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

담당 교수 : 김영재

학번 : 20191638

이름 : 임형준

개발 기간 : 10-01~ 10-07

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

이번 프로젝트인 user program(1)의 목표는 pintos에서 user program이 동작할 수 있도록 하는 필수 요소들을 구현하는 것이다.

크게 ‘Argument passing’, ‘System call handler’ , ‘user memory access’로 볼 수 있다.

이를 위해서는 pintos의 code level flow에 대한 이해가 필요하며, system call, system call handler, argument passing, user stack 등에 대한 개념의 이해가 필요하다.

이러한 이해를 통해 운영체제가 어떻게 user program을 실행하는 지를 도출하여 이를 코드 기반으로 구현하는 것이 이번 프로젝트 개발의 목표이다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

Argument passing을 구현하기 위해서는 먼저 Command-line Passing을 통해 token 단위로 나누어야 하고, 이를 stack에 set up 하는 과정을 거쳐야 한다.

Command-line Passing에서는 User program에게 입력값으로 주어지는 Command-line을 공백 즉 space 문자를 기준으로 argument로 token화 하는 것이 요구하는 결과이다.

또한 이러한 token들을 convention에 맞게 stack에 push 연산을 수행하는 것이 목적이다.

1. User Memory Access

운영체제에서 사용하는 메모리에는 user memory와 kernel memory로 이뤄져 있다. System call을 구현하기 위해서는 user memory에 접근하여 데이터를 읽고 쓰는 기능이 필요하다. 이는 virtual memory space라는 개념을 통해 구현되어 각 프로세스의 메모리와 운영체제의 메모리에 대한 추상화가 가능하다. 단 비정상적인 메모리에 대한 접근은 모두 제한하여야 하며, 이는 user memory가 아닌 kernel memory에 대한 접근을 의미한다.

즉 우리는 주어진 포인터가 유효한 주소 (user memory)가 맞는 지를 보장하는 기능을 구현해야 한다. 또한 읽고 쓰기 위해 접근하려는 메모리 영역이 kernel memory를 침범하지는 않는 지를 보장하는 기능 또한 구현해야 한다.

만약 접근하려는 메모리 주소가 user memory가 아니라 kernel memory일 경우, 이는 비정상적인 access이므로 page fault를 통해 이를 잡아내는 기능 또한 필요 할 것이다.

1. System Calls

System call은 user program이 kernel에게 특정한 작업을 수행하도록 요청하는 interface이다.

이번 프로젝트에서 구현해야 할 system call은 halt, exit, exec, wait 이며,  
make check 명령어를 통해 제시된 test들을 정상적으로 수행하여 통과하는 것을 확인하였다.

추가적으로 Thread 간의 race issue를 방지하기 위해 Synchronization를 적용하였다.

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

User Program에서 입력된 Command-line을 실행하기 위해서는 크게 명령어 이름과 인수로 구분하여야 한다. 이러한 과정을 argument passing으로 구현하였다. 이는 Command-line을 공백 문자를 기준으로 argument로 분리하는 기능이다. 이러한 argument를 정확하게 parsing 하였는 지 확인하는 절차를 거친 후 setup\_stack()를 호출하여 stack에 memory를 할당하여 확장한다.

Stack pointer인 esp를 통해 stack에 저장된 top element를 조회할 수 있으며,

이를 활용해 stack에 argument를 push하여 configuration을 진행한다.

주요한 특징들을 정리하면 다음과 같다.

Stack은 LIFO(last out, first in)의 구조를 가지고 있다. 때문에 argument의 push 연산 순서는 역순으로 진행된다.

먼저 argv에 저장된 argument의 값들을 push한다. 동시에 push한 argument의 total byte number를 저장한다. 이는 word alignment를 처리하기 위함이다.

argv를 모두 push하면, total byte number가 4배의 배수가 될 때까지, 0을 push한다.

