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

담당 교수 : 박성용

**20151619 최준수**

개발 기간 : 2020/10/03~2020/10/25

1. **개발 목표**

현재 pintos code는 kernel 영역에서 process를 load하고 실행시킬 수 있는 기능은 구현되어 있지만 아직 user program을 OS 상에서 실행시킬 수는 없다. 따라서 user program이 OS 상에서 실행될 수 있도록 kernel 영역에 접근하기 위한 인터페이스(system call)를 구현하는 것이 이번 프로젝트의 주된 목표이다. 또한 User program이 실행될 때 사용되는 argument들을 User stack을 구성하여 전달하는(Argument passing) 또한 중요한 과제 중 하나이다. 이렇게 user stack을 구성하고 접근하는 과정에서 잘못된 접근을 하는 program들을 막아주는 예외처리 또한 필요하겠다.

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

Pintos를 실행할 때 전달받은 Argument 를 space bar를 기준으로 parsing한다. 첫번째 parsing word는 프로그램 이름이 될 것이며, 이와 같은 이름을 가진 파일을 load한다. load가 성공하면 user stack을 받아 80x86 Calling Convention에 맞추어 memory에 배치한다. 이를 통해 프로그램의 올바른 수행을 기대할 수 있다.

1. **User Memory Access**

User program은 user memory에만 접근해야 하며 어떤 상황에서 kernel 접근하는 것은 심각한 오류를 초래할 수 있다. 따라서 이를 방지하기 위해 우리가 접근하는 address가 kernel memory에 있는지 user memory에 있는지 알아야 한다. 따라서 user program이 kernel memory를 참조하게 된다면 해당 프로그램을 종료하고 종료 코드 1을 출력해야한다. 이를 구현하기 위해서는 threads/vaddr.h에 구현되어 있는 is\_user\_addr 함수를 통하여 해당 address가 유효한지 검사하는 함수를 작성하여야 한다.. 또한 page fault의 주체가 kernel일때도 종료 코드를 반환해야 하므로 이를 처리하기위해 userprog/exception.c의 page\_fault함수를 수정하여야 한다.

1. **System Calls**

user program에 대한 정보(program 이름과 argument)들이 argument passing을 통하여 user stack에 저장되어 있다면 syscall\_handler 함수가 이를 읽어 들어 system call 종류별로 분류한다. 구현해야 되는 system call은 기존 pintos 프로젝트에서 요구하는 halt, exit, exec, wait, read write 와 더불어 Fibonacci 수열의 값을 구하는 fibonacci, 네 정수의 합을 구하는 sum\_of\_four\_integers 이렇게 총 8가지로 각 system call 마다 함수를 추가한 다음 manual의 요구사항에 따라 userprog/syscall.c에 구현하면 된다.

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

80x86 Calling convention에 따라 User stack을 쌓아준다. Stack pointer(esp)에서부터 시작하여 들어온 argument들을 뒤에서부터 순서대로 삽입하여준다. Argument의 바이트 크기만큼 esp를 차감하며 진행한다. argument들을 넣을 때 총 몇바이트를 넣는지를 저장하여두어야 한다. argument들의 값이 잘 저장이 되었다면, WORD align을 해주어야한다. Pintos는 1Word가 4바이트이므로 4단위로 떨어지도록 esp를 추가로 차감하여준다. 그리고 80x86 Calling convention에 따라 NULL을 삽입한다. 다음은 앞서 저장한 argument들이 저장된 주소를 저장해준다. Pintos는 32비트이므로 주소의 크기 또한 4바이트이고 따라서 주소를 저장하고 4바이트씩 차감하면 된다. 그리고 argument들이 저장된 주소를 저장하기 위해 현재의 esp의 주소를 저장한다. 그 다음 argument의 개수를 나타내는 argc를 저장하고, 종료후 돌아갈 주소인 return address를 저장하는데 0으로 저장한다.

정리하자면:

1. Argument들 저장
2. Word-alignment
3. NULL
4. Argument들 주소
5. Argument 시작 주소
6. Argument count 저장
7. Return Address (0)

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

Kernel은 User space의 메모리에 접근할 수 없고, User Program은 Kernel space의 메모리에 접근할 수 없다

* + **Invalid memory access를 어떻게 막을 것인지 설명**

exception.c의 page\_fault()를 보면 접근하는 process가 user인지 아니면 kernel인지를 알 수 있다. 해당 변수와 threads/vaddr.h에 선언된 is\_user\_vaddr()이나 is\_kernel\_vaddr()을 사용하면 kernel이 user space에 접근하려고 하는지, user program이 kernel space에 접근하는지 판단할 수 있을 것 같다.

