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

담당 교수 :박운상

이름: 김은원

개발 기간 :2022/9/20 ~ 2022/10/2

1. **개발 목표**

유저 프로그램이 종료되었을 때, kernel 에서 종료 메시지를 출력하는 것을 구현한다. 이를 위해서는 argument passing 구현, system call 구현, 유효하지 않은 포인터의 접근을 막는 구현을 해야 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

입력 받은 Arguments를 띄어쓰기 기준으로 나누어 스택에 저장되어야 한다. 오른쪽 인자부터 왼쪽으로 스택에 들어간다. 모든 단어가 들어가면 4의 배수로 맞춰지고 인자의 주소가 스택에 저장된다.

1. User Memory Access

NULL pointer 또는 매핑되지 않은 가상 메모리에 접근하는 pointer 또는 kernel 영역에 접근하는 pointer 들을 막는다.

1. System Calls

유저레벨에서 system call를 호출하면 kernel 에서 이를 수행하고 값을 다시 user에게return할 수 있다.

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

파일 원형을 변형시키지 않게 하기 위해서 우선 파일을 복사한다. 문자열을 delimiter를 기준으로 나누어 주는 strtok\_r 함수를 이용하여 arguments를 나눈다. 나눠진 arguments들을 오른쪽부터 하나씩 스택에 넣는다. 다 넣었다면 4의 배수로 맞춰주기 위해서 word-align을 이용해 맞춘다. 또한 주소와 구분하기 위해서 0을 넣어주는 작업도 필요하다. 그 다음 스택에 argument의 주소를 담아주고 그 다음 argument의 개수, 마지막으로 다시 0을 넣어준다. 스택에 넣을 때는 stack pointer인 \*esp를 이용한다.

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

널 포인터이거나 매핑되지 않은 가상공간에 접근하거나 커널에 침범하는 유효하지 않은 메모리 접근은 매우 프로그램에 치명적이다. 따라서 이때는 프로세스를 종료하거나 해제해야 한다.

* + Invalid memory access를 어떻게 막을 것인지 설명

Syscall.c에 위 경우의 유효하지 않은 메모리 접근을 막기위해 주소를 확인하는 함수를 만든다. 그리고 system call을 할 때마다 확인한다. 포인터가 NULL인지 또는 유저영역이 맞는지 is\_user\_vaddr() 함수를 이용해 확인한다. 물리 주소가 유저의 가상 주소에 일치하는지 즉 mapping 된 가상 메모리인지 확인하는 pagedir\_get\_page() 함수를 이용해 확인한다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명

유저영역에서 모든 기능을 구현하기 어렵다. 따라서 유저에서 프로세스를 실행시키면 해당 필요한 작업들을 커널에 부탁하는 것이다. 커널에서는 각 작업들을 수행하고 다시 유저에 되돌려준다. 이러한 구조는 유저가 커널에 쉽게 접근하기 어렵기 때문에 시스템을 보호한다.

* + 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: pintos를 종료한다.
2. Exit: 현재 유저 프로그램을 종료한다.
3. Exec: 자식 프로세스를 만든다.
4. Wait: 자식 프로세스를 만들고 이를 종료할 때까지 기다린다. 자식 프로세스가 죽으면 자식 스레드로부터 종료 상태를 받는다. Wait() 에서 반환값을 받을 때가지 프로세스의 종료를 멈춘다.
5. Read: 읽는 함수이다. 입력받은 size 만큼 읽고 버퍼에 저장한다. ‘\0’이 들어올 때 인덱스 값을 반환한다.
6. Write: 출력하는 함수이다. 버퍼에 있는 것을 파일에 출력한다.
   * 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명

먼저 유저 레벨에서 함수를 부르면 인자 개수에 따라 syscall1~4 중 하나가 선택된다. 인자의 개수가 2개이면 syscall2가 호출된다. System call handler에서 해당 함수의 System call number에 따라 system call를 부르게 된다. 불러진 함수에서 해당 작업을 수행하고 return 값을 eax에 저장한다. 유저레벨에서 저장한 eax를 통해 반환 받는다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

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

9/20~9/24: argument parsing

9/24~9/26: argument passing

9/26~9/30: system call 구현

9/30~10/1: additional 함수 구현

10/2: 파일 압축 및 보고서 작성

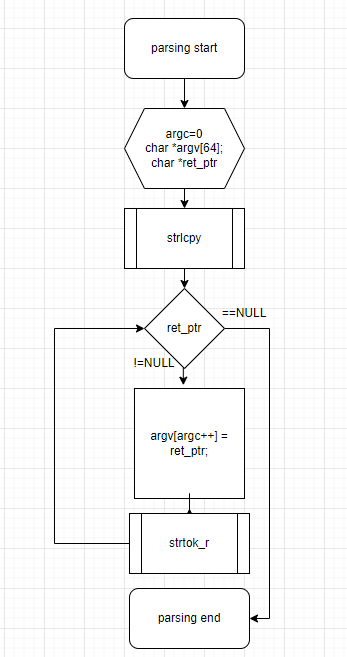
* 1. **개발 방법**
* **II. B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**

