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

담당 교수 : 김영재

조 / 조원 : 고동헌

개발 기간 : 2023-09-19~2023-10-01

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

User program이 실행되기 위해선 실행을 위한 Argument들을 User stack에 넣어줘야 한다. 또한, User Program에서 Process의 생성, Write 작업 등의 Restricted Operation을 할 수 있도록 System Call API 를 제공하고, 이러한 System Call 처리를 OS 에서 해줄 수 있어야 한다. 하지만 현재 Pintos는 Argument passing, System Call handler등의 구현이 돼 있지 않다.

이 프로젝트의 목표는 위의 사항들을 구현해 Pintos OS 상에서 User Program을 실행할 수 있도록 만들고, fibonacci, max\_of\_four\_int 의 추가적인 System call 을 만드는 것이다.

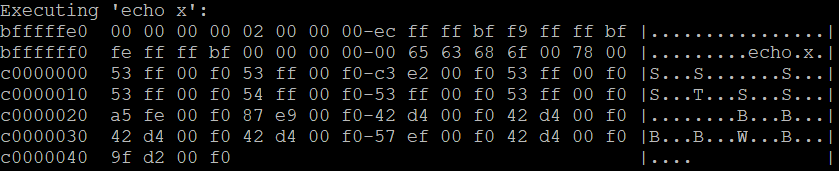
1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. **Argument Passing**

pintos에서 User Program을 실행시키면, process\_execute함수에 User가 입력한 Command Line이 Parsing 되지 않고 넘어오게 된다. 우선 thread\_create 함수를 호출하기 전, 이 command line에서 실행파일의 이름을 추출한 후 thread\_create 함수의 인자로 넘겨주었다.

실질적인 command line parsing과 argument passing 은 load 함수에서 구현하였다. load에서 char\*argv[], int argc 지역 변수를 선언한 뒤 새롭게 정의한 parse\_commandLine함수에서 strtok\_r 함수를 이용해 실행 파일의 이름과 Argument들을 argv에 순차적으로 넣어주었다. filesys\_open 함수가 성공적으로 return 된다면, setup\_stack 함수로 stack에 page를 한 개 할당해준 뒤 argument\_passing 함수를 호출한다. argument\_passing 함수에선 argv변수를 이용해 user stack에 data를 쌓아준다. 이 때 80x86 calling convention에 맞도록 argument의 길이와 해당 argument의 주소, word alignment를 고려해 stack을 구성했다. ‘echo x’를 실행했을 경우 hex\_dump 결과는 다음과 같다.



1. **User Memory Access**

Kernel이 System Call을 처리할 때 User 가 전달한 포인터 값을 통해 Memory를 Access 하는 경우가 많다. 이 때 해당 포인터 값이 kernel address space를 가리키거나, mapping 돼 있지 않은 virtual address를 접근하는 경우 해당 포인터를 Invalid Pointer라고 한다. 포인터 값이 NULL 인 경우 역시 Invalid Pointer이다.

kernel 에선 User Program이 이러한 Invalid Pointer를 전달하는 경우, 해당 process를 종료시켜야 한다. 따라서 System call handler 에서 is\_user\_vaddr, pagedir\_get\_page 등의 pintos 내장 함수를 이용해 Invalid Pointer를 확인하고, 만약 Invalid Pointer가 있는 경우 exit(-1) 함수를 호출해 System call을 발생시킨 process의 exit status 를 -1로 set 하고 thread\_exit() 함수를 호출해 process를 종료 시켰다.

이 외에도, user program이 실행하며 kernel 영역에 접근해 page fault가 발생했을 경우, page fault handler에서 exit(-1) 함수를 호출해 해당 process를 종료하였다.

1. **System Calls**

이번 프로젝트에선 기본적으로 6개의 System Call을 구현하고, 추가적으로 2개의 System call을 구현해야 한다. 구현한 System Call 과 결과는 다음과 같다.

**(i) halt**

- 내부적으로 shutdown\_power\_off 함수를 호출해 pintos를 종료시킨다.

**(ii) exit**

**-** 이 함수의 구현을 통해, 실행 중인 process를 종료 시키고 exit status를 kernel에게 전달할 수 있다. 또한, termination message를 띄워준다.

**(iii) exec**

**-** 이 함수의 구현을 통해, user program이 child process를 생성할 수 있다. child process 가 정상적으로 생성된 경우 해당 pid 가 return 돼고, 아닌 경우 -1이 return 된다.

**(iv) wait**

- 이 함수의 구현을 통해 parent process가 특정 child process가 종료될 때까지 기다리고, 종료된 child process의 exit status를 파악할 수 있다. 만약 invalid 한 pid 이거나 동일한 pid에 대해 두 번 호출되는 등의 경우 -1을 return 한다.