그 다음 argument들의 주소값을 저장한다. 역시나 역순으로 저장되며, 맨 처음엔 NULL값인 ‘\0’을 먼저 push하고 그 뒤에 argv의 주소를 push한다.

Argument의 개수를 저장한 변수인 argc를 stack에 push한다.

Argv의 시작 주소를 그 다음 push한다. 마지막으로 return address를 push한다.

stack에 어떠한 자료형으로 casting하여 저장해야 할 지, 몇바이트만큼 esp의 주소를 증가 혹은 감소시켜야 올바른 값을 저장할 수 있을 지를 잘 고려하는 것이 중요하다.

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

먼저 invalid memory access에는 다양한 상황이 존재한다.

* + 1. Null pointer를 역참조하는 것.
    2. User program이 kernel memory 영역에 접근하는 것.
    3. Unmapping memory에 접근하는 것.

이번 프로젝트에서는 이러한 상황을 방지하는 것에 초점을 맞춰져 user program이 이러한 접근을 하려하면 방지하는 기능을 구현한다.

Valid memory access가 보장되지 않는다면 User program이 memory에 저장되어 있는 다양한 data의 오염을 예기할 수 있다.

* + Invalid memory access를 어떻게 막을 것인지 설명

pintos에는 valid memory checking에 대한 함수를 제공한다.

is\_user\_vaddr(), is\_kernel\_vaddr() 이 두 함수는 parameter로 넘겨받은 address가 user virtual memory 영역에 존재하는지, kernel virtual memory 영역에 존재하는 지를 알려준다.

텍스트, 도표, 스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

PHYS\_BASE를 통해 이를 구분할 수 있다. 이 변수는 process virtual memory의 3GB 지점을 저장한다. 위의 두 함수는 PHYS\_BASE를 기준으로 인자로 넘겨받은 address가 위에 존재하는 지 아래에 존재하는 지를 확인하여 결과를 알려준다.

특히 이러한 기능은 memory에 빈번하게 접근하는 system call에서 반드시 구현되어야 한다. 또한 exception handler에서 page fault를 다루기 위해 구현되어야 한다. 접근하려는 address가 kernel memory space에 존재하면 page fault를 발생시켜 exit(-1)과 함께 강제 종료시키도록 구현하였다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명

System call은 User program에서 OS의 특정한 작업을 수행하기 위해 요청하는 interface이다. User program은 kernel memory에 접근할 수 없기 때문에, OS에서 수행하는 기능이 제한된다. OS는 program을 user mode에서 kernel mode로 변경하여 특정 작업을 수행할 수 있도록 한다. 이러한 kerner mode로 진입하게 해주는 것이 system call이다. 이렇게 mode를 나누어 접근 권한을 부여하는 이유는 시스템과 OS를 보호하기 위함이다. 이러한 보호성을 유지하면서 작업은 수행할 수 있도록 하는 것이 system call이기 때문에 매우 중요한 요소라 할 수 있다.

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

6개의 system call과 2개의 addition system call을 구현하는 것이 이번 프로젝트의 목표이다. 설명은 다음과 같다.

