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

담당 교수 : 박성용

조 / 조원 : 이우진

개발 기간 : 2023.09.18 – 2023.10.08

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

본 프로젝트에서는 OS가 수행하는 중요한 기능 중 하나인, 프로그램의 실행을 위한 argument passing과 system call의 구현을 목적으로 한다. 프로그램을 수행함으로써 pintos 상에서 프로그램을 실행할 수 있게 되고, 프로그램들이 system call을 호출할 수 있다. 이 때, user가 적절한 메모리 공간을 참조하려고 하는지 확인하여 잘못된 메모리 공간을 참조 시 에러를 발생시킬 수 있도록 구현한다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

Argument passing을 구현함으로써 pintos상에서 user가 Input로 입력한 값을 적절히 parsing하여 실행하고자 하는 프로그램의 이름과 프로그램에 전달하고자 하는 argument을 얻는다. 이후 parsing한 데이터를 stack 영역에 적절한 형태로 push하여 pintos 상에서 프로그램이 argument들을 이용해 실행될 수 있다.

1. User Memory Access

User Memory Access을 구현하여, kernel 이 user program에 의해 주어진 메모리 주소에 대한 참조를 할 때마다 잘못된 메모리 값이 아닌지 확인한다. 참조하고자 하는 주소가 적절하지 않은 메모리 주소인 경우 현재 process을 종료하여 메모리를 한다. 이를 통해 kernel과 다른 process의 메모리 공간을 보호할 수 있다.

1. System Calls

OS가 수행하는 중요한 기능 중 하나인 system call을 구현한다. IO와 같이 user program에서 필요한 기능이지만, 안정성으로 인해 user 영역에서 실행되어서는 안되는 기능들을 OS에서 제공하여 실행할 수 있다. 본 프로젝트에서 구현하는 system call은 halt, exit, exec, wait, read, write와 추가로 n번째 피보나치 수를 반환하는 Fibonacci, 4개의 int 수 중에서 가장 큰 수를 반환하는 max\_of\_four\_int 이다.

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

Pintos 상에서 프로그램을 실행하게 되면 프로그램을 실행하기 위한 stack 영역을 할당하고, user가 입력한 문자열을 적절히 parsing하여 해당 stack 영역에 쌓는다. 이 때, stack 영역에 쌓아야 하는 값은 순서대로 다음과 같다.

* + 1. Parsing한 null 문자를 포함한 문자열을 push한다. 이 때, 문자열의 참조는 추후에 push할 문자열 주소의 포인터를 이용해 이루어지므로 문자열을 push하는 순서는 크게 중요하지 않다.
    2. 앞에서 모든 문자열을 push한 이후, 만일 stack의 주소가 4의 배수가 아닌 경우, word alignment을 위해 stack의 주소가 4의 배수가 될 때까지 stack영역을 round 해준다.
    3. 앞에서 push한 문자열의 stack상에서의 주소를 push한다. 순서는 사용자가 입력한 순서의 반대, right-to-left의 순서로 push한다. 예를 들어, 사용자가 ‘/bin/ls -l foo bar’와 같이 입력했을 경우, ‘bar’의 주소, ‘foo’의 주소, ‘-l’의 주소, ‘bin/ls’ 의 순서로 push한다. 이 때, 가장 처음에 null 주소 값을 push하여 argument 주소 영역이 끝났음을 알려야 한다
    4. 앞에서 문자열의 주소를 push하였을 때 마지막으로 push한 주소의 스택 상에서의 주소를 push하고, argument의 개수를 push한다.
    5. 마지막으로, fake return address로 null pointer을 push한다.

Argument의 스택 상에서의 주소를 담는 배열을 argv라고 하였을 때, 3)에서 push한 값은 argv의 값이며 4)에서 push한 값은 argv[0]의 주소 값과 argv에 담긴 argument의 개수인 argc라고 볼 수 있다. 결과적으로 stack에 저장된 값의 예시는 다음과 같다.

텍스트, 폰트, 메뉴, 스크린샷이(가) 표시된 사진

자동 생성된 설명

* User Memory Access
  + Pintos 상에서의 invalid memory access 개념을 간략히 설명
  + Invalid memory access를 어떻게 막을 것인지 설명

Pintos 상에서, kernel은 user program에서 전달한 memory pointer을 이용하여 memory access을 수행한다. 이 때, user program에서 전달한 pointer가 null pointer 이거나, 매핑 되지 않은 pointer이거나, kernel 상의 메모리 공간의 pointer인 경우에는 invalid memory access로 보고 kernel 공간 또는 다른 process의 메모리 공간을 보호하기 위해 참조해서는 안된다.