**(v) write, read**

**-** 이번 프로젝트에서 write, read system call을 완전히 구현하지는 않는다. 단지, User Program이 STDOUT으로 출력하고, STDIN으로 입력을 받을 수 있도록 한다.

* 1. **개발 내용**
* **아래 항목의 내용만 서술 (기타 내용은 서술하지 않아도 됨.)**
* **Argument Passing**
  + 커널 내 스택에 argument를 쌓는 과정 설명
* 실행 파일이 성공적으로 메모리에 load된 경우, argument\_passing 함수를 호출해 stack에 argument를 쌓게 된다. 이 함수에 전달되는 parameter는 \*\*esp, argv[], argc이고, 각각 stack의 top / argument 배열 / argument 개수이다. 만약 User가 ‘echo x’를 입력한 경우, 각 값은 다음과 같다.

\*esp = PHYS\_BASE, argv[0] = “echo\0”, argv[1] = “x\0”, argc = 2

아래는 80x86 Calling Convention에 따른 argument passing 과정이다.

**(i) Argument 내용 쌓기**

**-** argv 배열 원소의 각 내용을 stack에 복사해 준다. 이 때 argv의 가장 마지막 원소부터 stack에 삽입해야 한다. 삽입할 때, 각 argv 원소의 길이만큼 esp을 감소시킨 뒤 해당 주소부터 길이 byte만큼 원소 내용을 복사해준다. 복사할 땐 strlcpy 함수를 이용하였다. 코드는 다음과 같다.

텍스트, 스크린샷, 폰트, 디자인이(가) 표시된 사진

자동 생성된 설명

여기서 len\_sum 값은 추후 word\_alignment를 위해 사용되고, address\_array 배열은 각 argument 원소의 주소를 stack에 쌓을 때 사용된다.

**(ii) Word-alignment**

**-** Calling Convention에 의해, esp 값을 4의 배수로 맞춰주어야 한다. 이는 위에서언급한 len\_sum 값을 통해 할 수 있다. len\_sum은 (i) 과정에서 esp가 감소한 값을 저장하고 있다.

esp의 초기 값은 4의 배수이므로(PHYS\_BASE), len\_sum 이 4의 배수라면 esp값의 조정은 필요없다. 하지만 4의 배수가 아니라면, esp에서 (4 – len\_sum%4) 값을 추가적으로 빼주어 4의 배수로 만들어줘야 한다. 코드는 다음과 같다.

스크린샷, 텍스트, 폰트이(가) 표시된 사진

자동 생성된 설명

**(iii) Null Value**

**-** Word-align 이 끝나면 추가적으로 4byte 만큼 0값을 넣어줘야 한다. 이는 단순히 esp 값을 4만큼 감소시켜주면 된다.

**(iv) Argument 주소 삽입**

**-** (i) 과정에서 각 argument의 내용을 stack에 push 하였다. 이 단계에선, push된 argument의 stack에서의 주소 값을 넣어줘야 한다. 이 주소값들은 위에서 언급한 address\_array배열에 들어있다. (address\_array[i] = argv[i] 의 주소) 이다. esp를 4씩 감소시키면서, argv의 마지막 원소의 주소부터 stack에 쌓아준다. 코드는 다음과 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

argv\_addr은 argv 배열의 시작주소를 stack에 쌓을 때 사용된다.

**(v) argv, argc, fake return address 삽입.**

- argv의 값은 argv\_addr에 저장돼 있고, argc 값은 parameter로 넘어온 argc 변수에 저장돼 있다. fake return address는 0이다. 따라서 각각 esp를 4씩 감소한 뒤, stack에 넣어주면 된다. 코드는 다음과 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

argument passing 과정에서 0으로 초기화해주는 부분을 단순히 esp 값의 감소만으로 해결한 것은, setup\_stack 함수에서 page를 할당 받을 때 PAL\_ZERO flag를 주었기 때문이다. 이 flag를 palloc\_get\_page 함수에 넘겨준 경우 해당 page가 모두 0으로 초기화된 상태로 return 되기 때문에 별도로 0으로 채워 넣을 필요가 없다. return address의 경우 명시적으로 표현하기 위해 코드를 작성했다.

* **User Memory Access**
  + Pintos 상에서의 invalid memory access 개념을 간략히 설명
  + Invalid memory access를 어떻게 막을 것인지 설명
* Pintos에서 Invalid memory access란, user mode에서 실행되는 중 kernel virtual address에 접근하거나, mapping 돼 있지 않은 virtual address에 접근하는 것을 말한다. Invalid memory access를 OS가 처리하지 않는다면 다른 Process의 Address Space 또는 Kernel 영역에 접근할 수 있어 반드시 막아야 한다.

이를 방지하기 위해선 user가 발생시킨 virtual address를 is\_kernel\_vaddr, pagedir\_get\_page == NULL 등의 조건문을 통해 검사하고, 만약 조건문의 결과가 true 인 경우 exit(-1)을 호출해 해당 process를 종료시켜야 한다.

