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

담당 교수 : 김영재

조 / 조원 : 20161663 허재성

개발 기간 : 2021.09.10 ~ 2021.10.01

1. **개발 목표**

기본적으로 주어진 Pintos 프로젝트의 경우, User program이 실행될 때 사용자가 기대하는 동작을 수행하지 못한다. 이를 가능하게 하기 위해 미구현된 System Call, System call handler, argument passing, user stack을 구현해야 한다. 추가적으로 미리 정의되어 있는 System call 외에 추가적인 System call인 fibonacci와 max\_of\_four\_int를 정의하여 System call에 대한 이해를 더 깊게 한다.

1. **개발 범위 및 내용**
   1. **개발 범위**
2. Argument Passing

Argument Passing을 구현하지 않고 Pintos에서 userprogram을 실행시키면 page fault가 발생하여 프로그램이 비정상적으로 종료된다. Argument Passing 과정에서 입력 command를 적적히 parsing 후 user stack에 적재하면서 Page fault 발생을 방지하고 해당 실행 파일에서 System call을 호출할 수 있게 된다.

1. User Memory Access

User stack에 쌓인 인자의 주소가 User Virtual Memory(0 ~ 0xc000000)에 해당할 경우에만 System call이 호출되어야 한다. 이를 벗어나는 주소가 입력될 경우 Kernel Virtual Memory를 User program이 침범하지 않도록 비정상적인 종료를 시켜줘야 한다. 이를 구현하면 User program이 해당 프로세스의 User Virtual Memory에만 접근 가능하여 안정적이다.

1. System Calls

User Program이 요구하는 기능을 직접 수행할 수 없고, Kernel이 수행해야 할 경우 User Program에서 호출한다. 예를 들어 디스크에 존재하는 파일을 열어 그 내용을 출력하는 cat을 수행하기 위해서 User Program에서 system call open을 호출하여 파일을 열고 그 내용을 system call write를 호출하여 콘솔(또는 파일)에 출력할 수 있다. 이번 프로젝트에서는 file system에 관한 건 구현하지 않고 halt, exit, exec, wait, read, write(각각 STDIN, STDOUT)에 대하여 구현하여 filesystem의 기능이 필요한 것을 기능들을 수행할 수 있도록 하였다.

* 1. **개발 내용**
* Argument Passing
  + 입력 command(예를 들어 echo x)에서 실행 파일명과 인자를 parsing하여 마지막 인자부터 거꾸로 user stack에 쌓는다. Stack에 쌓기 위해 intr\_frame의 esp 포인터를 인자로 전달받아 이를 이용하여 인자가 쌓일 때마다 esp 값을 감소시켜가며 쌓는다. 인자를 다 쌓고, 인자의 주소와 인자를 저장한 배열의 주소, 인자 개수, return address를 모두 스택에 쌓는다.

User Memory Access

* + 여러 User process가 동시에 돌아가고 있을 때, 한 프로세스 A가 다른 프로세스 B의 User Virtual Memory에 접근할 경우 A, B 모두 정상적으로 작동하지 않을 수 있다. 더 심각한 경우로 A가 Kernel Virtual Memory에 접근할 경우 A, B 뿐만 아니라 시스템 전체에 치명적인 결함이 생길 수 있다. 이러한 접근이 invalid memory access이다.
  + Pintos에서는 User address를 0x0 ~ 0xc0000000(0~3GB), Kernel address를 0xc0000000 ~ 0x100000000(3~4GB)로 정해 놓았다. User program에서 Kernel address에 접근하는지 확인 후(threads/vaddr.h의 is\_user\_vaddr, is\_kernel\_vaddr 함수 이용) 접근할 경우 프로그램을 종료시키면 된다.
* System Calls
  + User Program에서 수행하려는 기능이 Kernel에서 수행할 수 있을 경우, User Program이 직접 Kernel에 접근할 수 없으므로 Kernel에 해당 기능을 대신 수행하도록 요청해야 한다. 이를 위해 System call을 구현해야 하며 System call을 구현하면 Kernel에서 User program이 호출한 System call에 따라 알맞은 기능을 수행한다. System call은 User 영역과 Kernel 영역의 매개체 역할을 한다.
  + halt : Pintos 자체를 종료시키는 System call이다. devices/shutdown.h의 shutdown\_power\_off 함수를 호출하여 종료시킨다.
  + exit : 현재 user process를 종료시키는 System call이다. threads/thread.h의 thread\_exit를 호출하여 종료시킨다. 종료될 때 종료 상태를 kernel에 전달한다.
  + exec : 실행 파일명을 인자로 받아 해당 프로그램을 실행시키는 System call이다. userprog/process.c의 process\_execute를 호출하여 실행시킨다. 해당 쓰레드의 tid를 kernel에 전달한다.
  + wait : 현재 process의 child process가 종료될 때까지 기다린다. userprog/process.c의 process\_wait을 호출하여 해당 child process가 종료될 때까지 기다리고 child trhead의 종료 상태를 kernel에 전달한다.
  + read : STDIN(콘솔 입력)으로 주어진 입력을 size만큼 읽어 buffer에 저장한다. device/input.c의 input\_getc 함수를 이용해 읽어들이며 읽은 바이트 수를 kernel에 전달한다.
  + write : STDOUT(콘솔 출력)으로 buffer에 저장된 내용을 화면에 출력한다. lib/kernel/console.c의 putbuf 함수를 이용해 출력하며 출력한 바이트 수를 kernel에 전달한다.
  + fibonacci : 입력으로 받은 n번째 피보나치 수를 계산하여 kernel에 전달한다.
  + max\_of\_four\_int : 입력으로 받은 4개의 정수 중 최댓값을 정하여 kernel에 전달한다. 이를 위해 lib/user/syscall.c에 4개의 인자를 받는 System call syscall4를 정의하고 사용한다.
  + User program에서 System call을 호출하면 lib/user/syscall.c의 해당 system call을 호출한다. System call이 호출되면 스택에 인자와 해당하는 system call 번호를 삽입 후 interrupt handler를 부른다. 이 경우 userprog의 syscall\_handler가 호출된다. syscall\_handler에서 system call 번호에 따라 구현한 system call을 호출한다. 이 때 intr\_frame의 esp를 이용해 user stack에 접근하여 stack에 쌓은 인자들을 이용한다. esp를 통해 kernel에 넘겨주고 kernel에서 받은 return 값을 eax를 통해 user에 넘겨준다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

