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

담당 교수 : 김영재  
학생 : 20161614 원성현

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

이번 프로젝트의 목표는 3가지 항목을 구현하는 것이다.  
Argument Passing, User Memory Access, System Call이다. 각 항목(기능)은 유기적으로 같이 작동하여 User-Kernel을 연결하며 User가 원하는 작업을 OS에서 실행하게 만든다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

Argument Passing을 정상적으로 구현했다면, 우리는 Stack을 통해 User에서 Kernel(OS) 쪽에 우리가 원하는 명령어를 전달(Pass)해줄 수 있다.   
지금까지는 User-level에서 Passing을 할 때 function call을 통해 인자를 전달했다면, User-Kernel 간의 통신에서는 Stack이 기반이 된다.

1. User Memory Access

User Memory Access를 정상적으로 구현했다면, Invalid한 pointer들을 reject하게 된다. 이것은 kernel이나 다른 프로세스에 피해를 주지 않기 위함이다.

1. System Calls

System Call을 정상적으로 구현했다면, User Memory 영역에서 Kernel Memory 영역으로 System Call을 통해 Kernel Mode에서 User가 원하는 작업을 OS가 실행하게 한다.

* 1. **개발 내용**
* **아래 항목의 내용만 서술 (기타 내용은 서술하지 않아도 됨.)**
* Argument Passing
  + 커널 내 스택에 argument를 쌓는 과정 설명  
    스택에는 Command에 입력된 정보들이 차례대로 쌓이게 된다.   
    PPT와 Document에 서술된 대로, 반환주소, 인자의 개수, Command를 파싱한 문자열들의 주소, word-alignment 값, 문자열이 차례대로 쌓인다
* User Memory Access
  + Pintos 상에서의 invalid memory access 개념을 간략히 설명  
    Invalid Memory에 Access한다는 것은, pre-reserved된 메모리 영역에 접근해서 문제를 일으키는 것이다. 대표적인 invalid memory에는 다른 process가 이미 실행 중인 영역의 memory, 그리고 kernel이 사용하고 있는 영역의 memory가 있다.
  + Invalid memory access를 어떻게 막을 것인지 설명  
    Invalid memory임을 확인할 수 있는 함수가 고맙게도 vaddr.h에 제공된다.  
    이 함수를 파악해서 리턴값을 확인해서 invalid한 memory를 확인할 수 있다.
* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명  
    시스템 콜은 user 영역(프로그램)에서 요청한 작업을 커널의 기능을 이용해 처리하는 것이다. 예를 들어, user가 파일을 열고 닫고 싶을 때, system call을 요청하면 OS가 system call을 해석해서 user의 요청을 들어준다.
  + 이번 프로젝트에서 개발할 시스템 콜에 대한 간략한 설명 (하나의 시스템 콜 당 최대 3문장으로 간략히 설명; 3문장을 넘길 정도로 길게 작성하지 말 것)  
    halt : pintOS를 종료한다.  
    exit : 현재 프로세스를 종료한다.  
    exec : 특정 명령어를 실행한다.  
    read(stdin) : stdin에서 그 정보를 읽어온다.  
    write(stdout) : 정보를 stdout에 출력한다.  
    Fibonacci : 어떤 수를 인자로 받아, 피보나치 값을 계산한다.  
    max\_of\_four\_int : 네 수를 인자로 받아, 최대값을 계산한다.
  + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명  
    User이 system call API를 호출하면, 인자의 개수에 따라 syscallN(cmd)를 호출한다. 이 syscall\_N은 intr\_handler에 의해 syscall\_handler을 호출하게 된다.   
    syscall\_handler에서는 user에서 kernel로 전달해준 struct intr\_frame 구조체(스택)을 이용해서 user가 원하는 정보를 받아서 파악한다. 그 정보를 가지고 kernel API가 호출되서 유저가 원하는 작업을 한다. 작업이 끝난 뒤에는 ret\_from\_sys\_call 함수를 호출해서 유저 영역으로 돌아온다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