* **System Calls**
  + 시스템 콜의 필요성에 대한 간략한 설명
  + 이번 프로젝트에서 개발할 시스템 콜에 대한 간략한 설명 (하나의 시스템 콜 당 최대 3문장으로 간략히 설명; 3문장을 넘길 정도로 길게 작성하지 말 것)
  + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명
* 각 process들의 address space 및 kernel space를 보호하기 위해 OS는 user mode와 kernel mode를 제공한다. user mode에선 memory 나 disk 에 접근하거나 process를 생성하는 등의 privileged operation은 수행하지 못한다. User mode에서 실행되는 user program이 이러한 operation을 수행하고 싶은 경우, 직접 수행하는 것이 아니라 OS 에게 부탁해야 한다. 이를 위해 OS가 제공하는 것이 System Call이다. System Call을 통해 User program은 privileged operation을 수행할 수 있고, OS는 user program의 요구를 들어주면서 memory의 protection도 가능해진다.

이번 프로젝트에선 총 6개의 System Call을 구현했다.

**(i) halt**

- pintos 내장 함수인 shutdown\_power\_off 함수를 호출해 Pintos를 종료한다.

**(ii) exit**

- exit을 호출한 process를 종료시키고, parameter로 넘어온 exit status를 kernel 에게 return 한다. 또한, 종료된 process의 이름과 exit status를 kernel console에 띄워준다. 다음은 termination message의 예시이다.

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

**(iii) exec**

**-** parameter로 넘어온 실행 파일에 해당하는 process를 생성한다. 이 System call 을 호출한 process와 생성된 process는 parent-child 관계를 갖게 되며, parent의 child process list에 새롭게 생성된 process가 들어가게 된다.

**(iv) wait**

**-** parent process가 특정 child process가 종료될 때까지 block 돼고, 종료된 process의 return state를 얻어온다. 이를 위해 parent process와 child process간의 synchronizatoin 이 필요하다.

**(v) write**

**-** User program이 출력을 STDOUT으로 내보낼 수 있도록 해준다.

**(vi) read**

**-** User program이 STDIN으로 입력 받을 수 있도록 해준다.

* User가 System Call 을 호출하면, 해당 System call 함수 내부적으로 syscallN 함수를 호출한다. 이 때 N은 argument의 개수이다. 이 함수에서 argument, system call number를 stack 에 push 한 뒤, int$0x30 명령어를 수행하면 user mode 에서 kernel mode로 변경된다. int $0x30에서 user의 context를 kernel stack에 push하고, interrupt handler를 호출한 뒤 interrupt handler에서 interrupt frame에 있는 interrupt number를 보고 system call handler를 호출한다. system call handler에선 intr frame을 통해 system call number와 user stack을 접근하고, 그에 맞는 작업을 수행한다. 그 후 return from trap instruction을 통해 다시 user mode로 돌아온다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

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

**(I) Argument Passing, User Memory Access : 9/19 ~ 9/22**

**(II) System Call Handler : 9/23~9/24**

**(III) System Call Implementation : 9/25~9/28**

**(IV) Additional Implementation : 9/28-10/30**

**(V) Report : 9/30-10/01**

* 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, User Memory Access**

**\* 수정한 함수**

**(i) userprog/process.c/process\_execute (for argument passing)**

- thread의 이름은 thread\_create 함수의 첫 번째 인자로 넘겨준 char string으로 초기화 된다. 추후 exit 에서 termination message를 출력할 때 thread\_name() 함수를 호출하고, 여기서 return 되는 thread의 이름이 thread\_create로 넘겨주는 string이다. 따라서 input으로 들어온 file\_name에서 실행파일의 이름만을 추출해서 저장하기 위해 char execution\_name[256] 변수를 선언하고 저장하였다. 이 변수에 실행 파일 이름을 저장한 뒤, thread\_create에 이 string을 넘겨준다.

**(ii) userprog/process.c/load (for argument passing)**

- Argument Passing을 위해선, command line을 parsing하여 실행 파일과 Argument 들로 쪼개야 한다. 이를 위해, 두 개의 변수를 선언하고 parsing을 위한 함수를 추가하였다. 추가된 변수 및 함수는 다음과 같다.

***- char\* argv[MAXARGV]***

***- char\* commandline***

***- int argc***

***- int parse\_commandLine(char\*commandLine, char\*argv[])***

***- void argument\_passing(void\*\* esp, char\*argv[], int argc)***

commandline 변수는 load 함수에 넘어온 file\_name을 복사해 놓기 위한 변수이다. argv 변수는 쪼개진 실행 파일의 이름과 Argument들을 저장한다. 위의 변수를 parse\_commandLine 함수에 인자로 넘겨주면 strtok\_r 함수를 이용해 commandline string을 “ “ 을 기준으로 parsing 하고, argv 에 하나씩 넣어준다. return value 는 argv 원소의 개수이고, 이 값은 argc 변수에 저장된다.

argument\_passing 함수는 user program의 stack에 parsing 된 argument들을 push 해주는 함수이다. 이 함수에서 위에서 설명한 Argument Passing 과정을 거치면서 Argument 들이 stack에 삽입된다.

