로 돌아오고 mode bit에도 표시한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**
   * 9/14 ~ 9/16: Argument Passing 구현
   * 9/16 ~ 9/17: User Memory Access 구현
   * 9/17 ~ 9/20: System Calls 및 Additional System Calls 구현
   * 9/20 ~ 9/22: 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)**
* Argument Passing

userprog/process.c 파일 내 process\_execute 함수의 인자로 들어오는 file\_name 문자열에서 첫번째 공백 위치를 찾아 strtok\_r 함수를 이용해 입력 받은 command의 종류(echo, cat 등)와 나머지 부분으로 나누어 command의 종류를 찾는다. 같은 파일 내 load 함수에서는 마찬가지로 strtok\_r 함수를 이용하여 공백 단위로 argument들을 나눠주고 argument의 전체 개수를 센다. 이렇게 나눈 argument들과 이들의 개수를 이용해 stack을 쌓기 위해 새롭게 선언한 stack\_constructor 함수에서 x86-64 calling convention을 따라(pintos manual 참고) stack에 순서대로 쌓아 준다. 이때 stack pointer(esp)에 memcpy를 통해 차례대로 넘겨주고 alignment을 위한 null문자, 주소, argument 개수, return address를 차례대로 쌓게 된다. 필요한 모든 stack의 원소들을 쌓고 나면 esp는 return address를 가리키게 된다. 이렇게 쌓은 stack은 이후 system call 단계에서 활용하게 된다.

* User Memory Access

userprog/exception.c 파일 내의 is\_kernel\_vaddr 함수를 이용해 User program이 커널 메모리 영역을 접근하려고 하는지 확인해 접근하지 못하도록 막는다. 만약 접근하려고 한다면 Exit(-1)로 프로세스를 종료한다.

userprog/process.c 파일 내의 process\_execute 함수에서는 filesys\_open의 반환값이 NULL이라면 (파일 읽기에 실패했다면) TID\_ERROR를 반환해 예외 처리한다.

System call 단계에서 Invalid memory access를 막기 위해 threads/vaddr.h 파일에 정의된 is\_user\_vaddr과 is\_kernel\_vaddr 함수를 이용하여 주어진 메모리 주소가 알맞은 범위 안에 있는지 확인하여 만약 알맞은 범위가 아니라면 exit(-1)로 프로세스를 종료한다.

* System Calls

userprog/syscall.c 파일의 syscall\_handler 함수에서 switch문으로 각 system call number마다 수행할 작업을 정의한다. 이 때 앞서 User memory access에서 설명했듯이 각 system call을 호출하기 전에 접근하는 address가 접근하면 안되는 kernel 메모리에 접근하려고 하지 않는지 확인하여 오류를 방지한다. 각 함수의 반환값은 eax register에 저장한다. 또한 기존에 정의되어 있지 않은, 추가적으로 구현해야 하는 추system call인 fibonacci와 max\_of\_four\_int에 대한 number를 할당해주어야 하므로 lib/syscall-nr.h 파일에 추가해준다.

또한, max\_of\_four\_int는 4개의 인자를 요구하는데 이 system call에 대한 정의가 되어있지 않으므로 lib/user/syscall.c에 syscall4 함수를 새롭게 선언하고, 정의해준다.

그리고 새롭게 정의된 system call이 제대로 작동하는지 확인하기 위한 예제 파일을 examples 폴더 내에 새롭게 작성하여준다.

마지막으로, Synchronization을 위해 thread 구조체에 child process(thread)를 관리하기 위한 child list와 semaphore(mem\_lock, child\_lock)를 추가하고 이를 userprog/process.c 파일의process\_exit, process\_wait 함수에서 적절히 사용해 원하는 결과가 정상적으로 결과가 출력될 수 있도록 한다.

1. **연구 결과**
   1. **Flow Chart**
2. Argument Passing

A diagram of a software system

Description automatically generated

1. User Memory Access

A diagram of a software process

Description automatically generated

1. System Calls

A diagram of a process

Description automatically generated

* 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.**

1. Argument Passing

II.B의 Argument Passing에서 설명했듯이, userprog/process.c 파일 내 process\_execute 함수에서 command의 종류를 찾는다.

A computer screen shot of text

Description automatically generated

그 다음 load 함수에서는 이후 arugment들을 띄어쓰기 단위로 파싱하고, 총 argument 개수를 센다.

