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

담당 교수 : 김영재 교수님

이름 : 곽우철

개발 기간 : ~10/2

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

user program을 실행시키는 것으로 입력한 argument를 stack에 pass해야 하는데 이를 순서에 맞게 구현한다. 또, user program에서 system call을 호출했을 때 동작하는 system call handler와 각각의 system call 함수들을 만든다. 또한, user pointer가 kernel 영역을 침범하지 않도록 memory access도 관리해준다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

setup\_stack()으로 제공되는 스택에 arguments들을 passing하는 것인데 맨 끝의 argument 문자열부터 앞의 것까지 차례대로 push되고 alignment 밑 null 삽입 후 각 argument들의 주소가 그 다음으로 push되고(마찬가지로 끝의 것부터) agrv의 주소, argument 개수인 argc, return address등도 마지막에 push됨으로써 스택이 완성된다.

1. User Memory Access

user address가 kernel 영역을 침범했을 때 exit을 통해 종료해준다.

1. System Calls

user program에서 system call 호출 시 system call handler를 통해 stack에 접근하여(interrupt handler의 esp로) 구현해준 각 system call function들이 kernel API를 이용하여 알맞은 역할을 해준다.

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

stack의 esp가 있으면 위에서 말한 것과 같이 먼저 끝의 문자열부터 차례대로 들어가는데 각각의 길이만큼(NULL 포함) esp를 내려주고 그 곳에 넣어준다. 맨 앞의 문자열까지 넣어주면 word alignment 과정을 거쳐야 하는데 WORD의 단위가 4이므로 앞의 문자열들의 길이의 합이 4의 배수에 맞는지 확인하고 부족하면 그만큼 esp를 내려주고 스택을 채운다. 그 다음으로는 각 argument들의 주소가 들어가는데 규칙상 먼저 NULL이 들어가야 한다. 주소들을 넣어줄 때에는 앞과 달리 esp를 일정하게 4만 내리면 된다(일정하므로). 그리고 argv 주소 즉, 바로 위에 argv[0]을 넣었으므로 그것의 주소를 넣어주고 agrv의 개수인 argc, return address도 각각 esp 4만큼 내려서 넣어주면 된다.

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

system call 호출 시 user pointer가 kernel space에 접근할 수도 있기에 이는 invalid한 memory access이다. 또한, NULL 포인터도 invalid한 것에 포함하므로 처리해줘야 한다. 마지막으로 user virtual memory가 있으면 mapped되지 않은 부분이 있는데 이곳에 침범하는 것도 invalid memory access라 할 수 있다.

* + Invalid memory access를 어떻게 막을 것인지 설명

"threads/vaddr.h"에서 제공해주는 is\_user\_vaddr()를 통해 해당 주소가 user space인지 확인 가능하고 마찬가지로 is\_kernel\_vaddr()를 통해 kerenel space인지 확인 가능하다. unmapped memory의 경우는 userprog/pagedir.c의 pagedir\_get\_page() 함수를 통해 확인한다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명

안전성을 위해 user memory, kernel memory를 나누어 놓는데 user mode의 경우에는 kernel의 유용한 기능들을 사용하지 못하기 때문에 OS에서 제공하는 system call을 통해 kernel mode에 접근한다.

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

halt의 경우는 "devices/shutdown.h"에서 제공하는 shutdown\_power\_off(); 함수를 통해 pintos가 terminate되도록 하였다.

exit의 경우는 현재 thread의 status를 출력하고 thread의 요소인 exit\_status에 status를 저장 후 thread\_exit()을 통해 종료된다. (정확히는 process\_exit()이 또 호출된다)

exec의 경우는 process\_execute를 호출해서 전달받은 file\_name을 통해 새로운 thread를 생성하고(이 과정에서 file\_name parsing이 한 번 더 필요하다) 이것의 id를 return 값으로 주고 system call의 output으로 한다.

wait의 경우도 process\_wait를 호출하는데 child이 죽기전까지 parent도 죽으면 안되고 기다려야 하므로 semaphore를 이용한다. 여기서 그 child의 exit\_status를 system call의 output으로 한다.

read, write의 경우 stdin, stdout만 고려하면 되므로 각각의 fd값이 0, 1인지 확인 후에 input\_getc(), putbuf()를 통해 입출력을 한다. output 값은 입출력한 문자의 길이로 한다.

additional로 구현한 fibonacci의 경우 n번째 피보나치 숫자를 구하면 되므로 p3=p1+p2에서 p1, p2, p3 숫자를 옮겨가면서 지정하고 count가 n이 되면 p3을 output으로 하였다..

max\_of\_four\_int는 a, b, c, d를 비교해서 max를 return 값을 하고 이를 system call의 output으로 하였다.

* + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명4

user에서 system call 호출 시 system call number와 arguments들은 stack에 쌓이고 interrupt handle이 호출된다. 이 interrupt handler가 다시 system call handler를 호출하고 여기서 stack에 접근하여 알맞은 함수를 접근해서 처리해준다. 그리고 함수 종료 시 반대쪽으로 return 되면서 user쪽으로 돌아간다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

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

suggested implementation order에 따라 argument parsing/passing 먼저 하고 memory access 예외처리, 그 다음으로 system call handler 작성하고 각 함수들을 구현하였다. 마지막으로 fibonnaci, max\_of\_four\_int도 구현하였다.

* 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 parsing/passing의 경우 userprog/process.c에서 parsing 함수를 따로 구현해서 이를 load 안에 넣어주고(setup\_stack()뒤에) passing 과정도 stack\_pass() 함수를 따로 만들어 바로 뒤에 넣어준다.

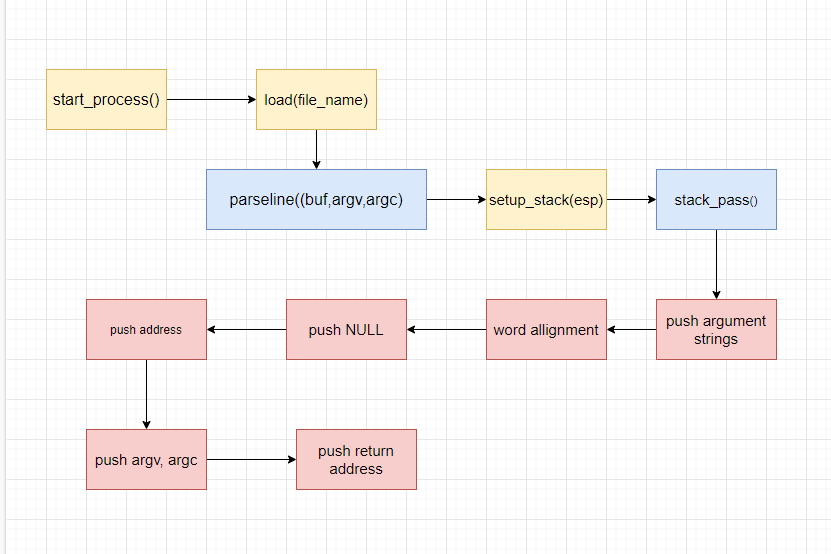
memory access는 userprog/exception.c의 page fault 함수안에 is\_kernel\_vaddr()를 통해 판별하는데 이를 위해 "threads/vaddr.h"를 include한다. 또 syscall.c에서 각 system call을 구현할 때에도 is\_user\_vaddr()를 하는 valid\_check 함수를 매번 호출해줌.

system call handler의 경우는 syscall.c 안에서 switch문을 통해 구현한다. 각 함수들의 구현도 syscall.c에서 이루어지는데 wait과 exit의 경우는 semaphore 및 list가 사용되므로 threads/thread.h에서 #ifdef USERPROG 아래에 exit\_status, list childs, list\_elem childs\_elem, 그리고 semaphore 2개를 추가해준다. 그리고 thread.c로 가서 init\_thread 함수에서 2개의 semaphore와 list를 init해준다.

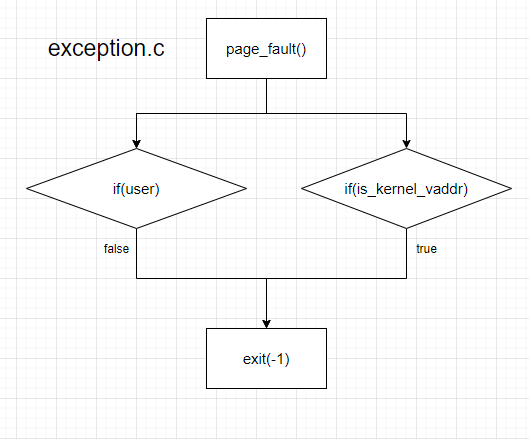
**연구 결과**

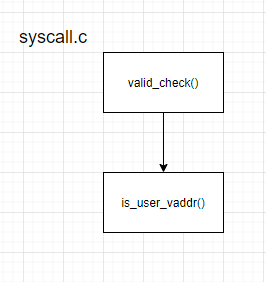
* 1. **Flow Chart**
* **II. B. 개발 내용에 대한 Flow Chart를 작성**