* **System Calls**
  + **시스템 콜의 필요성에 대한 간략한 설명**

User program이 Kernel space에 접근하려고 하면 앞서 언급한 Invalid memory access가 일어나게 된다. 하지만 User program에서도 Kernel에서 제공하는 기능들을 사용해야되는 경우가 종종 있다. 가장 쉽게 떠올릴 수 있는 echo, printf 처럼 콘솔창에 내용을 출력하는 것 또한 device driver를 다루고있는 Kernel에서 제공하는 기능을 사용하여야한다. 이 때 User program이 직접 kernel space에 접근할 수 없기 때문에 system call을 사용하여 kernel에게 요청하게 된다.

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

halt : devices/shutdown.h에 선언되어 있는 shutdown\_power\_off 함수를 호출함으로써 pintos를 종료시킨다.

exec : 해당 command line을 가지고 process.c/process\_execute를 실행한다. exec은 만든 process가 load가 될 때까지 동기화 시키고 load가 성공적으로 수행되면 실행시킨 process의 id를, 실패하면 -1을 반환한다.

wait : 주어진 tid를 가진 process가 종료되었는지 process.c/process\_wait을 사용하여 계속 확인하며, 만약 종료되었다면 프로세스의 종료코드를 반환하고 종료되지 않았다면 semaphore를 통해 기다린다.

syscall\_read : fd가 0일 때 input\_getc를 이용해서 키보드로부터 받은 입력을 buffer에 저장한다. 그리고 실제로 입력 받은 byte 수를 return한다.

write : fd가 1일 때 putbuf를 호출하여 buffer에 저장된 데이터를 출력하고 출력한 byte 수를 return한다.

exit : 종료하려는 thread의 이름과 상태를 출력한 후 thread\_exit()을 사용하여 thread를 종료시킨다. 인자로 넘겨받은 status는 thread의 exit\_status에 저장해둔다.

fibonacci : input n에 대한 fibonacci 수를 구하여 return한다. 구현 방법은 dynamic programming, recursion 등 다양하다.

max\_of\_four\_int : 4개의 정수가 input으로 들어오면 그 중 가장 큰 정수를 반환한다.

* + **유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명**

우선, user program에서 호출하는 system call API는 lib/user/syscall.h와 lib/user/syscall.c에 선언되어있다. 그리고 lib/user/syscall.c에서 사용하는 system call number enum들은 lib/syscall-nr.h에 선언되어있다. user program에서 system call API를 호출하면 lib/user/syscall.c의 함수가 호출이 되는 것이고 이 함수들은 predefine된 syscall0(), syscall1() 등 자신의 argument 개수에 맞는 것을 호출하여 stack에 argument들과 system call number를 넣어준다.

그런 후, threads/intr-stubs.S의 assembly code에서 call intr\_handler를 호출하면 threads/interrupt.c의 void intr\_handler(struct intr\_frame \*frame)이 호출된다. intr\_handler는 frame에 저장된 frame->vec\_no를 통해 어떤 interrupt가 발생했는지를 확인하고 interrupt에 맞는 handler를 호출해준다. userprog/syscall.c의 syscall\_init()을 보면 system call은 0x30 interrupt number로 등록하고 있으며, syscall\_handler가 처리하도록 되어있다.

intr\_handler()가 0x30번 interrupt에 대해서는 등록되어있는 handler인 syscall\_handler를 호출할 것이고, syscall\_handler는 stack에 저장된 system call number를 추출하여 각각의 system call에 맞는 작업을 수행한다. 만약 user program에게 반환하여야되는 값이 있다면 struct int\_frame \*f의 eax에 저장을 하면 user program이 받아서 사용한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

10/3 ~ 10/7 프로젝트 분석, 요구사항 및 manual 숙지

10/7~10/10 Argument Passing

10/10~10/12 User Memory Access

10/12~10/21 System call

10/21 ~ 10/25 오류 수정 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)**

**Argument Passing**userprog/process.c 파일에서 command line으로 넘어온 문자열을 strtok\_r()을 사용하여 스페이스바 기준으로 분리하고 첫번째 단어가 프로그램의 이름으로 이 이름을 가진 프로그램을 load()한다. load()가 끝나면 setup\_esp함수를 통해서 user stack의 주소를 전달받고 이 주소에서 시작하여 80x86 Calling convention에 까라 user stack을 쌓아주면된다. 이를 쌓는 순서는 위에서 언급한 것처럼 다음과 같다:

* Argument들 저장
* Word-alignment
* NULL
* Argument들 주소
* Argument 시작 주소
* Argument count 저장
* Return Address (0)

**User Memory Access**

userprog/syscall.c에서 user program이 kernel memory의 참조를 막기 위하여 syscall 호출시 쌓여지는 스택을 검사하는 함수 check\_addr()를 만들 것이다. 해당 stack의 끝 주소를 is\_user\_vaddr 함수로 검사하여 kernel memory인지 판단할 것이다. 이 외에도 kernel에 의한 page fault를 처리하여야 하므로 userprog/exception.c의 page\_fault 함수를 수정할 것이다.