User program에서 전달한 memory pointer가 적절한 invalid pointer인지 확인하기 위해, pintos에서 제공하는 함수인 pagedir\_get\_page( ), is\_user\_addr( ), is\_kernel\_addr( )을 이용한다. 위 함수들을 이용해 pointer가 참조하는 메모리 공간이 unmapped virtual memory 영역인지, user에 할당된 영역인지, kernel에 할당된 영역인지 각각 확인할 수 있다. User로부터 제공받은 stack 상에서의 주소들이 적절한 주소인지 위 함수를 이용하여 확인하고, page\_fault( ) 함수에서 계산한 값을 이용해 적절한 주소인지 확인하여 Invalid memory access을 막을 수 있다.

* 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

IO, file 접근, process 관리와 같이 OS에서 제공하는 기능을 모두 user program 상에서 직접 실행할 수 있게 되면 잘못된 메모리 참조로 인한 심각한 오류가 발생할 수 있고 악의적인 user program으로 인해 전체 시스템을 사용하지 못하게 될 수 있다. 따라서, user program에서 실행하더라도 안전한 기능은 user program에서 실행할 수 있도록 하고, 그렇지 않은 기능은 user의 요청을 받아 kernel 영역에서 실행될 수 있도록 하여 위와 같은 문제가 발생하지 않도록 할 수 있다.

이번 프로젝트에서는 halt, exit, exec, wait, read, write system call과 추가적으로 Fibonacci, max\_of\_four\_int system call을 구현한다. 각각의 기능은 다음과 같다.

* + - halt : Pintos을 종료하는 system call이다.
    - exit : 현재 pintos 상에서 수행중인 user program을 종료하고, kernel에 status을 전달한다.
    - exec : child process을 생성하고 pintos 상에서 새로운 user program을 실행한다.
    - wait : 지정한 child process의 수행이 완료될 때 까지 현재 process을 중지한다. Child process가 존재하면 해당 process의 exit status값을 반환하고, 존재하지 않다면 -1을 반환한다.
    - read : 지정한 file에서 값을 읽어 들인다. 본 프로젝트에서는 stdin에서의 read 만을 구현한다.
    - write : 지정한 file에 값을 write한다. 본 프로젝트에서는 stdout에서의 write만을 구현한다.
    - fibonacci : n번째 피보나치 값을 반환한다.
    - max\_of\_four\_int : 네 개의 int 값 중 가장 큰 값을 반환한다.

User program에서 system call을 위해 함수를 호출하면, 함수 내에 구현된 코드가 syscall interrupt을 발생시킨다. Interrupt가 발생되었으므로, control이 kernel 로 넘어간다. Kernel에서는 먼저 interrupt vector에서 발생한 interrupt가 어떤 interrupt인지 확인하여 적절한 interrupt handler을 실행하는데, syscall의 경우 syscall\_handler이 실행된다. Syscall handler에서 user로부터 요청 받은 syscall을 수행하고, 값을 반환하며 control이 user 영역으로 다시 넘어가면서 syscall이 종료된다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

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

본 프로젝트에서는 Argument Passing, User Memory Access, System call 세 가지의 개발을 수행하므로, 이를 1주 씩 나누어 구현한다.

* 9/18 ~ 9/24 : 가장 중요한 요소인 Argument Passing 부분을 구현한다. Pintos 상에서 프로그램을 실행하기 위해서는 Argument Passing이 구현 되어있어야 하지만, 현재 구현이 되어 있지 않으므로 가장 먼저 구현하여 기본적인 program을 실행할 수 있도록 한다. 구현해야 할 요소는 parsing 파트, stack 파트로 나눌 수 있는데, parsing한 값을 stack에 push해야 하므로 순서대로 구현한다.
* 9/25 ~ 10/01 : Fibonacci, max\_of\_four\_int을 제외한 System call을 구현한다. 특히, wait system call과 exit system call이 구현되어야 결과를 제대로 확인할 수 있으므로 가장 먼저 구현하고, 이후 다른 system call을 구현한다. User memory access의 경우, 우선 구현하지 않더라도 system call의 구현이 가능하므로 기본적인 부분만 구현해두고 추후 필요한 모든 기능을 구현한다.
* 10/02 ~ 10/08 : Page fault( ) 함수를 수정하고, 아직 구현이 되어 있지 않은 invalid pointer에 대한 확인을 구현한다. 추가적인 system call인 Fibonacci, nax\_of\_four\_int 또한 구현한다.
  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)**

각각의 요소에 대해, 추가해야 할 요소는 다음과 같다.