**(iii) userprog/syscall.c/syscall\_handler, syscall\_init (for user memory access)**

추가된 변수 및 함수는 다음과 같다.

***- int argNums[SYS\_CALL\_NUM];***

***- bool checkUserMemoryAccess(uint32\_t\* offset)***

***- void argNumInit()***

**-** user program이 system call을 호출하면, stack에 system call을 위한 argument들push 하고 이 함수가 호출된다. 이 때 user program이 Invalid Pointer를 넘겨줄 수 있으므로, stack에 있는 모든 parameter 들에 대해 체크해주어야 한다.

system call 마다 parameter의 개수가 다르므로, 체크해야 하는 Parameter의 개수도 다르다. 이를 하나의 반복문으로 처리하기 위해 argNums 배열을 추가하고 각 system call의 parameter 개수를 저장해놓았다. argNums의 초기화는 argNumInit() 함수에서 일어나고, argNumInit() 함수는 syscall\_init 함수에서 호출된다.

위의 변수들과 함수들을 이용해 syscall\_handler 함수에서 parameter에 대한 validation을 수행한다. 확인 부분은 다음과 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

checkUserMemoryAccess 함수는 is\_user\_vaddr, pagedir\_get\_page 등의 함수를 이용해 Invalid pointer를 확인한다.

**(iv) userprog/exception.c/page\_fault (for memory access)**

User process가 잘못된 접근을 했을 경우, page fault가 발생하고 해당 process를 종료시켜야 한다. 이를 위해 page\_fault 함수에서 잘못된 접근 발생 시 exit(-1)을 호출해 user process를 종료시키는 코드를 추가하였다. 코드는 다음과 같다.



**(2) System Call Handler**

system call 을 처리하기 위해, userprog/syscall.c 파일에 구현하려는 system call 에 해당하는 함수들을 선언하였다. 또한, system call 번호에 따른 분기를 위해 syscall\_handler 함수에 switch 문을 작성하였다. 추가된 함수는 다음과 같다.

***- void halt(void)***

***- void exit(int status)***

***- pid\_t exec(struct intr\_frame\* f)***

***- int wait(struct intr\_frame\* f)***

***- int read(struct intr\_frame\* f)***

***- int write(struct intr\_frame\* f)***

exit을 제외한 system call들은 intr\_frame\* 를 인자로 받고, 해당 함수 내에서 필요한 paramter를 접근한다. exit의 경우 exit(-1) 과 같은 형식으로 빈번하게 호출되므로 예외적으로 int를 넘겨준다. 또한, exit의 경우 page\_fault() 함수 등에서도 호출돼야 하므로 다른 함수들과 다르게 userprog/syscall.h 에 prototype을 선언해두었다.

**(3) System Call Implementation**

**(i) halt**

- halt system call의 구현을 위해, userprog/syscall.c 에 halt 함수를 추가하고, 내부적으로 sutdown\_power\_off() 함수를 호출하도록 하였다.

**(ii) exit**  
- exit system call의 구현을 위해, userprog/syscall.c 에 exit 함수를 추가하고, termination message를 띄우도록 하였다. 또한, exit 같은 경우 호출한 process의 exit status를 kernel 이 저장해야 한다. 이를 위해 threads/thread.h 의 struct thread의 멤버 변수로 ‘int exit\_status’ 를 추가하였다. 호출한 process의 exit status는 이 변수에 저장된다.

**(iii) exec**

exec 함수는 내부적으로 process\_execute 함수를 호출하고, 이 함수의 return 값을 return 한다. 이 때 exec 함수에 전달된 인자에 따라 child process가 성공적으로 생성될 수도 있고, 실패할 수도 있다. 만약 실패할 경우 -1을 return 해 주어야 한다. 이를 위해 process.c/process\_execute 및 process.c/start\_process 함수에 올바른 실행 파일을 넘겨주었는지 synchronization 을 통해 확인하였다. synchronization 관련 변수 및 초기화는 thread.c, thread.h 파일에 추가하였다.

추가된 변수는 다음과 같다.(thread.h)

**- struct thread\*parent**

**- bool child\_succeed\_flag**

**- struct semaphore exec\_lock**

parent는 자신을 생성한 parent process를 가리키고 있다. child\_succeed\_flag는 exec 명령어 수행 후 자신이 생성한 child process가 정상적으로 load된 경우 true, 아닐 경우 false 이다. exec\_lock은 child process의 성공 여부가 확정난 이후 parent process를 실행하기 위한 semaphore이다.