**System Call**userprog/syscall.c안의 syscall\_init()함수가 0x30번 interrupt를 system call로 등록하고 있다. 그리고 0x30번 interrupt가 발생하면 바로 아래에 선언된 syscall\_handler()를 호출하도록 인자로 넘겨주고 있기 때문에 system call은 syscall\_handler에 구현하면 된다. syscall\_hander의 인자인 struct intr\_frame \*f의 esp에서 system call number와 인자들을 읽어 switch문으로 나누어 각각에 맞는 작업을 수행하면된다. 아래는 system call별로 구현해야되는 함수들이다.

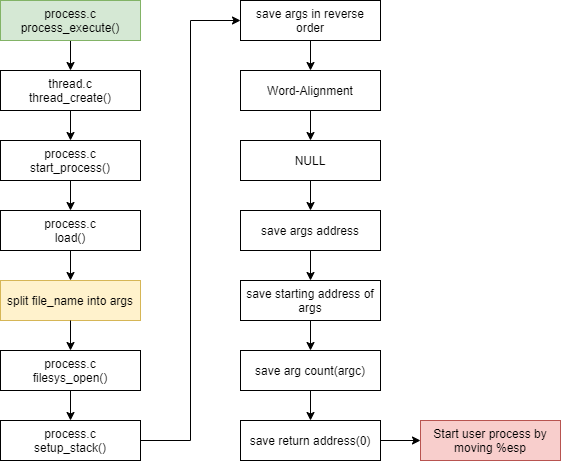
* halt : devices/shutdown.h에 선언되어 있는 shutdown\_power\_off 함수를 호출함으로써 pintos를 종료시킨다.
* exec : 해당 command line을 가지고 process.c/process\_execute를 실행한다. exec은 만든 process가 load가 될 때까지 동기화 시키고 load가 성공적으로 수행되면 실행시킨 process의 id를, 실패하면 -1을 반환한다.
* wait : 주어진 tid를 가진 process가 종료되었는지 process.c/process\_wait을 사용하여 계속 확인하며, 만약 종료되었다면 프로세스의 종료코드를 반환하고 종료되지 않았다면 semaphore를 통해 기다린다.
* syscall\_read : fd가 0일 때 input\_getc를 이용해서 키보드로부터 받은 입력을 buffer에 저장한다. 그리고 실제로 입력 받은 byte 수를 return한다.
* write : fd가 1일 때 putbuf를 호출하여 buffer에 저장된 데이터를 출력하고 출력한 byte 수를 return한다.
* exit : 종료하려는 thread의 이름과 상태를 출력한 후 thread\_exit()을 사용하여 thread를 종료시킨다. 인자로 넘겨받은 status는 thread의 exit\_status에 저장해둔다.
* fibonacci : input n에 대한 fibonacci 수를 구하여 return한다. 구현 방법은 dynamic programming, recursion 등 다양하다. custom system call이기 때문에 lib/syscall-nr.h에 SYS\_FIBONACCI를 추가한다
* max\_of\_four\_int : 4개의 정수가 input으로 들어오면 그 중 가장 큰 정수를 반환한다. custom system call이기 때문에 lib/syscall-nr.h에 SYS\_MAX\_OF\_FOUR\_INT를 추가한다.

이 system call 함수들을 처리하는 함수들을 userprog/syscall.c에 구현하고, parameter 가 4개인 max\_of\_four\_int을 위하여 lib/user/syscall.c에 4개를 스택에 쌓는 assembly code를 추가할 것이다.