1. Argument Passing

Argument Passing에 있어 가장 중요한 요소는 Parsing과 Stack 구성이므로, 각각 parsing을 수행하는 parse\_file\_name( ) 함수와 stack을 구성하는 push\_args\_stack( ) 함수를 userprog/process.c 파일에 추가한다. 이를 user program에 필요한 영역을 구성하는 load 함수에서 실행하는 함수를 추가한다.

1. User Memory Access

userprog/syscall.c에 user가 제공한 pointer가 valid한 pointer인지 확인하는 check\_addr\_validity 함수를 추가한다. Stack 영역을 참조할 때마다, 해당 함수를 호출하여 valid한 메모리 영역인지 확인할 수 있다. 이후 userprog/exception.c 함수에서 user\_pointer가 PHYS\_BASE 아래 영역을 참조하는지 확인하는 코드를 추가한다.

1. System calls

userprog/syscall.c상에서, 각각의 syscall을 수행하는 함수를 작성하고 syscall\_handler에서 user가 요청한 syscall의 종류를 확인하여 적절한 함수를 호출하도록 하는 코드를 추가한다. 이 때, wait system call의 수행을 위해 threads/thread.h에 child process의 정보를 저장하기 위한 child struct을 추가하는데, semaphore을 포함하여 process가 종료 할 때 parent process에 알릴 수 있도록 한다. Thread에 child struct을 저장하기 위한 list 변수 또한 추가한다. 그리고, child struct에 관련된 값을 초기화 하기 위한 코드를 thread\_create( ) 함수에 추가한다. 마지막으로, wait system call과 exit system call의 실제 구현부라고 할 수 있는 userprog/process.c 의 process\_wait( ) 함수와 process\_exit( ) 함수에 관련된 코드를 추가한다.

1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

1. Argument Passing

**stack을 구성하기 까지의 flow chart**

**텍스트, 스크린샷, 폰트, 디자인이(가) 표시된 사진

자동 생성된 설명**

**parse\_file\_name( ), push\_args\_stack( )**

**텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명**

1. User Memory Access

**check\_addr\_validity( )**

**텍스트, 스크린샷, 폰트, 디자인이(가) 표시된 사진

자동 생성된 설명**

**Page\_fault( )**

**텍스트, 스크린샷, 폰트, 디자인이(가) 표시된 사진

자동 생성된 설명**

1. System Calls

텍스트, 스크린샷, 폰트, 그래픽 디자인이(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

Argument Passing을 구현하기 위해, 먼저 파싱을 위한 parse\_file\_name( ) 함수를 구현하였다. Parse\_file\_name( ) 함수는 사용자가 입력한 문자열과, 파싱한 결과를 저장하기 위한 배열 argv을 인자로 입력 받고, strtok\_r( ) 함수를 이용하여 파싱을 수행하여 argv 배열에 저장한다. Strtok\_r( ) 함수는 “string.h”에 포함된 함수로, 지정된 구분자의 위치에 null값을 저장하고 null값으로 구분된 token의 주소를 반환하는 함수이다. 본 프로젝트에서는 공백 문자를 기준으로 token을 구분하므로, 이를 이용해 strtok( ) 반복해 호출하면서 argv 배열에 저장해 파싱을 수행한다.

위에서 파싱한 결과를 이용해 스택을 구성하는 push\_args\_stack( ) 함수를 구현하였다. 이 때, esp 변수는 void pointer 변수이므로, 4byte만큼 esp 값을 내리기 위해 sizeof(char \*)만큼 빼 주어야 했고 int 값을 push하기 위해 (int \*) 형변환을, char pointer 값을 push하기 위해 (char \*) 형변환을 하는 등의 형변환이 필요했다. 또한, 스택에 저장한 argument에서의 stack 상에서의 주소를 저장하기 위한 배열인 argv\_addr 배열을 선언하여 argument 문자열을 저장할 때 stack 의 주소를 저장해 두고, 이후 argv\_addr 배열을 이용해 argument address을 추가하였다.

위에서 구현한 parse\_file\_name( ), push\_args\_stack( ) 함수를 load 함수에서 호출하였다. Parse\_file\_name( )의 경우 load 함수가 실행된 직후에 호출하였고, push\_args\_stack() 함수는 stack을 set up 해주는 setup\_stack( ) 함수의 호출 직후에 호출하였다. 위 과정 중 stack에 적절한 값이 push 되었는지는 현재 stack 내의 값을 출력해 주는 hex\_dump( ) 함수를 이용해 확인할 수 있었다.