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

1. Argument Passing : ~ 9.10.  
2. User Memory Access : 9.10 ~ 9.15.  
3. System Call : 9.15 ~ 10.2.

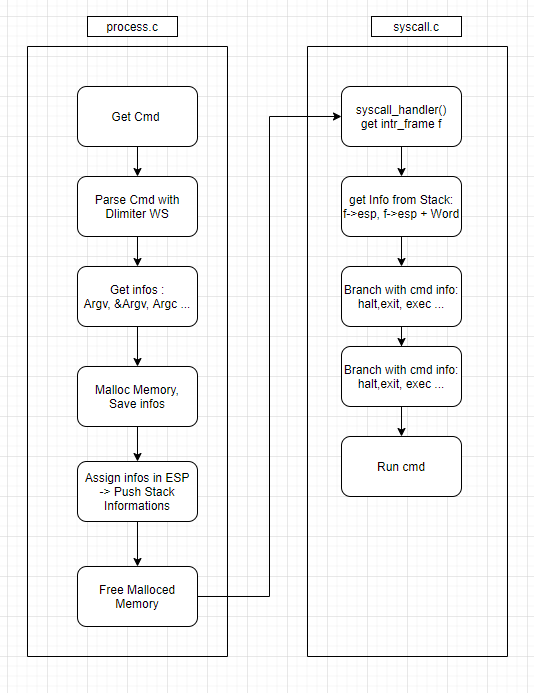
* 1. **개발 방법**
* **II. B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)**

1. Argument Passing   
-> Argument Passing을 위해서, 우선적으로 Parsing을 해야 하므로 명령어를 띄어쓰기 단위로 Parsing을 하는 함수를 작성하고, 이것들을 저장할 char\* 의 배열을 만들어야 할 것이다.   
또한, 배열들에 반드시 null이 붙어야 하므로, strcpy 함수가 아니라 strlcpy함수를 사용해야 할 것이다. 그리고 word-alignment, return address를 담기 위한 다양한 형식의 변수를 추가해야 할 것이다. 이러한 내용들을 esp에 assign 함으로서 argument passing이 가능할 것이다. 이것들은 process.c에서 이루어진다.  
syscall.c에서는 frame인 f를 이용해서 esp에서 원하는 정보를 받아와서, 명령어에 따라 syscall을 호출할 것이다.  
2. User Memory Access  
-> User Memory access는 vaddr.h에 함수를 제공하는 것으로 확인되었다.  
따라서 우리는 invalid한지 check하는 함수인 check\_address함수를 만들어서, 주어진 is\_user\_vaddr(addr)의 리턴값을 잘 이용하면 될 것이다.  
system call을 할 때마다, 메모리값에 check\_addr 함수를 호출에 안전한 메모리인지 확인하면 될 것이다.  
3. System Call  
-> System Call에서는 명세서에서 요구한 system call인 exit, halt, exec, wait, read, write, Fibonacci, max\_of\_four\_int 함수를 syscall.c, syscall.h 에 추가하고, os와 userprog 간의 소통을 위해 lib/user의 syscall.c, syscall.h에 새로 추가하는 Fibo, max\_of\_four\_int를 추가한다. Syscall-nr.h에도 enum형에 추가해줘야한다.  
마지막으로 Synchronization 문제를 해결하기 위해서 thread.h, thread.c에서 child에 관한 정보를 추가하고, process.c에서 semaphore를 이용해 겹치지 않게 구현한다.