**(iv) wait**

* wait system call은 호출한 process가 특정 child process 가 종료되는 것을 기다리고 exit status를 return 해준다. 이를 위해 각 process의 child process들을 관리해야 한다. 또한, child process가 종료될 때까지 return 되면 안 되므로 synchronization 도 필요하다. 이를 위해 threads/thread.h 의 struct thread에 다음 멤버를 추가하였다.

**- struct list child\_process\_list**

**- struct list\_elem child\_list\_elem**

**- struct semaphore sema\_for\_wait**

**- struct semaphore sema\_for\_locking\_memory**

child\_process\_list, child\_list\_elem 는 parent process에서 child process의 struct thread 접근하기 위한 list 변수들이다. sema\_for\_wait, sema\_for\_locking\_memory 는 synchronization 을 위한 변수들이다.

wait를 위해 수정된 함수는 다음과 같다.

**- threads/thread.c/init\_thread**

**- userprog/process.c/process\_wait**

**- userprog/process.c/process\_exit**

init\_thread엔 추가된 struct thread 변수들의 초기화 코드가 추가됐다. process\_wait는 wait system call이 내부적으로 호출하는 함수로, semaphore를 통해 synchronization을 한 후 child 의 exit status를 return 해주도록 구현됐다. process\_exit은 parent process의 lock을 풀어주고, parent process가 child process의 자원을 reaping 할 때까지 block 된 뒤 return 된다.

**(v) write, read**

- write, read system call 구현을 위해 userprog/syscall.c 에 각 함수를 추가하고 내부적으로 putbuf, input\_getc 함수를 이용해 구현하였다.

1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

1. **Argument Passing**

텍스트, 도표, 스크린샷, 영수증이(가) 표시된 사진

자동 생성된 설명

1. **User Memory Access**

텍스트, 도표, 라인, 폰트이(가) 표시된 사진

자동 생성된 설명

1. 텍스트, 스크린샷, 영수증, 도표이(가) 표시된 사진

   자동 생성된 설명**System Calls**

* 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**

Argument passing 을 위해 우선적으로 수정한 함수는 process\_execute 함수이다. struct thread에 thread의 이름이 제대로 들어갈 수 있도록, 인자로 들어온 file\_name 에서 실행 파일의 이름만을 추출하여 thread\_create 함수에 인자로 넘겼다. 코드는 다음과 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

첫 번째 while 문은 file\_name의 앞 부분에 공백이 있을 경우, 이를 제거하기 위한 것이고, 두 번째 while 문은 실행 파일의 이름이 끝나는 부분을 찾기 위한 것이다.

start\_process 함수에서 load 함수를 호출하고, load 함수에서 실질적인 parsing 과 passing 이 일어난다. thread\_safe 한 parsing을 위해, 전역 변수를 사용하지 않고 argument를 저장할 char\*argv[]와 parsing에 사용될 char \*commandline를 지역변수로 선언하고, parsing 함수에 넘겨주는 형태로 구현하였다. 변수 선언 부분은 다음과 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

file\_name을 변형하지 않기 위해, commandline 변수에 copy 한 뒤 parse\_commandLine 함수엔 이 변수를 넘겨주었다. parse\_commandLine은 strtok\_r 함수를 이용해 commnadline 변수를 각각의 argument 로 parsing하고, 이를 argv 변수에 순차적으로 넣어준다. 또한, return 값은 argument의 개수이고 이는 argc에 저장된다. parse\_commandLine 함수의 코드는 다음과 같다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

parsing 이 끝나면, argv[0] 에는 실행 파일의 이름이, argv[1] 부턴 argument가 들어있을 것이다. 또한, argc에는 argument의 개수가 들어있을 것이다. 실행 파일의 이름인 argv[0] 을 이용해, filesys\_open 함수를 호출한다. 만약 open 이 성공한다면, ELF 파일을 읽어 필요한 부분을 메모리에 적재한다. 그 후, stack을 setup 하고 argument passing 함수를 호출해 stack에 argument 를 쌓아준다. argument\_passing 함수는 인자로 esp 와 argv, argc를 받고, 80x86 Calling Convention대로 구현됐다. 호출 부분과 argument\_passing 함수는 다음과 같다.

* **argument\_passing 호출**

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

* **argument\_passing 함수 구현**

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

순차적으로 argv의 내용, word-align, Null Pointer, argv 원소 주소, argv 주소, argc, return address를 쌓아주는 것을 확인할 수 있다.

1. **User Memory Access**

* User가 Invalid 한 Memory를 Access 할 경우, 해당 User process를 종료 시켜야 한다. 따라서 System Call 에 전달한 Pointer 값을 확인하고 잘못된 영역에 접근해 page\_fault가 발생한 경우 이를 처리하는 코드를 작성하였다.

우선 System Call의 경우, 위에서 설명한 것처럼 System Call의 종류에 따라서 확인해야 하는 인자의 수가 다르다. 이를 하나의 반복문으로 처리하기 위해 argNums 배열을 선언하고, argNumInit() 이라는 함수를 만들어 syscall\_init 함수에서 호출하도록 하였다. 선언 부분 및 초기화 함수 부분은 다음과 같다.