1. User Memory Access

User Memory Access에서 invalid pointer을 확인하기 위해, ‘userprog/syscall.c’ 파일에 check\_addr\_validity( ) 함수를 추가하였다. 이 함수는 포인터를 인자로 받아 해당 포인터가 null pointer인지 확인한다. 이후, 포인터가 user 영역에 위치한 포인터인지 확인해주는 is\_user\_vaddr( ) 함수를 이용하여 포인터가 user 영역에 위치해 있는지 확인한다. 마지막으로, 포인터가 mapping된 page에 위치해 있는지 확인해주는pagedir\_get\_page( ) 함수를 이용하여 포인터가 적절한 위치에 있는지 확인한다. 만약 위 세 가지 테스트를 통과하지 못했을 경우, 아래에서 구현한 system call에서 sys\_exit( ) 함수를 호출하여 에러를 반환하며 process가 종료되도록 구현하였다.  
 Check\_addr\_validity( ) 함수는 아래의 system call 구현에서 사용하는 모든 stack 영역의 포인터에 대해 호출되어 적절한 포인터인지 검사하도록 하였다.

‘userprog/exception.c’ 파일에서는 page\_fault( ) 함수에서 page가 존재하는지, user에 의해 access 되는지에 관한 변수를 이용해 적절하지 않은 값이 결과로 나왔을 경우에는 마찬가지로 sys\_exit( ) 함수를 호출하여 process가 에러를 반환하며 종료하도록 구현하였다. 이 때, 종료하기 전에 명세서에 적힌 대로 f->eax 에 0xfffffff를 저장하고, f->eax에 저장되었던 값을 f->eip에 저장하였다.

1. System Calls

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

먼저 syscall\_handler에서 user가 요청한 system call의 종류를 확인하여 해당 system call에 필요한 코드가 수행될 수 있도록 구현해야 했는데 이 값은 f->esp 변수에 저장된 주소를 참조하여 얻을 수 있다. 함수가 실행된 직후 system call의 종류를 저장하는 syscall\_num에 저장하고, if-else if 문을 사용하여 해당하는 system call에 필요한 argument의 수에 따라 esp로부터 argument을 가져와 각각의 system call을 수행하는 함수를 호출하였다. 각각의 system call을 수행하는 함수들은 다음과 같이 구현하였다.

* halt : halt의 경우, 추가적인 argument가 필요하지 않으므로 추가적인 함수를 구현하지 않고 if문에서 즉시 구현하였다. Halt 시스템 콜을 구현하기 위한 함수는 pintos에서 제공하는 shutdown\_power\_off( )인데, 이 함수는 “devices/shutdown.h”에 구현되어 있으며 pintos을 즉시 종료하는 함수이다. 따라서, shutdown\_power\_off( )을 즉시 호출하여 시스템 콜을 구현하였다.
* write : write 시스템 콜을 구현하기 위한 sys\_write( ) 함수를 구현하였다. Write 시스템 콜에는 file descriptor, 입력할 문자열이 저장된 buffer, 입력할 문자의 개수인 size가 필요하므로, stack 상에서 argument을 가져와 sys\_write( ) 을 호출하였다. 본 프로젝트에서는 stdout 상에서의 write만 구현하는데, 이를 위한 함수는 “lib/kernel/console.c” 파일의 putbuf( ) 함수로 stdout으로의 write을 해주는 함수이다. 또한, 본래는 write( ) 한 실제 개수를 user에 반환해주어야 하지만, 본 프로젝트에서는 size로 고정되어 있으므로, f->eax에 size을 저장해주었다.
* read : read 시스템 콜을 구현하기 위한 sys\_read( ) 함수를 구현하였다. read 시스템 콜에는 file descriptor, 문자열이 저장될 buffer, 입력될 문자의 개수인 size가 필요하므로, stack 상에서 argument을 가져와 sys\_read( ) 을 호출하였다. 본 프로젝트에서는 stdin 상에서의 read만 구현하는데, 이를 위한 함수는 “devices/input.c” 파일의 input\_getc( ) 함수로 stdin에서의 read을 해주는 함수이다. 입력으로 0이 들어오지 않을 때 까지, 최대 size만큼 input\_getc( ) 함수를 호출하여 read을 수행하였고, 실제로 읽은 개수 만큼 f->eax에 저장하여 반환하였다.
* exec : exec 시스템 콜을 구현하기 위한 sys\_exac( ) 함수를 구현하였다. Exec 시스템 콜에는 실행하고자 하는 파일의 이름인 fname 한 가지의 argument가 필요하므로, stack 상에서 argument을 가져와 sys\_exac( )에 넘겨 주었다. Sys\_exec( ) 함수에서는 process\_execute( ) 함수에 file name을 넘겨주어 구현할 수 있다. Process\_execute( ) 함수는 ‘userprog/process.c’ 파일에 구현되어 있고, child process을 만들어 새로운 프로그램을 실행하는 함수이다.