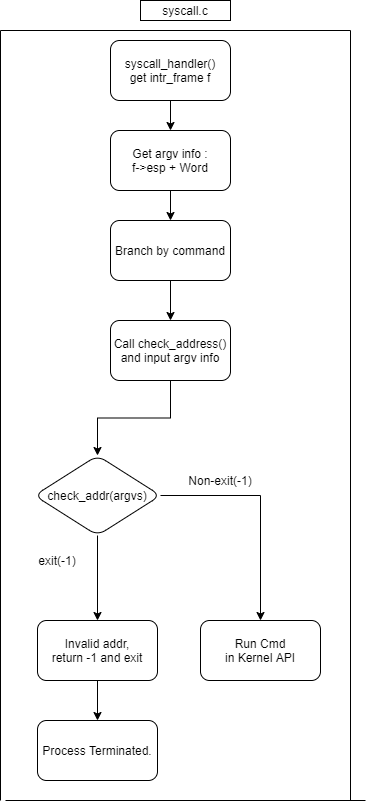
1. **연구 결과**
   1. **Flow Chart**

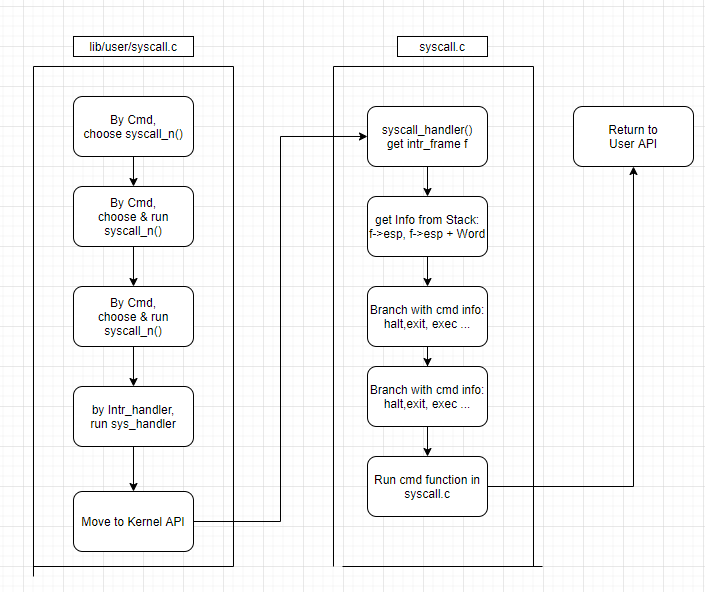
* **II. B. 개발 내용에 대한 Flow Chart를 작성**

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

Argument Passing은 주로 process.c에서 이루어진다. Load()함수에서 filesys\_open을 위해 우선 사용자가 입력한 command를 parse하는 작업이 필요하다.  
따라서 char\*의 array를 만든다음, strtok\_r함수를 이용해 white space의 delimiter 기준으로 command를 쪼개어줌과 동시에, chunk가 몇 개 나오는지 확인한다. 그 이유는 이렇게 쪼갠 것들과 인자의 개수 등의 정보를 stack에 넣어야 하기 때문이다.   
쪼갠 다음에, setup\_stack(esp)가 호출되고, 본격적으로 스택을 쌓는 과정이 시작된다.  
앞서 보고서 3번에서 이야기했던 것 처럼, argc를 담기 위한 int 변수와, 인자의 정보를 담을 수 있는 char\*\* 변수, 리턴주소를 담을 void\* 형 변수 등을 선언하고, 메모리를 allocate해준다.   
그리고 3번에서 이야기한 순서대로 스택을 쌓는데, 이것은 \*esp를 decrement하는 방식을 통해 구현된다. 주의할 점은 각 인자들이 스택에 assign될 때, 반드시 NULL이 뒤에 있어야 하므로 strlcpy함수를 사용한다. Word Alignment의 값은 Word Alignment를 쌓기 직전, 스택이 쌓인 길이를 확인해서 4로 나누어 봄으로서 계산할 수 있다.   
이렇게 스택을 쌓은 뒤에, allocated memory를 free해주면 스택을 쌓는 과정이 종료된다.   
이제 syscall.c에서 스택의 정보를 가져오는 과정에 대해 이야기해보자.  
syscall.c에서는 intr\_frame\* f변수를 통해 정보를 가져온다. 앞서 file\_sys의 인자, 즉 명령의 종류에 따라 syscall\_handler에서 f->esp를 확인한다. F->esp에는 pass한 스택에 관한 정보가 담겨있다. 명령어가 EXEC였다면, 추가 인자가 1개이므로, f->esp에는 명령어의 number, f->esp+4에는 스택의 정보가 담겨 있는 식이다.   
이렇게 f->esp를 이용해 argument passing이 완료된다.

1. User Memory Access

Argument Passing이 완료되었다고 바로 write()나 read()의 function을 syscall\_handler에서 실행하면 안된다. 그 이유는 stack에서 넘겨받은 정보, 즉 위에서 f->esp를 이용해 접근할 수 있는 포인터가 invalid할 수 있기 때문이다.   
이러한 포인터의 invalidity에 대해서는 앞에서 설명했으므로, check\_address(addr)함수가 어떻게 invalidity를 검증하는지 적겠다. Max\_of\_four\_int처럼 인자가 4개인 함수인 경우에는, 총 4가지 포인터값(f->esp, f->esp + 4/8/12/16)을 check\_addr()의 인자로 넘겨주는데, check\_addr함수는 단순하다. Vaddr.h에 포함된 is\_user\_vaddr(addr)값이 false라면 exit(-1)을 호출한다. 즉, user\_vaddr가 true일 때, 다른 kernel memory가 아닐 때 void를 리턴하고, 문제가 있는 경우에는 error code -1을 exit()에 전달한다.  
이렇게 f->esp + a 를 계속 check\_address 해줌으로서 user memory access를 구현한다.

1. System Calls

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

우선 src/userprog/syscall.c 에서 많은 코드들을 추가해주어야 한다.  
가장 먼저, f->esp를 받아와 그것을 uint32\_t로 형변환하면 우리는 스택의 가장 최초값에서 명령어를 알 수 있다. 여기서 명령어는 정수 enum형이고 syscall.h에 정의되어 있다. 따라서 우리는 switch문을 이용, 명령어 인자의 개수에 따라 esp, esp+4, esp+8, esp+12의 값을 이용해 원하는 명령어를 수행한다.  
여기서 명령어에 해당하는 함수를 syscall.c,h에 정의하고, 구현해주어야 한다.  
  