1. Argument passing : userprog/process.c 에서 load 함수를 수정한다. Load 함수에서 문자열을 나누고 스택에 저장하는 코드를 추가한다. Test에 통과하기 위해서 첫번째 인자만 출력하는 argument parsing이 필요하다. 따라서 process\_excute함수에서 thread\_create() 에 file\_name에 첫번째 인자만 들어가도록 수정한다.
2. Accessing user memory: userprog/syscall.c 에서 invalid memory access를 막는 함수 check\_user\_vaddr 함수를 만든다. 그리고 userprog/syscall.c 의 system\_handler() 에서 호출한다.
3. System call: userprog/syscall.c 에서 switch 문으로 syscall\_handler() 함수에 halt(), exit(), exec(), wait(), read(), write() 을 각각 나누어 실행하도록 한다. Syscall.c 에 halt(), exit(), exec(), wait(), read(), write()를 정의한다. Syscall.h에도 halt(), exit(), exec(), wait(), read(), write() 의 prototype을 작성한다. Syscall.c 맨 위 #include <syscall.h>를 추가하여 참조하도록 한다. Wait 함수에서는 자식 스레드 정보를 알알아 끝날 때까지 기다릴 수 있기 때문에 threads/thread.c 와 thread.h 파일도 수정해야한다.
4. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

1. Argument Passing

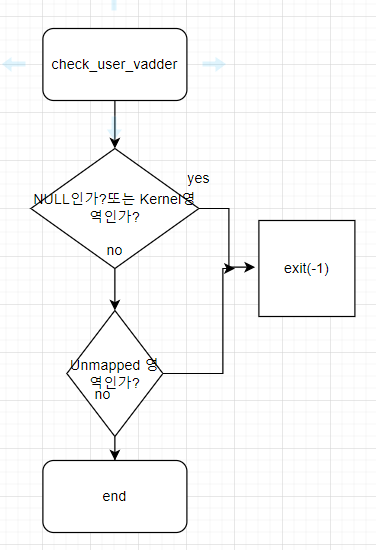
Argument parsing 부분 flow chart



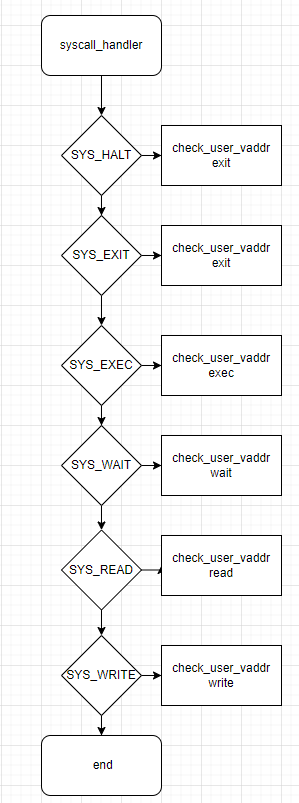
Parsing이 끝나고 stack에 저장하는 flow chart

|  |  |
| --- | --- |
| 1 | 2 |

1. User Memory Access



1. 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

우선 인자들을 받으면 parsing을 한다. 이때 “ “을 기준으로 단어를 만든다. 만약 ‘echo x’ 인 경우 echo, x 로 나눈다. 이 구현을 load 함수에서 filesys\_open() 전에 한다. 우선 인자들의 개수를 세기 위해서 int argc를 선언한다. 또한 인자들을 저장할 char \* argv[64] 를 선언했다. 또한 strok\_r함수를 사용할 때 필요한 포인터 두개 char \*ret\_ptr, \*next\_ptr를 선언했다. 파일 이름이 변형되지 않게 파일 이름을 복사할 char \*file\_cp[128]을 선언했다.

파일 이름이 변형되지 않게 strlcpy로 파일 이름을 file\_cp에 복사했다. 그리고 strok\_r 함수를 이용해 구분자 “ “을 기준으로 나누어 만약 ‘echo x’가 들어온 경우 ret\_ptr이 echo를 가리키도록 한다. 다시 open\_file이 ret\_ptr이 가리키는 것을 가리키도록 하고 2차원 배열 argv에 인자들을 넣는다. Ret\_ptr이 NULL이 아닐 때까지, 즉 구분자가 끝나고 ‘\0’을 가리킬 때까지 반복문을 실행한다. ‘echo x’를 예로 들면 argv[0]=”x\0” 이고 argv[1]=”echo\0”가 된다.