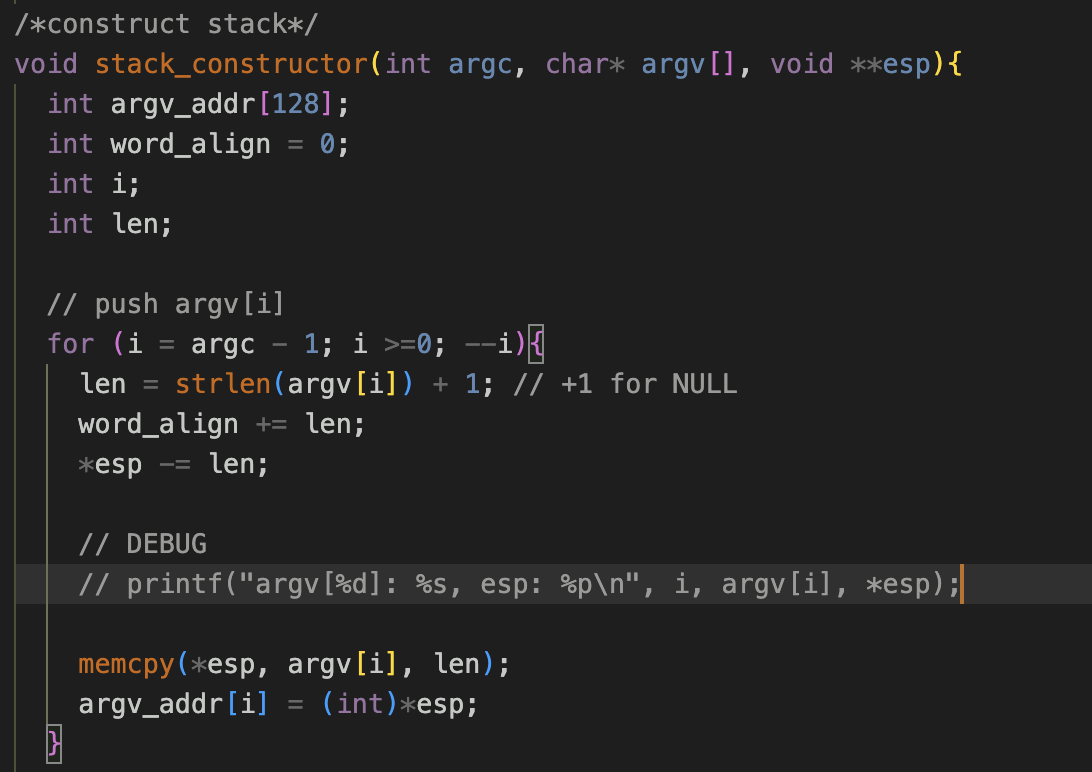
A screen shot of a computer program

Description automatically generated

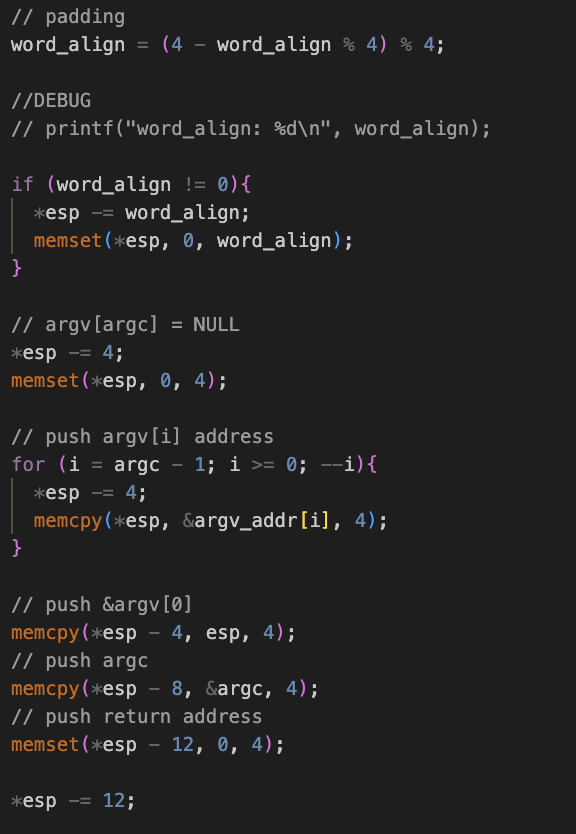
이후 이렇게 파싱한 argument token들이 들어있는 argv 배열과 총 개수인 argc, 그리고 stackpointer(esp)를 stack을 쌓기 위해 새롭게 선언한 stack\_constructor 함수에 인자로 넘겨 stack을 쌓는다.

A screen shot of a computer

Description automatically generated



먼저, 파싱된 argument들을 역순으로 쌓는다.



