**Project 2: User Program**

1조

권 민철<ventania1680@gmail.com>

김 준영<john5910@naver.com>

이 장원<rex94@naver.com>

---- PRELIMINARIES ----

[5월 11일] 오후 2시에 학교에 다같이 모여서 프로젝트 시작하였습니다. 5월 7일 보충강의 들은 것 바탕으로 Part 1 : **Argument Passing** 구현을 시작했습니다.

[5월 12일]오후 2시에 학교에서 다같이 다시 모여서 Part 1 : **Argument Passing** 을 마저 끝내고 process\_wait()를 대기시간을 1000000짜리 for문으로 두고 hex\_dump() 를 통해 구현을

확인하였습니다.

[5월 13일] 수업 끝난 후 저녁 7시부터 밤 12시까지 권민철 학우가 Part 2: **System Call** 의**process\_wait()** 구현을 시작하였습니다. 이장원 학우와 김준영 학우는 용무를 끝내고 9시부터 함께 하였습니다. system call handler와 필요한 몇 개의 system call을 구현하였습니다.

[5월 14일] 수업 끝난 후 저녁 7시부터 새벽 1시까지 권민철 학우와 김준영 학우가 Part 2: **System Call** 의 **process\_wait()** 구현을 계속 하였습니다. 이장원 학우는 용무를 끝내고 9시부터 함께 하였습니다. process\_wait() 이 제대로 호출되어서 syscall 이 발생하는 것을 확인하고 make check를 통해 pass가 뜨는 것을 확인하였습니다.

[5월 15일] 수업 끝난 후 저녁 7시부터 권민철 학우가 Part 2: **System Call** 의exit, write, halt, wait, create, remove, open, read, write 등 syscall function을 구현 했습니다. 이장원 학우와 김준영 학우는 용무를 끝내고 9시부터 함께 하였습니다. 구현 결과 76개중 72개 pass 가 뜨는 것을 확인하였습니다.

Contribution

권 민철 :

김 준영 :

이 장원 :

======ARGUMENT PASSING ======

---- DATA STRUCTURES ----

*Process.c의*

*Process\_execute() 에서*

char \*token = (char\*)malloc(sizeof(char)\*128);

char \*ptr;

\*token 은 filename을 token 단위로 분리하여 filesys\_open()을 호출해주기 위한 변수이다.

\*ptr은 strtok\_r의 3번째 parameter로 들어가기 위해 만든 변수이다.

*load() 에서*

char \*token[128];

char \*ptr;

int tok\_cnt = 0;

\*token[128] 은 file name 을 공백을 기준으로 분리해서 저장하기 위해 만들어 주었고

\*ptr은 strtok\_r()에서 첫번째 parameter에 NULL을 주었을 때 이전까지 token화한 위치를

저장해주기 위해 선언하였습니다.

tok\_cnt는 배열 index를 위해 선언하였습니다.

---- ALGORITHMS ----

*Process.c의 load()에서*

token[tok\_cnt] = strtok\_r((char\*)file\_name, " ", &ptr);

while(true) {

if(token[tok\_cnt] == NULL)

break;

tok\_cnt++;

token[tok\_cnt] = strtok\_r(NULL, " ", &ptr);

}

file\_name = token[0];

file = filesys\_open (file\_name);

strtok\_r이용해서 parsing 하여서 입력 받은 작업을 호출해주고,

while(--i >= 0) {

unsigned int tok\_len = strlen(token[i]);

\*esp -= tok\_len;

memcpy(\*esp, token[i], tok\_len);

argv\_addr[i] = (uint32\_t)(\*esp);

\*esp -= 1;

/\*if((unsigned int)(\*esp) % 4 != 0)

\*esp -= (unsigned int)(\*esp) % 4;\*/

}

\*esp -= (unsigned int)(\*esp) % 4;

\*esp -= 4;

memset(\*esp, 0, sizeof(uint32\_t));

i = tok\_cnt;

while(--i >= 0) {

\*esp -= 4;

\*(uint32\_t\*)(\*esp) = argv\_addr[i];

}

\*esp -= 4;

\*(uint32\_t\*)(\*esp) = (uint32\_t)(\*esp) + 4;

\*esp -= 4;

\*(uint32\_t\*)(\*esp) = tok\_cnt;

\*esp -= 4;

memset(\*esp, 0, sizeof(uint32\_t));

/\* Start address. \*/

\*eip = (void (\*) (void)) ehdr.e\_entry;

success = true;

//hex\_dump(\*esp, \*esp, 100, 1);

done:

/\* We arrive here whether the load is successful or not. \*/

file\_close (file);

return success;

}

에서 esp에 4씩 크기를 맞춰 감소시키면서 stack에 맨 뒤에 것 부터 push 해주면서 반대 순서로 들어가게 해준다.