Parsing이 끝나면 스택에 argument을 넣는다. 인자들의 주소를 먼저 저장할 변수와 인자들을 저장할 스택 주소 크기를 가늠할 sum을 선언한다. 그리고 오른쪽부터 저장해야 하기 때문에 반대로 반복문을 실행한다.

n은 입력 받은 인자 문자의 수+1이다. +1을 한 이유는 문자열 맨 뒤 ‘\0’을 추가해야 하기 때문이다. \*esp는 스택포인터의 주소이기 때문에 n만큼 빼서 해당 주소를 가리키도록 한다. 그리고 스택에 인자를 저장한다. 그리고 주소를 arg\_address에 우선 저장한다.

4의 배수로 맞추어야 하기 때문에 word-align이 필요하다. Sum을 4로 나누어 0이면 이미 4의 배수로 맞추었기 때문에 4를 바로 빼면 된다. 하지만 0이 아니라면 나머지를 빼서 4로 맞춰준다. Word-align에는 0을 넣는다. 그리고 argv[argc]에는 0을 넣어서 주소를 넣는 기준점을 만들어 준다.

반복문을 통해 argv에 arg\_address를 저장한다. Argc와 0을 차례로 스택에 저장한다.

1. User Memory Access

Userprog/syscall.c에 check\_user\_addr 함수를 만들어 유저가 유효하지 않은 접근을 하면 종료시킨다. Is\_user\_vaddr 함수를 이용하여 vaddr가 PHYS\_BASE 이내에 있는지 확인했다. 또한 Unmapped 가상 메모리인지 확인하기 위해서 pagedir\_get\_page 함수를 이용하였다. pagedir\_get\_page 함수는 map이 되었는지 확인하는 함수이고 만약 아니라면 NULL을 반환하는 함수이다.

|  |
| --- |
| 18 void check\_user\_vaddr(const void \*vaddr){  19 if(vaddr == NULL ||!is\_user\_vaddr(vaddr)){  20 exit(-1);  21 }  22 void \*pointer= pagedir\_get\_page(thread\_current()->pagedir, vaddr);  23 if(pointer == NULL)  24 exit(-1);  25 } |

1. System Calls

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

syscall\_handler() 함수에 우선 시스템 넘버에 맞추어 switch문을 작성한다.

|  |
| --- |
| 26 static void  27 syscall\_handler (struct intr\_frame \*f)  28 {  29  30 switch(\*(uint32\_t\*)(f->esp)){  31 case SYS\_HALT:  32 halt();  33 break;  34  35 case SYS\_EXIT:  36 check\_user\_vaddr(f->esp+4);  37 exit(\*(uint32\_t\*)(f->esp+4));  38 break;  39 case SYS\_EXEC:  40 check\_user\_vaddr(f->esp+4);  41 f->eax = exec((char\*)\*(uint32\_t\*)(f->esp+4));  42 break;  43 case SYS\_WAIT:  44 check\_user\_vaddr(f->esp+4);  45 f->eax = wait(\*(uint32\_t\*)(f->esp+4));  46 break;  47 case SYS\_CREATE:  48 break;  49 case SYS\_REMOVE:  50 break;  51 case SYS\_OPEN:  52 break;  53 case SYS\_FILESIZE:  54 break;  55  56 case SYS\_READ:  57 check\_user\_vaddr(f->esp+4);  58 check\_user\_vaddr(f->esp+8);  59 check\_user\_vaddr(f->esp+12);  60  61 f->eax = read((int)\*(uint32\_t\*)(f->esp+4), (const void\*)\*(uint32\_t\*)( f->esp+8), (unsigned)\*(uint32\_t\*)(f->esp+12));  62 break;  63 case SYS\_WRITE:  64 check\_user\_vaddr(f->esp+4);  65 check\_user\_vaddr(f->esp+8);  66 check\_user\_vaddr(f->esp+12);  67 f->eax = write((int)\*(uint32\_t\*)(f->esp+4), (const void\*)\*(uint32\_t\*) (f->esp+8), (unsigned)\*(uint32\_t\*)(f->esp+12));  68 break; |

1. Halt() : shutdown\_power\_off()을 호출했다.

|  |
| --- |
| 87 void halt(void){  88 shutdown\_power\_off();  89  90 } |

1. Exit(): 테스트 통과할 때 필요한 스레드의 이름을 출력한다. 입력 받은 status를 스레드에 저장하고 종료한다.

|  |
| --- |
| 91 void exit(int status){  92 //debug  93 printf("%s: exit(%d)\n",thread\_name(),status);  94 thread\_current()->child\_status = status;  95 thread\_exit();  96  97 } |

1. Exec(): child process 를 만드는 함수이다. Process\_execute를 호출한다.

|  |
| --- |
| 99 pid\_t exec(const char \*file\_name){  100  101 return process\_execute(file\_name);  102  103 } |