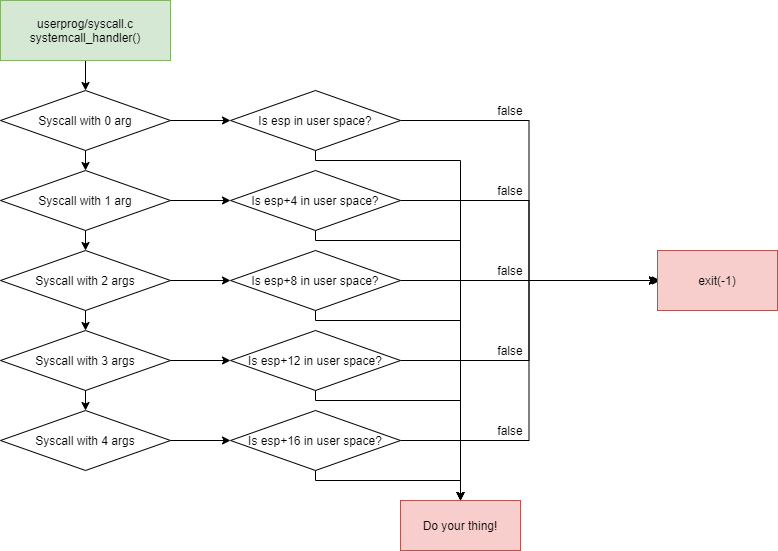
1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

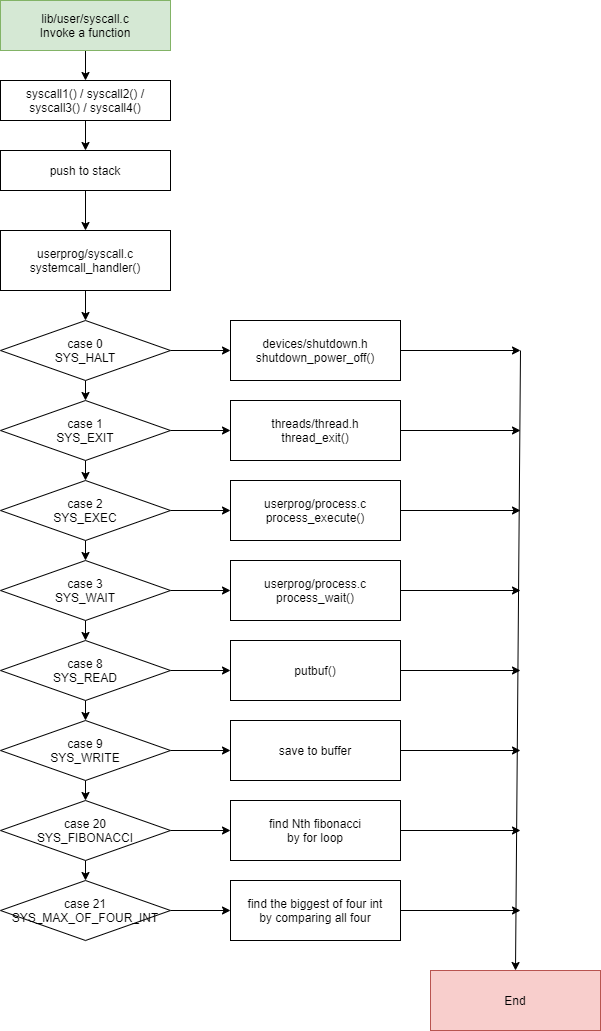
1. Argument Passing



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

userprog/process.c

전달받는 command는 const 데이터 형이기 때문에, parsing하기 전 같은 내용을 가진 새로운 문자열 배열을 생성한다. 이 문자열 배열에서 space bar를 기준으로 parsing을 진행하고 이때 생성된 첫번째 parsing 문자열을 가지고 load를 진행한다. load가 완료된다면 stack을 받아 argument passing함수를 호출한다. 이때 함수는 parsing 정보, ESP, argument count, argument length를 받는다. argument passing은 80x86 Calling Convention 기준에 의해 stack에 정해진 순서대로 쌓여야 하는데 이때 parsing 된 문자열이 역순으로 저장된다. 따라서 편의를 위하여 argv라는 문자열 배열에 parsing된 문자열들을 역순으로 저장한뒤 차례대로 argument passing을 진행하도록 하였다. command passing이 완료되면 word\_align을 위해 빈 공간을 추가하여야 한다. 이 때 word\_aling을 위한 공간 size는 다음과 같이 구현하였다. (WORD\_SIZE – (ARG\_LEN) % WORD\_SIZE) % WORD\_SIZE. command가 passing된 곳의 주소를 쌓고, argument count, return address를 차례대로 쌓고 나면 모든 argument passing과정이 성공적으로 종료된다.

1. User Memory Access

User Memory Access에서 kernel memory에 접근하지 않기 위한 방법으로서 두가지를 구현하였다.

userprog/syscall.c

첫번째로는 syscall 호출시, interupt의 esp가 kernel memory를 참조하는가?

이는 check\_addr() 이라는 함수를 만들어 kernel memory를 참조한다면 -1를 종료코드로 하는 syscall exit을 호출하여 쓰레드를 종료하도록 구현하였다.

userprog/exception.c

두번째로는 kernel이 user memory를 접근하는가?