2021.09.11~2021.09.20 : 명세서 분석, 요구 사항 파악, 코드 분석

2021.09.21~2021.09.25 : Argument Passing, User Memory Access, System call 구현

2021.09.26~2021.10.01 : System call(wait) 구현, Additional System call 구현

* 1. **개발 방법**

Argument Passing을 구현하기 위해 userprog/process.c에 push\_stack 함수를 추가하였다. load 함수에서 실행 파일명을 분리하고 push\_stack 함수를 호출한다. push\_stack 함수에서 입력 command에서 실행 파일명과 인자를 구분하여 esp를 이용해 스택에 쌓는다.

User Memory Access를 구현하기 위해 thread/vaddr.h의 is\_user\_vaddr 함수를 이용한다. stack에 쌓인 주소들을 이용해 system call을 호출할 때 주소들이 kernel address에 해당하는 지 확인한다. userprog/syscall.c의 syscall\_handler에서 스택에서 주소를 가져와 is\_user\_vaddr 함수로 주소가 user address인지 확인 후, kernel address일 경우 종료한다. User address에 해당할 경우 정상적인 접근이므로 해당 system call을 호출한다.

System calls를 구현하기 위해 lib/syscall-nr.h에서 구현하고자 하는 system call 번호를 확인하여 userprog/syscall.c의 syscall\_handler에서 이용한다. Userprog/syscall.c에는 syscall\_handler 외에 syscall을 구현할 함수를 추가한다. 편의상 userprogram이 호출하는 system call 이름 앞에 my\_를 붙여 정의했다. Syscall handler에서 syscall 번호에 맞게 추가한 함수들을 호출한다. 이 때 인자는 user stack에 쌓은 것들을 이용하기 위해 intr\_frame의 esp 포인터를 이용한다.

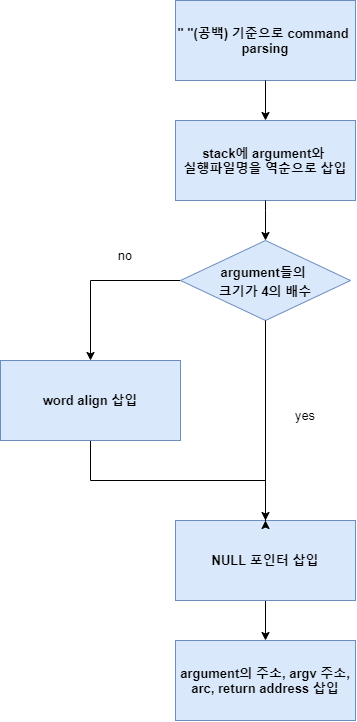
System call 중 wait을 구현하기 위해서는 user thread의 자식 thread들을 관리할 리스트 자료구조가 필요하다. threads/thread.h의 thread 구조체에 list\_child 리스트와 리스트의 원소 child\_elm을 추가한다. 또한 동기화 문제를 해결하기 위해 semaphore를 추가한다. Child가 종료될 때 까지 기다리기 위한 child\_lock과 execute\_lock을 추가한다. 이를 이용해 userprog/process.c의 process\_wait 함수와 process-exit 함수를 수정한다.