halt는 shutdown\_power\_off를 단순하게 호출하면 된다.  
exit은 현재 쓰레드의 이름을 thread.h에서 제공하는 함수인 thread\_name()를 호출해 알아내고, status를 출력한다. 마지막으로, 현재 쓰레드의 종료 status를 인자로 받은 status로 바꾸어주고 쓰레드를 종료한다.  
exec는 단순하게 process\_execute를, wait는 단순하게 process\_wait를 호출하면 된다.  
check\_addr는 위에서 설명 하였다. Read는 stdin에서만 진행되기 때문에, fd가 0일 때에만 buffer로부터 input\_getc()를 NULL을 만나기 전까지 하고, 읽어들인 size를 리턴해준다. Write는 fd가 1일때에 putbut(buffer,size)함수를 호출하면 된다. 현재 프로젝트에서는 stdout에 size만큼 buffer의 내용을 출력한다.  
마지막으로 리턴값이 존재하는 함수는 f->eax에 그 값을 대입해준다.  
  
이렇게 함수의 구현이 끝난 뒤에는 synchronization 문제를 해결해야 한다.  
process간의 wait를 제대로 구현해야 하기 때문이다. 이 말인 즉슨, child process가 완전히 종료되었는지, 어떤 exit status로 종료되었는지 등의 정보가 필요하다.  
따라서 threads/thread.h에서 exit\_status, child의 process, memory를 제어하는 세마포어, 그리고 struct list와 list\_elem의 자료형을 추가해준다.  
thread.c에서는 쓰레드가 만들어 질 때, child의 memory와 thread 세마포어를 sema\_init()을 이용해 초기화하고, list형의 thread->child역시 초기화한다. 마지막으로 list에 child\_thread\_elem을 추가한다.  
  
이제 마지막으로 process.c에 돌아가서 process\_wait, process\_exit을 수정해야 한다.  
Process\_wait에서는 현재 thread의 child\_thread의 list를 돌면서 인자의 child\_tid와 일치하는 thread가 있는지 검사한다. 있다면 sema\_down을 해서 Parent Process의 종료를 막는다. 자식 process의 작업이 다 끝난 다음에야 부모 process가 종료할 수 있으므로, exit\_status에 현재 thread의 자식의 exit\_status를 삽입하고 부모 thread를 list에서 제거한다. 그리고 sema\_up을 한다.  
Process\_exit에서는 sema\_up과 sema\_down을 해서 앞의 lock들을 풀어준다. 이렇게 synchronization 과정이 전부 끝난다.

1. Additional System calls

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

User-defined system call을 구현하기 위해서는 손봐야 하는 코드들이 몇 개 있다.  
첫째로, syscall.c에서 Fibonacci(int n)과 max\_of\_four\_int를 구현해야 한다.  
Fibonacci()는 recursive하지 않게 구현했다. 재귀는 스택이 계속 쌓이는 형식이고, 현재 민감한 system을 다루는 코드에서는 반복 스택을 쌓는 것은 위험하다고 판단했기 때문이다. Max\_of\_four\_int는 배열을 이용해서 원소의 최대값을 찾는 것으로 구현했다.  
syscall\_handler의 switch문에서 case\_fibo, case\_max\_of\_cour\_int를 추가해준다.  
주의할 점은, 이러한 enum형에 새로운 시스템 콜에 해당되는 자료는 없기 때문에,   
우리는 src/lib/syscall-nr.h에 가서 2가지 syscall에 대한 enum형을 추가해주도록 한다.  
그 뒤에 case에서 addr validity를 체크하고, 똑같이 실행해주면 된다. 두 함수 모두 리턴값이 존재하므로, f->eax에 그 값을 넣어준다.  
syscall.h에 syscall.c에 작성한 함수들의 원형을 추가해주면 된다.  
  
둘째로, 이제 lib/user/syscall.h와 syscall.c를 수정해야 한다. 우선 syscall.h에서 앞선 문단에서 했던 것 처럼 두 함수를 추가해준다. 그리고 나서 syscall.c를 수정하는데, 현재까지 우리는 syscall3까지, 즉 추가인자가 3개인 경우에만 define 되어있으므로 max\_of\_four\_int를 위해 syscall4를 추가해준다. Fibonacci의 경우 인자가 1개이기 때문에 필요가 없다. 그 뒤 2개의 함수를 추가해준다. Fibonacci는 syscall1을 호출하고, max\_of\_four\_int는 새로 추가한 syscall4를 호출하는 방식이다.  
  
마지막으로 src/examples에 들어가 테스트 코드 additional.c를 작성한다.  
내용은 간단하다, char\*\* argv로부터 정보를 받아서, 제대로 인자가 passing 되었는지 확인한 후에, fibonacci와 max\_of\_four\_int를 function call하고 return값을 저장한다.  
마지막으로 내용을 콘솔에 출력하면 끝이다.

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