1. halt : OS를 종료하는 system call이다. shutdown\_power\_off() 함수를 통해 간단하게 구현할 수 있다.
2. exit : 현재 실행되고 있는 user program을 종료하는 system call이다. exit status을 kernel에 return하여 정상 종료인 지, 모종의 이유로 비정상적으로 종료되었는 지를 알린다.
3. exec : Command line을 인자로 넘겨받고 포함되어 있는 file name을 이름으로 가지는 thread를 생성한다. fork와 exec의 기능이 동시에 구현되어 있는 system call이다.
4. wait : child process가 종료될 때까지 parent process가 기다리도록 하는 system call이다. synchronization을 통해 race condition을 방지하고 데이터의 일관성을 보장한다.
5. read : file descriptor에서 size만큼의 byte를 읽어 buffer에 저장하는 system call이다. 이번 프로젝트에서는 standard input에 대한 read의 기능만을 수행하도록 구현되었다.
6. write : buffer에서 file desciptor에 data를 size만큼의 byte를 출력하는 system call이다. read와 마찬가지로 standard output의 write 기능만 구현하였다.
7. Fibonacci : 넘겨 받은 integer에 해당하는 순서의 Fibonacci 값을 계산하여 return하는 system call이다. recursion을 활용하는 방식이 일반적이지만 이번 프로젝트에서는 단일 for문으로 계산하도록 구현하였다.
8. max\_of\_four\_int : 인자로 넘겨받은 4개의 integer 중 가장 큰 값을 return하는 system call이다. 계속해서 가장 큰 값을 갱신하는 방식으로 구현하였다.
   * 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명
9. User application program에서 write, read, open과 같은 library fucntion을 호출한다. 이러한 함수들은 wrapper function로 실제로 이러한 함수들은 system call interrupt handler를 동작시키는 역할만 수행한다.
10. wrapper function에서 signal을 보내면서 같이 system call number, parameter 등을 같이 넘겨준다. 이는 stack에 arguments, argc, argv, return address 등을 저장하는 방식으로 전달되며, interrupt vector table에서 system call number에 해당하는 system call을 실행시킨다.
11. interrupt와 trap을 통해 OS에 kernel mode로의 전환을 요청하고 system call을 수행한다. 이러한 수행의 결과를 eax와 esp에 저장하여 return한다.

**추진 일정 및 개발 방법**

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

1) argument passing (10-01 ~ 10-02) command line passing과 stack set-up을 구현하였다. 강의 자료와 manual에서 제공된 표를 esp hex\_dump를 비교하며 올바르게 passing되는 지 확인하고 올바른 address에 argument가 저장되어 있는 지를 확인하는 과정을 거쳤다.

2) system call (10-03 ~ 10-07) halt와 같이 간단한 함수들은 하루만에 전부 구현하였다. system call에 paremeter를 전달하는 과정이 오래 걸렸다. unvalid address checking은 제공된 함수로 쉽게 구현하였기에 어려운 과정이 아니였다.

synchronization을 구현하기 위해, thread.c와 thread.h에 struct를 수정하고 wait와 exit에 적용하는 과정이 있었다.

**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 : process.c 내의 load 함수에 command line을 parsing 하는 요소를 추가한다. loading하는 과정을 거치고 난 후 stack에 argument들을 저장하는 함수, argument\_stack을 추가하였다.
    2. system call : system-nr.h에서 구현할 system call number들을 enum 형태로 저장한다. system.c에서 syscall\_handler 함수에 switch의 형태로 먼저 user virtual address checking을 위한 check\_user\_addr() 함수를 구현하였다. 이는 제공되는 is\_user\_vaddr()로 구현한다. 그 후 system call number에 따라 해당하는 wrapper function을 호출한다. syscall\_handler 함수 밑에 wrapper function가 구현되어 있다.
    3. synchronization : thread.h에서는 thread struct에 semaphore인 child\_lock과 parent\_lock을 선언한다. 종료 상태를 반환하기 위한 exit\_status를 선언한다. thread.c에서 init\_thread 함수 내의 #ifdef USERPROC 영역안에 semaphore를 초기화 하는 sema\_init을 호출하고 list\_init을 호출해 thread의 child thread list을 초기화한다.
    4. exception : user program이 kernel memory에 접근하는 것을 방지하기 위해 exception.c내의 page\_fault 함수에서 is\_kernel\_vaddr 함수를 호출해 이를 checking 한다.
    5. process execute, wait, exit : 출력 양식에 맞추기 위해 process.c 내의 process\_execute에서 인자로 전달받은 cmdline에 있는 file\_name만 parsing 하는 요소가 추가되었다. process\_exit에서 synchronization을 위해 sema\_up(parent\_lock)과sema\_down(child\_lock)이 추가되었다. process\_wait에는 현재 실행되고 있는 thread의 child list을 탐색하며 child가 종료되어 exit\_status를 return할 때까지 기다리도록 한다. sema\_down(parent\_lock), sema\_up(child\_lock)으로 이를 감싸 부모 프로세스가 먼저 종료되지 않도록 하였다.