추가로 구현할 wait과 exit 시스템 콜의 경우에는 현재 process와 parent process의 상호작용을 위해 “threads/thread.h” 파일에 child struct을 추가하였다. Child struct는 아래와 같다.

텍스트, 폰트, 스크린샷, 번호이(가) 표시된 사진

자동 생성된 설명

Child process에는 child을 식별하기 위한 tid, child가 exit 시스템 콜을 호출 하였을 때의 status을 저장하는 exit\_status, wait을 구현하기 위한 semaphore인 wait\_sema, 마지막으로 parent의 child\_list에의 저장을 위한 elem을 정의하였다. 추가로, thread가 자신의 parent process가 어떤 것인지 알 수 있도록 관련된 변수를 thread struct에 포함하였고 child\_list에서 tid 값을 가지는 child struct을 찾을 수 있도록 find\_child( )함수를 추가하였다. 이를 바탕으로, wait과 exit 시스템 콜을 구현하였다.

* exit : exit을 구현하기 위해 함수 sys\_exit( )을 구현하였다. Sys\_exit( )은 status을 인자로 인자로 받는데, 인자로 받은 status을 child struct에 저장하여 나중에 parent process가 wait을 통해 status 값을 얻을 수 있도록 구현하였다. 최종적으로, thread\_exit( ) 호출을 통해 process를 제거하였다.
* Wait : wiat 시스템 콜을 구현하기 위해 함수 sys\_wait( )을 구현하였다. Sys\_ wait( )은 기다리고자 하는 child process의 pid을 인자로 입력 받아 process\_wait( ) 함수를 호출한다. “userprog/process.c”에 구현된 process\_wait( ) 함수는 위에서 구현한 find\_child( ) 함수를 이용해 입력받은 pid을 가지는 child process의 child struct을 찾고, 해당 child\_sutrct의 semaphore에 대한 sema\_down( )을 시도한다. Semaphore은 child process가 시작할 때 0으로 초기화 되어 process\_exit( ) 가 호출되었을 때 sema\_up ( ) 함수가 호출되므로, child process가 종료될 때 까지 parent process는 실행을 멈추게 된다. 만약 child pid가 적절하지 않은 값이라면 -1을 반환했고, 아니라면 child process의 exit status 값을 반환하였다.d

1. Additional System calls

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

새로운 시스템 콜인 fibonacci, max\_of\_four\_int을 구현하기 위해서 먼저 ‘lib/syscall-nr.h’ 상에서 fibonacci, max\_of\_four\_int을 위한 새로운 system call number을 추가하였다.   
 시스템 콜을 user가 호출할 수 있도록 ‘lib/user/syscall.h’ 과 ‘lib/user/syscall.c’에 시스템 콜 함수를 추가 및 구현하였는데, 이 때 max\_of\_four\_int 시스템 콜의 경우 stack에 총 5개의 argument을 전달하여야 하는데, 총 4개의 argument을 전달하는 매크로 까지만 구현이 되어 있어 새롭게 syscall4( ) 함수를 추가하여 max\_of\_four\_int 함수에서 호출할 수 있도록 하였다.  
 마지막으로, ‘userprog/syscall.h’ 및 ‘userprog/syscall.c’ 에 새로운 system call에 관한 sys\_fibonacci( ) 함수와 sys\_max\_of\_four\_int( ) 함수를 추가하고 구현하였다.  
 pintos 상에서 해당 추가한 시스템 콜을 실행하기 위해, ‘example’ 디렉토리에 additional.c 파일을 추가하고, user가 입력한 argument의 개수에 따라 fibonacci 또는 sys\_max\_of\_four\_int 시스템 콜을 실행하도록 구현하였다.

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

*pintos --filesys-size=2 -p ../examples/additional -a additional -- -f -q run 'additional 10 20 62 40'*의 실행 결과 ( **max\_of\_four\_int ( 10, 20, 62, 40 )** )

텍스트, 스크린샷, 소프트웨어, 웹 페이지이(가) 표시된 사진

자동 생성된 설명

*pintos --filesys-size=2 -p ../examples/additional -a additional -- -f -q run 'additional 10'*의 실행 결과 ( **fibonacci ( 10 )** )

텍스트, 스크린샷, 소프트웨어, 웹 페이지이(가) 표시된 사진

자동 생성된 설명