이때는 page fault시에 접근 주체가 kernel이고 접근 주소가 user memory일 경우 탐지가 되며, -1를 종료코드로 하는 syscall exit을 호출하여 쓰레드를 종료하도록 구현하였다.

1. System Calls

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

**halt**  
userprg/syscall.c devices/shutdown.c  
shutdown.c에 구현되어있는 shutdown\_power\_off를 호출함으로써 실행되고 있는 process는 물론 virtual machine까지 종료하도록 구현하였다.

**wait**userprog/syscall.c userprog/process.c  
syscall.c에서 wait는 단순하게 userprog/process.c의 process\_wait()를 호출해준다. process.c의 process\_wait()는 semaphore를 사용하여 thread의 종료를 기다리게되는데, thread가 처음 생성될 때 child라는 리스트를 가지고 시작한다. 각 thread가 새로운 thread를 생성할 때마다 이 child라는 리스트에 추가된다. process\_wait()에서는 인자로 넘어옴 tid\_t child\_tid와 같은 tid를 가진 thread를 child 리스트에서 찾는다. 찾은 후에는 sema\_down으로 기다리다가 종료되어 child가 sema\_up을 호출하면 list에서 제거하고 exit status를 반환한다.

**exec**  
userprog/syscall.c userprog/process.c  
exec은 스택으로 넘어온 파일명을 실행시켜주는 함수로서 userprog/process.c의 process\_execute을 실행한다. process\_execute은 인자로 넘어온 file\_name이라는 프로그램을 thread\_create()를 사용하여 실행시키고, 새로 생성한 thread의 id인 tid를 반환한다.

**read**  
userprog/syscall.c  
syscall\_read 함수에 들어오는 parameter는 fd, buffer, size 이렇게 세 가지가 들어오는데 우선 fd가 0 일 때가 standard input을 받는 경우이므로 fd 가 0인 경우에 한해서 input\_getc 함수를 통해 한 글자씩 입력 받아 buffer에 저장하도록 구현하였다. 그리고 마지막은 입력받은 byte 수인 size를 return하도록 하였다.

**write**  
userprog/syscall.c  
syscall\_read 함수와 마찬가지로 parameter에 fd, buffer, size가 들어오는데 콘솔 출력을 할 경우에 fd는 1이므로 fd가 1인 경우에 한해서 buffer 저장된 데이터를 모두 출력해주는 putbuf 함수를 호출하도록 구현하였다. 그리고 출력한 byte 수를 나타내는 size를 return하도록 하였다.

**exit**  
userprog/syscall.c threads/thread.c  
종료하려는 thread의 이름과 상태를 출력하기 위해 thread\_name 함수를 통해 이름을 저장한다. Eixt 문구를 출력한 후에는 thread\_exit\_status 함수를 호출해 thread가 종료되도록 구현하였다. 성공적으로 종료가 되면 status값은 0이 되며 error가 발생할 경우 status는 0이 아닌 값이 된다.

1. Additional System calls

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

**공통:**

userprog/syscall.c lib/syscall-nr.h lib/user/syscall.c

새로운 system call인 fibonacci와 max\_of\_four\_int를 구현하기위해서는 일단 두 system call에 새로운 system call number를 부여해줘야되므로 lib/syscall-nr.h에 선언된 enum의 끝에 SYS\_FIBONACCI와 SYS\_MAX\_OF\_FOUR\_INT를 추가했다. 그리고 user program에서 접근할 수 있도록 lib/user/syscall.c와 lib/user/syscall.h에 추가하였으며, max\_of\_four\_int는 인자를 4개를 받기 때문에 syscall4를 새로 define해 주었다.

**fibonacci**

userprog/syscall.c에 실제로 피보나치 수열을 계산하는 함수를 추가했다. 반환하고자하는 값인 result와 이전 값을 가지고 있는 last를 서로 더해주고 업데이트해주며 우리가 흔히 아는 fibonacci수열을 반복문을 사용하여 구해주었다. 반환하는 계산값은 다른 system call과 같이 f->eax에 저장해주었다.

**max\_of\_four\_int**

userprog/sycall.c에 4개의 정수 중 가장 큰 값을 찾는 함수를 추가하였다.

상단에 #define GET\_MAX(a,b) (((a)>(b))?(a):(b))를 선언하여 두 정수를 비교하여 더 큰 정수를 반환하도록 하였고 a,b,c,d에 대해 순차적으로 호출하여 가장 큰 값을 f->eax에 저장해주었다.

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