1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

1. Argument Passing

텍스트, 스크린샷, 도표, 폰트이(가) 표시된 사진

자동 생성된 설명

1. User Memory Access

텍스트, 스크린샷, 폰트, 도표이(가) 표시된 사진

자동 생성된 설명

1. System Calls

텍스트, 스크린샷, 폰트, 도표이(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.**

1. Argument Passing

ppt에서 안내한 내용에 따라 process.c 파일에 load 함수에서 stack set up과 parsing 하는 기능을 구현하였다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

strtok\_r 함수를 통해 공백을 기준으로 token을 만들고 parsed\_cmd 함수에 저장하였다. 자연스럽게 parsed\_cmd[0]은 command line에서 file name만을 가지게 된다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

가독성을 위해 일반적으로 argument를 담는 argv에 다시 저장하였으며 argument count를 의미하는 argc로 계산해주었다. 정상적으로 parsing이 완료되었다면, stack을 set\_up하는 기능을 수행하는 argument\_stack 함수를 호출한다. 이때 argument array인 argv와 count인 argc, 스택 포인터인 esp를 인자로 넘겨준다.

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

자동 생성된 설명stack에는 각 argument들의 주소를 저장해야 하기 때문에 arg\_addr 배열을 선언한다.

argv의 값들을 stack에 저장하면서 address들을 arg\_addr에 저장한다.

stack에 data를 쌓을 때 주의할 점은 data의 크기에 맞게 pointer를 이동하고 저장하는 것이였다. 이를 위해 memcpy를 통해 정확한 크기만 메모리 복사가 이뤄지게 하였다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

argument의 값들을 stack에 저장하고 나면 stack figure에 맞추기 위해 word alignment를 구현하였다. memset 함수를 통해 argument size의 합인 total\_len이 4의 배수가 될 때까지 0을 저장하였다.

stack set up의 핵심은 type casting 이였다. esp는 double pointer이므로 \*esp는 여전히 pointer이다. 이를 (uint32\_t) 즉 4byte로 casting 하여 다시 asterisk를 붙여야 그 주소가 가르키는 값을 가져올 수 있다.

1. User Memory Access

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

syscall.c에서 memory에 접근할 때 user memory 영역임을 checking 하는 함수인 check\_user\_addr를 구현하였다.

가독성을 높이기 위해 몇 바이트만큼을 검사할 것인지 인자 int i를 통해 그 범위를 입력받는다.

텍스트, 폰트, 스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 폰트, 라인이(가) 표시된 사진

자동 생성된 설명

각 system call case마다 parameter로 전달되는 esp pointer내의 값들에 대해 address checking 과정이 이루어진다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

더불어 exception.c 파일의 page\_fault 함수에서도 user memory access checking을 검사한다.

1. System Calls

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

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

이전에 이미 설명된 내용이므로 각 system call의 기능에 대한 설명은 생략한다.

syscall\_handler 함수가 실행됐을 때, esp가 가르키고 있는 값이 system call number이므로 이 값과 syscall-nr.h에 저장된 enum 형태의 system call과 비교해 switch문을 실행시켜 일치하는 case를 실행한다.

작업을 수행하고 난 뒤 결과에 해당하는 return value는 manual에 의거해 eax pointer에 저장하도록 하였다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

실질적으로 작업을 수행하는 함수들의 정의이다. 단순히 다른 함수를 호출하는 wrapper function의 꼴을 가지고 있다.

exec 함수는 process\_execute를 출력하고 그 return값을 다시 return하는데 이 함수는 process.c 파일에 정의되어있다.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

const 형태로 command line을 인자로 받기 때문에 이를 copy\_str에 strlcpy 함수를 통해 복사한다. 그 뒤 strtok\_r 함수로 file name에 해당하는 값을 token에 저장하고 이를 thread\_create에 넘겨 실행한다.