*syscall.c 에서*

void

check\_user\_vaddr(const void \*vaddr)

{

if(!is\_user\_vaddr(vaddr))

exit(-1);

}

is\_user\_vaddr 을 통해 system call이 발생할 때 stack esp 영역이 kernel 영역을 참조 하지 않는지 확인하여 overflow가 발생하지 않게 해준다.

======SYSTEM CALLS=======

---- DATA STRUCTURES ----

B1

Void check\_user\_vaddr(const void \*vaddr) : system call이 발생할 때 stack esp 영역이 kernel 영역을 참조 하지 않는지 확인하여 overflow가 발생하지 않게 해준다.

Void halt(void) : shutdown\_power\_off()를 이용하여 pintOS를 끝낸다.

Void exit(int status) : 현재 쓰레드(유저프로그램)을 끝내고 status를 저장시킨다. wait하고 있는 프로세스가 있다면 전달될 것이다.

pid\_t exec(const char \*cmd\_line) : 받은 명령어 라인을 인자로 프로세스를 실행

int wait(pid\_t pid) : 자식프로세스의 status를 받을 때 까지 기다린다.

Bool create(const char \*file, unsigned initial\_size) : 새로운 파일을 받은 사이즈 만큼 초기화하여 만든다.

Bool remove(const char \*file) : 파일을 삭제

Int open(const char \*file) : 파일을 연다

Int filesize(int fd) : 파일의 크기를 전달해준다.

Int read(int fd, void \*buffer, unsigned size) 받은 사이즈만큼 파일에서 읽어 버퍼에 저장

Int write(int fd, const void \*buffer, unsigned size) 버퍼에 있는 것을 사이즈만큼 파일에 저장

Void seek(int fd, unsigned position) : 파일에서 초기부터 포지션만큼 옮긴다.

Unsigned tell(int fd) : 파일의 현재 포지션을 알려준다.

Void close(int fd) : 파일을 닫는다.

B2: Describe how file descriptors are associated with open files. Are file

descriptors unique within the entire OS or just within a single process?

파일을 오픈 할 때 filesys\_open()함수를 써서 파일 디스크립터를 저장한다. 이 때 저장한 파일디스크립터 정보를 따로 저장해 두지 않으면 나중에 불러서 쓰지 못하므로 현재 쓰레드(thread\_current)에 파일 디스크립터를 저장해 준다. 그러므로 이는 전체 os가 아닌 프로세스에 저장.

---- ALGORITHMS ----

B3: Describe your code for reading and writing user data from the kernel.

Read 와 write를 사용할 때 메모리 주소를 stack에 저장한다. 그리고 그 메모리를 공유함으로써 kernel에서 사용할 때 공유된 메모리에서 주소를 찾아내어 read 와 write를 한다.

B4: Briefly describe your implementation of each system call.

SYS\_HALT : shutdown\_power\_off()함수를 이용하여 핀토스 종료

SYS\_EXIT: 쓰레드의 이름을 출력한 후 exit\_status를 현재 status로 바꿔준 후 thread\_exit()

SYS\_EXEC : process\_excute()를 사용하여 프로세스 실행

SYS\_WAIT : process\_wait()를 사용하여 자식 status 기다림

SYS\_CREATE : 파일이 없다면 종료

SYS\_REMOVE : 파일 이름을 받으면 파일을 없앤다.

SYS\_OPEN : 파일 이름을 받아 파일을 열고 파일 포인터를 저장

SYS\_FILESIZE : 현재 쓰레드의 파일 디스크립트를 받아 file\_length()를 불러 파일 크기 리턴

SYS\_READ : 버퍼를 받아 file\_read()를 호출하여 현재 쓰레드의 파일 디스크립터를 이용하여 파일을 읽는다

SYS\_WRITE : file\_write()를 호출하여 현재 쓰레드의 파일 디스크립터를 이용하여 파일에 버퍼를 쓴다

SYS\_SEEK : file\_seek()를 호출하여 포지션을 탐색한다.

SYS\_TELL : file\_tell()를 호출하여 현재의 포지션을 알려준다.

SYS\_CLOSE : 현재 쓰레드의 파일 디스크립터를 확인하고 fp에 저장한다음 NULL로 바꿔준 후 file\_close()를 호출하여 닫는다.