그 다음 alignment가 필요하다면 필요한만큼 0으로 memory를 채워주고, 0 4개를 추가로 채워준다. 그 다음 위에서 쌓은 argument들이 저장된 주소들을 stack에 쌓는다. 마지막으로,a argv[0]의 주소, argument 개수, return address를 차례대로 쌓고 ,stack pointer가 return address를 가리키도록 한다.

1. User Memory Access

* userprog/process.c

process\_execute 함수에서 command\_name이 invalid 메모리에 access 하는지 확인하는 코드가 없어서 이를 추가했다. Return 값이 NULL이라면 TID\_ERROR 코드와 함께 곧바로 return하여 종료하도록 하였다.

A computer screen shot of text

Description automatically generated

* userprog/syscall.c

syscall\_handler 함수에서 switch문을 이용해 어떤 system call을 호출해야하는지 확인하고 이를 호출하기 전에 invalid한 메모리 접근이 있지 않은지 확인을 해야한다. 따라서, check\_user\_vaddr 함수를 새롭게 선언하여, exit, exec, wait, read system call을 호출하기 전에 user\_vaddr가 아닌 kernel memory에 접근하지 않는지 확인하고, 만약 접근한다면 곧바로 exit(-1)과 함께 종료되도록 하였다.

텍스트, 스크린샷, 소프트웨어, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

1. System Calls

* **이번 프로젝트에서 개발한 시스템 콜을 구현 관점에서 상세히 서술.**
* userprog/syscall.c

[1] syscall\_handler

**텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명**

syscall\_handler 함수에서는 함수 argument로 받은 intr\_frame의 esp (stack pointer)가 가리키는 메모리에 저장된 값을 확인해 어떤 시스템 콜을 호출해야 하는지 확인하고 알맞게 시스템 콜을 호출해준다. 이때 switch문을 활용해주었으며, 추가적으로 시스템 콜 중 비정상적으로 kernel memory에 접근하면 안되는 시스템 콜의 경우 시스템 콜의 argument가 user\_vaddr가 아닌 kernel memory에 접근하지 않는지 확인해주고, 만약 kernel 메모리에 접근한다면 exit(-1)로 강제 종료한다.

다음으로는 각 시스템 콜의 기능에 대해 설명한다. 각 시스템 콜은 핀토스 매뉴얼(p.29)을 참고해 작성하였다.

[2] Halt()

A black background with white text

Description automatically generated

Halt에서는 미리 선언되어있는 shutdown\_power\_off()를 호출하여 현재 프로그램을 종료한다.

[3] Exit(int status)

A computer screen with white text

Description automatically generated

Thread\_exit()을 호출해 현재 쓰레드(프로세스)를 종료한다. 또한, exit하기 전에 인자로 전달받은 status를 출력하고 현재 쓰레드의 exit status를 전달받은 status로 업데이트하고 종료한다.

[4] Exec(const char \*cmd\_line)

A black screen with white text

Description automatically generated

process\_execute를 실행해 새 thread를 생성하고 이 함수의 반환값을 반환한다. 정상적으로 쓰레드가 실행되었다면, 해당 쓰레드의 pid를 반환, 비정상적으로 종료되었다면 -1을 반환한다.

[5] Wait(int pid)

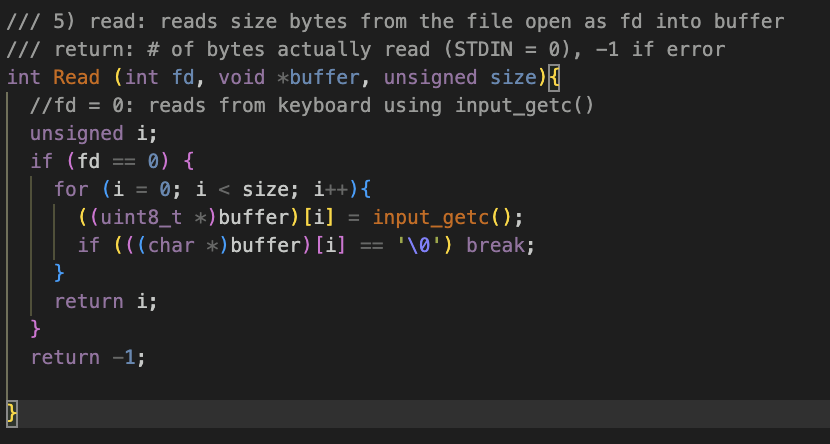
A black screen with white text

Description automatically generated

Exec와 유사하게 process\_wait 함수를 호출하고, 이 함수의 반환 값을 반환한다. Process\_wait 함수에서는 인자로 전달받은 pid가 종료될 때까지 wait하고 reaping하는 역할을 하는데, 이때 semaphore를 이용해 wait하고, semaphore가 1(up)되면 exit status를 반환한다.