* **변수 선언**

텍스트, 폰트, 스크린샷, 디자인이(가) 표시된 사진

자동 생성된 설명

* **함수 호출 및 함수 구현**

스크린샷, 텍스트이(가) 표시된 사진

자동 생성된 설명

스크린샷, 텍스트, 블랙이(가) 표시된 사진

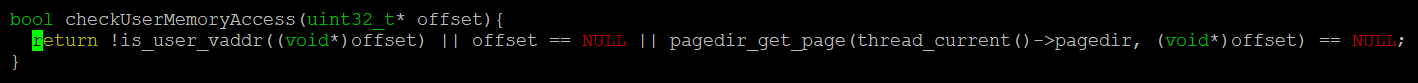
자동 생성된 설명

이 argNums 배열을 이용해, sycall\_handler 함수에서 User가 전달한 Pointer에 대한 Validation 을 반복문을 통해 진행한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

각 System Call의 argument의 수만큼 checkUserMemoryAccess 함수를 호출하면서, 만약 true 가 return 된 경우 Invalid Pointer 이므로 바로 exit(-1) 함수를 호출해 해당 process 를 종료한다. checkUserMemoryAccess 함수는 다음과 같다.



user virtual address가 아닌 곳에 접근하거나 NULL pointer 인 경우, 혹은 mapping 돼 있지 않은 virtual address를 접근하는 경우 true를 return 한다.

System Call 외에도, user가 Invalid address에 접근하려는 경우 page fault가 발생하고 page\_fault() 함수가 호출된다. 이 함수에서도 역시 process를 종료시켜야 하므로, exception.c/page\_fault() 함수도 수정해 주었다. 수정된 부분은 다음과 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

page fault가 발생했을 때 만약 user가 아니거나, kernel virtual address에 접근한 경우 exit(-1) 함수를 호출해 user process를 종료 시킨다.

1. **System Calls**

* **이번 프로젝트에서 개발한 시스템 콜을 구현 관점에서 상세히 서술.**
* User 가 System Call을 호출하는 경우, 최종적으로 syscall\_handler가 호출된다. 이 때 system call number에 따라서 적절한 함수를 호출해주어야 한다. 이를 위해 syscall\_handler에 switch 문을 추가해, system call number에 해당하는 handler를 호출해주었다.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

syscall\_number 변수에 system call 번호를 저장하고, 그에 따라 system call 을 호출하는 것을 확인할 수 있다. 이 때 return value가 있는 system call의 경우 eax register에 return value를 저장해주어야 하는데, 그 부분도 구현돼 있는 것을 확인할 수 있다. 이제 각 시스템 콜의 구현을 하나씩 설명하겠다.

**(I) halt**

**-** halt system call은 pintos를 종료시키는 system call이다. pintos 내장 함수인 shutdown\_power\_off() 함수를 호출하고 종료된다. 코드는 다음과 같다.

텍스트, 폰트, 스크린샷, 그래픽이(가) 표시된 사진

자동 생성된 설명

**(II) exit**

**-** exit system call은 호출한 thread의 name 과 exit status를 출력하고, kernel에 해당 process의 exit status를 반환한 뒤 process를 종료한다. argument passing 과정에서 설명한 것처럼, thread\_create 함수를 호출할 때 실행 파일의 이름을 parsing 해서 넘겨주었으므로 thread\_name() 함수를 호출하면 exit system call을 호출한 thread의 이름이 return 된다. 또한, exit status를 kernel 자료구조인 struct thread의 member에 저장함으로써 kernel에 exit status를 반환하였다.

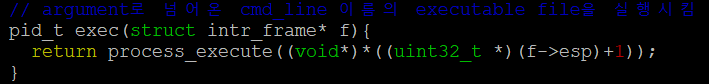
마지막으로 호출한 process를 종료하기 위해 thread\_exit() 함수를 호출한다. 코드는 다음과 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

**(III) exec**

exec system call은 user program이 child process를 생성할 수 있도록 한다. process의 생성은 process\_execute 함수에서 시작되므로, exec 내부에선 단순히 process\_execute 함수를 호출하고 그대로 return 된다. exec 코드는 다음과 같다.