이전까지는 ‘echo x’를 실행하면 thread name이 ‘echo x’ command line 전체가 되었으나 정상적으로 echo만 thread name으로 가지게 된다.

wait 함수 또한 process\_wait를 실행하는 함수이다. wait와 특히 exit의 구현을 위해서는 synchronization의 적용이 필요하다.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

thread.h 파일내에 존재하며, 부모 thread와 자식 thread의 semaphore를 생성하였다. 각 thread마다 존재해야 하기 때문에 thread 구조체 내에서 수정하였다.

또한 자신이 생성하는 child들을 저장하기 위한 list와 thread struct들을 list로 저장하는 elem 형태의 list를 추가하였다.

이는 당연히 user program에서만 적용되어 하므로 #ifdef USERPROC 내부에 작성되었다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

그 다음 thread.c에서 init\_thread 함수 내에 semaphore와 list들을 초기화하도록 구현했다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

child list를 순회하며 wait의 target이 되는 child tid를 찾는다.

찾게 되면 sema\_down 함수를 통해 parent\_lock을 걸게된다. 즉 current\_process에 P 연산을 한 것이다. 그럼 언제 remove 작업을 수행하게 될까?

같은 process.c 파일에 정의되어 있는 process\_exit 함수를 살펴보자

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

child process가 모든 작업을 수행하고 종료하게 된다면 parent\_lock에 sema\_up 즉 V 연산을 수행하고 child\_lock에 P 연산을 하게 해 parent process를 깨우고 child process를 재우게 된다.

이 시점에서 다시 돌아와서

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

list에서 child process를 제거한다. 즉 reaping을 수행하는 것이다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명  
read 함수에서는 input\_getc를 통해 한 문자씩 값을 입력받아 buffer에 저장하고 저장한 data의 개수를 return한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

write함수는 putbuf 함수를 통해 size만큼 한꺼번에 출력한다.

1. Additional System calls

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

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

먼저 syscall.c 내부의 switch문에서 인식할 수 있도록 system-nr.h 내부의 enum에 additional system call의 이름을 추가해준다.

max\_of\_four 함수는 특히나 인자를 4개 받는 system call이므로 이에 대한 정의가 필요하다

pintos/src/lib/user/syscall.c에서 이미 정의되어 있는 syscall들을 참고해 syscall4를 새로 정의하였다.

텍스트, 스크린샷이(가) 표시된 사진

자동 생성된 설명

pintos/src/lib/user/syscall.h에 addition function에 대한 정의를 추가한다.

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

다시 pintos/src/lib/user/syscall.c 에 이에 대한 wrapper 함수를 추가한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

userproc/syscall.c에 정의된 call routine function은 다음과 같다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

fibonacci는 loop 문을 이용해 구현하였으며

max\_of\_four\_int는 최대값을 갱신하는 형태로 구현하였다.

정상적으로 작동하는 지 확인하기 위해 example 파일을 만들었다.

src/examples에 additional.c를 추가하여 다음과 같이 구현하였다.

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

또한 additional 파일을 컴파일하기 위해 makefile 또한 수정하였다.

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

* 1. **시험 및 평가 내용**
* **fibonacci 및 max\_of\_four\_int 시스템 콜 수행 결과를 캡처하여 첨부.**
* **텍스트, 스크린샷, 폰트이(가) 표시된 사진

  자동 생성된 설명**
* **fibonacci와 max value가 정상적으로 출력되는 것을 볼 수 있다.**
* **텍스트, 폰트, 스크린샷, 타이포그래피이(가) 표시된 사진

  자동 생성된 설명**텍스트, 폰트, 스크린샷, 흑백이(가) 표시된 사진

  자동 생성된 설명
* addition system call이 아닌 test의 결과이다 채점 test 21개 전부 pass하였다.