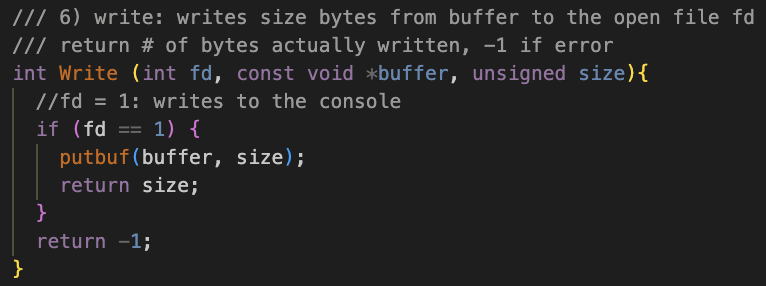


[6] Read(int fd, void \*buffer, unsigned size)



이번 프로젝트에서는 file descriptor가 0일 때(STDIN)일 때에만 버퍼에 저장된 값을 읽고, 전체 읽을 길이를 반환한다. 이외의 file descriptor에 대해서는 -1를 반환한다.

[7] Write(int fd, const void\* buffer, unsigned size)



이번 프로젝트에서는 file descriptor가 1일 때(STDOUT)일 때에만 버퍼에 저장된 값을 putbuf 함수를 이용해 출력한다. 전체 버퍼 사이즈를 반환하고, 이외의 file descriptor에 대해서는 -1를 반환한다.

* threads/thread.h

기존 thread 구조체에 User program을 위한 새 변수들을 추가해주어야한다.

A screen shot of a computer

Description automatically generated

Child thread(process)들을 관리하기 위한 list와 list\_elem type의 변수를 추가하고, Synchronization을 위한 semaphore인 mem\_lock과 child\_lock을 추가한다.

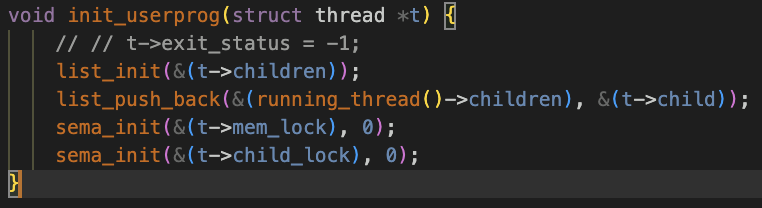
여기서 선언한 변수들은 thread(thread.c 파일의 init\_thread 함수)를 만들고, reaping할 때(process\_wait, 위에 Wait 부분 참고) 이용한다.

* threads/thread.c

새 thread를 만들 때 list에 추가하고 semaphore를 초기화한다.

A screen shot of a computer program

Description automatically generated



1. Additional System calls

* **새로운 시스템 콜(fibonacci, max\_of\_four\_int)을 구현하기 위해 수정하거나 작성한 코드에 대해 서술**
* lib/syscall-nr.h

syscall-nr.h에 추가적인 시스템 콜을 위한 넘버를 enum에 추가한다.

A computer screen shot of a program

Description automatically generated

* lib/user/syscall.c

현재 system call은 인자가 4개인 system call을 처리해주지 못하므로, lib/user/syscall.c 파일에 인자가 4개인 system call에 대해 처리할 수 있도록 코드를 작성해주어야 한다.

A computer screen with text and numbers

Description automatically generated

* userprog/syscall.c

그 다음, userprog/syscall.c 파일의 syscall\_handler에 fibonacci와 max\_of\_four\_int에 대해 처리할 수 있도록 switch문의 case에 추가하고, 함수 내용을 작성해준다.



텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

* examples/additional.c

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

Fibonacci, max\_of\_four\_int system call을 확인하기 위한 c파일을 작성하였다.

* examples/Makefile

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

새롭게 작성한 파일도 같이 컴파일 될 수 있도록 Makefile을 수정하였다.

* 1. **시험 및 평가 내용**
* 먼저, prj1에서 확인하는 test case들에 대해서 make grade로 실행 후 나온 결과는 아래와 같다.
* **Test Cases(Functionality)**

**텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

**텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

**폰트, 텍스트, 스크린샷, 타이포그래피이(가) 표시된 사진

자동 생성된 설명**

**텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

* **Test cases(Robustness)**

**텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

****

**텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명**

* **Additional system call**

****

위와 같은 커맨드로 example file을 실행하였고, 결과는 아래와 같다.

**텍스트, 스크린샷, 메뉴, 폰트이(가) 표시된 사진

자동 생성된 설명**