처음에는 단순히 이렇게 구현하였는데, 테스트를 해보니 문제가 발생하였다. 올바른 입력이 들어왔을 때는 문제가 없었지만, 잘못된 실행파일 이름을 exec 에게 전달하면 -1을 return 하지 않고 양수 tid 값을 return 하는 것이었다.

exec 함수는 인자로 전달된 executable file을 load 하지 못하는 등의 이유로 process가 정상적으로 생성되지 않은 경우 -1을 return 해야 한다. 하지만 이렇게 구현할 경우 child process가 정상적으로 load 되지 못해도 -1을 return 하지 못하였다.

process의 생성은 thread\_create->start\_process->load->jump to main 순으로 진행되는데, thread\_create 가 호출되는 순간 child process는 별도의 thread로서 scheduling 되며 concurrent 하게 실행된다. 또한, thread\_create가 return 해주는 tid 값은 child process가 정상적으로 load되는지 모른채 allocate\_tid 함수가 return 해주는 tid 값을 process\_execute 에게 전달한다. process\_execute 역시, child process의 정상 load 여부를 모르고 exec 함수에게 tid 값을 return 한다. 즉, 이렇게만 구현했을 경우 process\_execute는 항상 child process가 올바르게 실행됐다는 가정이 성립할 때의 tid 값만 return 해준다.

이를 해결하기 위해 exec를 호출한 parent process와 생성된 child process와의 synchronization을 해주었다. process\_execute를 thread\_create함수가 끝나고 바로 return 하는 것이 아니라, child process의 load 성공 여부를 보고 그에 따라 tid 값을 return 하도록 하는 것이다. 이를 위해 struct thread에 다음과 같은 세 가지 변수를 추가했다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

struct thread\* parent는 자신의 parent process의 struct thread 를 가리킨다. child\_succeed\_flag 는 parent process가 생성한 child process 가 성공적으로 load 됐을 경우 true이고, 아닐 경우 false이다. process\_execute 함수에선 이 flag 값을 보고 return 값을 정한다. struct semaphore exec\_lock은 parent와 child의 synchronization을 위한 semaphore이다. 0으로 초기화 되며, parent process 에선 process\_execute 에서 return 하기 전 이 semaphore에 sema\_down operation 수행해 block 되고, child process에서 정상 실행 여부가 결정된 경우 flag 값을 수정하고 struct thread\* parent를 통해 해당 semaphore에 대해 sema\_up 연산을 수행해 parent process를 release 해준다. 코드는 다음과 같다.

**(i) semaphore 및 parent 초기화 부분(threads/thread.c/init\_thread)**

텍스트, 폰트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

**(ii) process\_execute의 sema\_down 및 flag 값에 따른 return 부분**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명 **(iii) start\_process 에서 child process가 semaphore를 release 해주는 부분**

**// load 실패**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

**// load 성공**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

위와 같은 synchronization 과정을 통해 exec 명령어가 실패할 경우 exec를 호출한 process 에게 정상적으로 -1을 return 할 수 있었다.

**(iv) wait**

wait system call은 parent process가 특정 child process가 종료될 때까지 기다리고 child process의 exit status 를 return 해준다. 내부적으로 process\_wait(pid) 를 호출하고 return 한다. 코드는 다음과 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

wait를 위해선, parent process가 자신의 child process들의 struct thread list 를 가지고 있어야 한다. 그래야 exit status를 가져오고 해당 tid 값을 갖는 child process의 reaping이 가능하기 때문이다. 여기서 reaping은 parent process의 child process list에서 해당 chlid process struct thread 를 제거하고 exit status를 가져오는 것을 말한다.

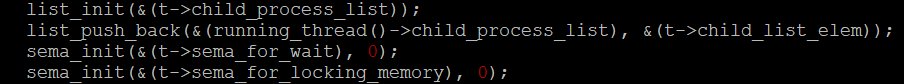
또한, parent 는 child 가 끝날 때까지 기다려야 하므로 semaphore를 통한 synchronization 이 필요하다. 이와 더불어, 어떤 이유로든 child process가 종료되면 thread\_exit 함수에서 process\_exit함수를 호출하고 child process의 struct thread 자료구조를 free 해준다. 이러한 자료구조의 free는 parent process의 reaping이 끝난 뒤 수행돼야 한다. parent process가 child process의 struct thread 구조에 접근하기 때문이다. 따라서 이를 위한 synchronization 도 필요하다.

위 내용의 구현을 위해 struct thread에 추가한 member는 다음과 같다.

스크린샷, 텍스트, 폰트이(가) 표시된 사진

자동 생성된 설명

4개의 변수 중 위의 2개는 chlid process thread 관리 및 접근을 위한 것이고, sema\_for\_wait는 parent와 child 간의 synchronization, sema\_for\_locking\_memory는 child struct thread free의 synchronization 을 위한 semaphore 이다. 이 변수들은 threads/thread.c/init\_thread에서 초기화 되고, 코드는 다음과 같다.