1. Wait(): process\_wait()함수를 호출한다. Process\_wait 함수를 수정할 필요가 있다. Process\_wait 함수는 userprog/process.c에 있다. 자식 프로세스가 끝날 때까지 기다려야하기 때문에 자식 스레드를 알고 있어야 한다. 이때 필요한 thread 구조체를 수정한다. 우선 Threads/thread.h와 thread.c를 수정해야한다. Thread.c에서 init\_thread 함수를 수정한다. 이때 동기화 기법을 사용해야 하는데, thread 폴더의 synch.h에 구현된 코드인 semaphore를 보면 value와 waiters가 있다. Value는 공유자원의 개수이고 waiters는 공유자원을 사용하기 위해 대기하는 waiter의 list이다. 공유자원을 사용하고자 한다면 sema\_down을 실행한다. 공유자원이 없다면 리스트에 list\_push\_back 함수로 맨 뒤에 푸쉬한다.

Thread.h에 필요한 구조체들을 선언한다.

|  |
| --- |
| int child\_status; //child의 load를 나타내고 0이면 아직 load가 되지않음  101 //1이면 성공 -1이면 실패  102 struct semaphore child\_sema;  103 struct semaphore memory\_sema;  104 struct list children;  105 struct file \*exec\_file;  106 struct list\_elem child\_elem; |

Thread.c에 필요한 코드를 추가한다. Init\_thread에 추가한다. Sema\_init은 semaphored의 value를 입력받은 값으로 초기화하고 list도 초기화한다. List\_init는 대기자를 초기화하는 함수이고 list\_push\_back은 맨 뒤로 보내는 함수이다.

|  |
| --- |
| 469 intr\_set\_level (old\_level);  470  471 #ifdef USERPROG  472 sema\_init(&(t->child\_sema),0);  473 sema\_init(&(t->memory\_sema),0);  474 list\_init(&(t->children));  475 list\_push\_back(&(running\_thread()->children),&(t->child\_elem));  476  477 #endif |

그리고 userprog/process.c를 수정한다. 현재 스레드 정보를 통해 자식 스레드를 관리해야 한다. 자식 스레드를 확인하고 종료하기 전 sema\_down을 한다. 리스트를 제거하고 sema\_up을 한다. 정상 종료 시 child의 load인 child\_status를 반환한다.

1. Read(): syscall.c에서 STDIN=0이면 입력 받은 크기만큼 버퍼에 저장하고 ‘\0’이 오면 멈춘다. 이때 input\_getc()가 ‘\0’일 때 break 한다.
2. Write(): syscall.cd서 SIDOUT=1이면 입력 받은 크기만큼 출력하고 크기를 반환한다. putbuf함수를 이용한다.
3. Additional System calls

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

우선 src/lib/user 에서 syscall.h를 수정했다. fibonacci함수와 max\_of\_four\_int의 prototype을 만들었다. lib/user/syscall.c 에서 syscall4를 만들었다. 인자 4개를 받을 때 수행하는 시스템 콜이다. 같은 파일 내에서 fibonacci는 인자가 1개이기 때문에 syscall1을 호출하고 max\_of\_four\_int는 4개의 인자가 들어오기 때문에 syscall4를 호출한다. Lib/syscall-nr.h에 가서 syscall number를 부여 해야한다. 따라서 syscall-nr.h enum에 SYS\_FIBONACCI 와 SYS\_MAX를 추가했다. Userprog/syscall.h에 돌아가서 구현할 2개의 함수의 prototype을 작성한다. Userprog/syscall.c에서 syscall handler와 두 함수를 구현한다.

|  |
| --- |
| 134 int fibonacci(int a){  135 if(a==0) return 0;  136 else if (a==1) return 1;  137 else return fibonacci(a-1)+fibonacci(a-2);  138 }  139  140 int max\_of\_four\_int(int a, int b, int c ,int d){  141  142  143 int max = a;  144  145 if(max < b) max = b;  146 if(max < c) max = c;  147 if(max <d) max= d;  148 return max;  149  150  151 } |

Pintos/src/example에서 additional.c 파일을 만들어서 입력이 들어오면 출력을 어떻게 할지 코드를 작성한다. 이때 syscall의 Fibonacci, max\_of\_four\_int를 호출한다.

|  |
| --- |
| 7 if(argc !=5) return -1;  8  9 printf("%d ",fibonacci(atoi(\*(argv+1))));  10 printf("%d \n",max\_of\_four\_int(atoi(\*(argv+1)), atoi(\*(argv+2)),  11 atoi(\*(argv+3)), atoi(\*(argv+4))));  12  13 return 1; |

Examples 내 Makefile을 수정해야 compile이 된다.

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