1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

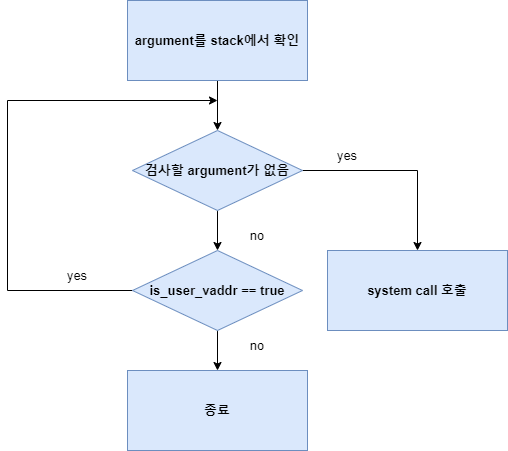
1. Argument Passing

userprog/process.c의 push\_stack 함수에서

****

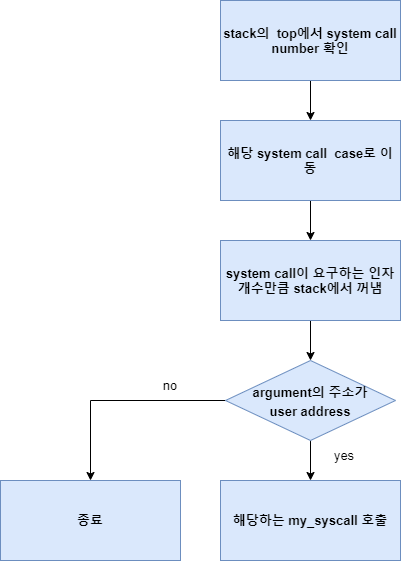
1. User Memory Access

userprog/syscall.c의 syscall\_handler에서



1. System Calls

userprog/syscall.c의 syscall\_handler에서



* 1. **제작 내용**

1. Argument Passing

Argument Passing 과정에서 인자를 parsing하고 stack에 쌓기 위해서 push\_stack 함수를 userprog/process.c에 추가하였다. 문자열 복사, 분리를 위해 lib/string.h의 strlcpy, strlcat 등을 이용하였다. load에 전달된 intr\_frame의 esp를 그대로 push\_stack에 전달하여 stack을 쌓는데 이용하였다.

1. User Memory Access

threads/vaddr.h의 is\_user\_vaddr 함수를 이용해 user address와 kernel address를 구분할 수 있다. 이를 userprog/syscall.c의 syscall\_handler에 적용시켜 system call을 호출하기 전에 유효한 접근인지 검사 후, 유효한 접근일 경우에만 system call을 호출할 수 있다.

1. System Calls

Halt, exit, exec, wait, read, write를 구현했는데 halt의 경우 shutdown\_power\_off를 호출하여 Pintos가 종료되도록 구현했고, exit는 현재 thread의 이름과 status를 출력하고 thread\_exit()를 호출하여 종료하도록 구현하였다. Exec는 process\_execute 함수를 호출하여 구현할 수 있었다. Read와 write의 경우 각각 STDIN과 STDOUT에 대해서 구현하면 되서 위에서 언급한 input\_getc 함수와 putbuf 함수를 이용해 구현할 수 있었다. Wait을 구현하기 위해 userprog/process.c의 process\_wait 함수를 구현해야 하는데 thread에 추가한 child thread들을 관리하기 위한 list를 반복문을 이용하여 하나씩 검사한다. process\_wait의 인자로 전달받은 tid와 child의 tid를 비교하여 기다릴 child를 찾아 리스트에서 제거한다. 이 때 동기화 문제를 해결하기 위해 wait\_lock과 execute\_lock을 이용한다. 위의 과정을 구현하기 위해 list 라이브러리의 list\_begin, list\_end 함수를 이용했고 list\_entry 매크로를 이용하여 리스트 원소를 감싸는 thread에 접근할 수 있었다.

1. Additional System calls

Fibonacci와 max\_of\_four\_int를 구현하기 위해 먼저 기존에 정의되어 있던 system call 외의 system call을 추가해야 하므로 lib/syscall-nr.h에 두 system call 번호를 등록하였다. 그리고 lib/user/syscall.c에 syscall4를 추가하여 인자가 4개인 max\_of\_four\_int를 구현할 수 있도록 했으며 이를 호출하는 fibonacci, max\_of\_four\_int system call도 등록하였다. userprog/syscall.c에는 syscall\_handler에 번호를 두개 추가하여 추가된 두 system call도 처리 가능하도록 했으며, my\_fibonacci와 my\_max\_of\_four\_int를 구현하여 실제 기능을 수행할 수 있도록 하였다.

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

  자동 생성된 설명

명세서의 예시를 수행한 결과이다. Pintos 상에서 원하는 결과가 출력됨을 알 수 있다.