두 semaphore 모두 특정 코드 flow에 대한 scheduling을 위한 것이므로 0으로 초기화 된다. list\_push\_back 부분을 보면, running\_thread의 child process list에 생성된 child process를 추가해준다. 이 때의 running thread는 process\_execute 함수를 호출한 thread 일 것이다. 결과적으로 어떤 user program이 exec system call 을 호출하면 process\_execute 함수가 호출되고, 호출한 process의 child process list에 생성된 process 가 들어가게 된다.

process\_wait 함수의 코드는 다음과 같다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

thread\_current() 함수를 통해 process\_wait 함수를 호출한 thread의 struct thread를 가져오고, 이를 통해 child process list 에 접근한다. 모든 child thread들을 살펴보면서 기다리고자 하는 tid 와 같은 tid 값을 갖는 child process를 찾은 경우, sema\_down을 수행해 해당 process가 종료될 때까지 기다린다. sema\_down에서 release 되는 순간은 child process가 process\_exit에서 sema\_up 연산을 수행해 줄 때이다. 그 부분은 다음과 같다.

**// userprog/process.c/process\_exit**



parent process가 release 되면 종료된 child process의 exit status를 받고, child process list에서 종료된 child process를 제거해준다. 이 때 위에서 언급한 것처럼 parent process의 reaping이 끝나기 전까지 child process의 자료구조는 free 되면 안된다. 따라서 process\_exit 에서 child process 가 return 되기 전, memory lock semaphore 에 sema\_down 연산을 수행해 parent process 가 reaping 하기 전까지 block 상태로 들어간다. 그 부분의 코드는 다음과 같다.



child process의 reaping이 끝나면, parent process는 해당 semaphore에 대한 sema\_up 연산을 수행하고 child process의 자료구조는 free 된다.

만약 parent process의 child process에 찾고자 하는 tid 를 가진 child process가 있다면, 해당 exit status를 return 하면 된다. child process list에 해당 child process가 없다면 -1을 return 한다. 똑같은 tid 에 대해 wait 를 두 번 수행하는 경우 child process list에 해당 child process가 없을 것이므로 -1이 return 될 것이다. kernel에 의해 종료돼 exit status에 -1이 들어간 경우 역시, parent process가 해당 exit status를 return 할 것이므로 정상적으로 -1이 return 된다.

**(v) write**

- 이번 프로젝트에서 write system call의 완전한 구현은 하지 않는다. 단지 user program이 STDOUT 으로 화면에 내용을 출력할 수 있도록 해준다. 코드는 다음과 같다.

텍스트, 스크린샷, 디스플레이, 폰트이(가) 표시된 사진

자동 생성된 설명

parameter로 넘어온 intr\_frame 을 통해, file descriptor, buffer, size를 가지고 온다. buffer엔 fd 가 가리키는 file struct에 쓰고자 하는 내용이, size는 몇 byte 만큼 쓸 것인지 나타낸다. 만약 fd 값이 STDOUT\_FILENO 이라면 pintos 내장 함수인 putbuf 함수를 이용해 화면에 내용을 출력해주고 출력한 byte 수를 return 한다. fd 가 STDOUT\_FILENO 이 아니면 -1 을 return 해준다.

**(vi) read**

* write 와 마찬가지로, 이번 프로젝트에서 read에 대한 완전한 구현은 하지 않는다. input\_getc 함수를 통해 STDIN 으로 입력 받는 형태로만 구현해 놓았다. 코드는 다음과 같다.

텍스트, 스크린샷, 디스플레이, 소프트웨어이(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**
* System call 을 구현하기 위해선, 해당 System Call을 Handler 에 mapping 시키기 위한 System call number 와 User에게 해당 System Call을 제공하기 위한 System Call API 가 필요하다. 또한, max\_of\_four\_int 의 경우 4개의 parameter 를 갖는 새로운 System Call 이기 때문에 syscall4를 새롭게 구현해줘야 한다. 우선 Syscall number는 src/lib/syscall-nr.h 에 새롭게 추가했다. 추가한 부분은 다음과 같다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

User 를 위한 System Call API 및 syscall4에 대한 구현은 src/lib/user/syscall.h 및 src/lib/user/syscall.c 부분에 추가하였다. 해당 부분 코드는 다음과 같다.

**// new system call API prototype(lib/user/syscall.h)**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

**// new system call API implementation(lib/user/syscall.c)**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

**// syscall4 Implementation(lib/user/syscall.c)**

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

User가 해당 API 를 사용해 System Call 을 호출하면, System Call 호출 과정을 따라 syscall\_handler가 호출된다. 따라서 userprog/syscall.c 파일의 수정이 필요하다. 이곳에 새롭게 만든 system call에 대한 handler를 추가하고, switch 문에 새로운 case문을 추가해줬다. 코드는 다음과 같다.

**(i) case문 추가(userprog/syscall.c/syscall\_handler())**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

**(ii) fibonacci, max\_of\_four\_int 함수 추가(userprog/syscall.c)**

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

두 함수 모두 interrupt frame 을 parameter로 받아서 필요한 인자들을 저장한다. fibonacci 함수의 경우 메모리를 효율적으로 사용하기 위해서 int 변수 3개를 통해 n 번째 피보나치 수열 값을 구했다. max\_of\_four\_int 함수의 경우 분할 정복을 통해 조금이라도 효율적으로 구현하려고 노력하였다.

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

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명