1. Argument Passing

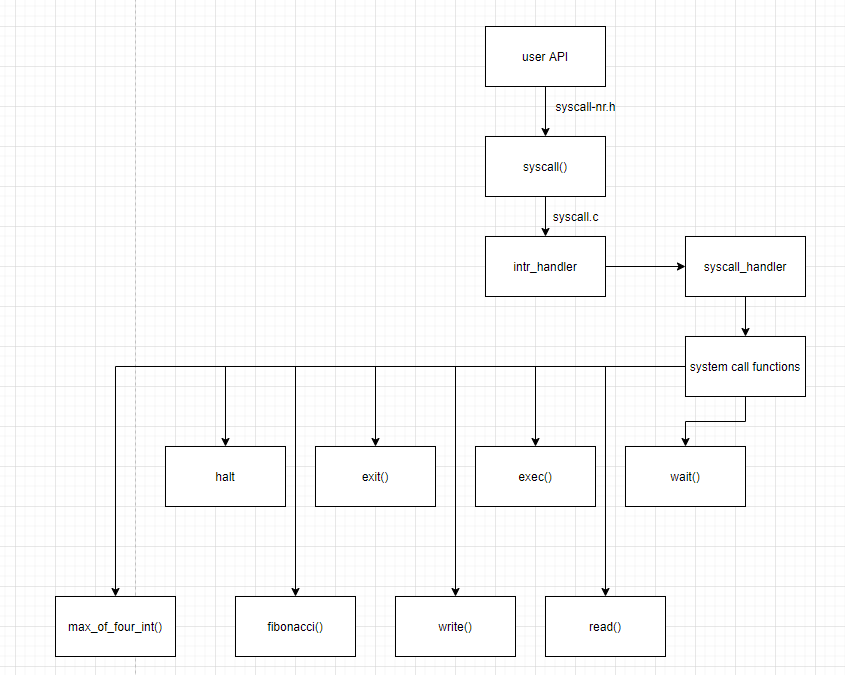


2. User Memory Access





3. 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

B 개발 내용에서 자세하게 설명했기 때문에 간단하게 stack\_pass 함수 코드 내용만 살펴보면 다음과 같다. 여기서 전달받은 parameter는 parse된 문자열 집합 argv와 그 개수 argc, 그리고 stack의 esp이다. 각각의 요소들을 push 할 때에는 \*esp를 잘 낮춰주고( -를 의미) str1cpy 혹은 (\*esp)= 꼴 혹은 memset((\*esp), ,) 꼴로 하였다. 여기서 esp의 포인터 형을 그때 그때 지정하는데 조금 어려움이 있었지만 익숙해지면 괜찮다.

1. User Memory Access

syscall.c에서 쓰이는 void valid\_check(const void\* addr) 함수는 is\_user\_vaddr() 함수를 이용하여 이것이 valid한 memory access인지 확인한다. 또한 exception.c에서 page\_fault 함수 안에서도 다음과 같은 코드를 추가한다.

if(user==false || is\_kernel\_vaddr(fault\_addr))

exit(-1);

1. System Calls

* **이번 프로젝트에서 개발한 시스템 콜을 구현 관점에서 상세히 서술.**
* f->esp를 통해 stack에 접근하여 switch문 이용. (각 SYS\_HALT, SYS\_EXIT 등은 syscall-nr.h에서 enum되어 있다) 각 case에 대해 각각의 함수를 호출하는데 여기서 parameter로 전달하는 f->esp 관련 값들(주로 인자 1개이므로 f->esp+4인데 read, write는 3개씩이므로 +8, +12도 있음)은 먼저 valid\_check를 통해 valid한 주소인지 확인을 한다. 각 함수의 꼴에 맞게 전달하는 형태를 잘 조정해야 하는데 exit이나 read/write의 첫번째 parameter 같은 경우는 int 형이므로 \*(int\*)(f->esp+4)를 넘겨주고 exec의 경우는 const char\*를 parameter로 가지므로 (const char\*)\*(uint32\_t\*)(f->esp+4)와 같이 넘겨준다.

halt는 shutdown\_power\_off();를 이용, read나 write도 stdin, stdout만 고려하면 되므로 input\_getc()와 putbuf()를 이용하면 간단히 구현 가능하다. exit의 경우 현재 exit하는 thread의 exit\_status 요소(앞에서 말했듯이 thread.h에 추가)에 parameter로 온 status로 저장한다. 그리고 thread\_exit()을 통해 process\_exit()으로 가는데 여기부터는 synchronization을 고려해야한다. parent process가 child process를 만들 것이고 child보다 parent가 먼저 종료되면 안 된다. Pintos에서 child가 죽으면 그 child의 exit status를 return하라고 요구하기 때문에 앞에서 exit에서도 그것을 저장해둔 것이다. process\_wait에서 child가 종료전까지 기다리게 하는 방법을 구현해야 하는데 semaphore를 이용한다(thread.h에 추가는 위쪽 부분 참고). process\_wait 내부를 살펴보면 child list를 for문으로 돌면서 tid가 일치하는 것이 있는지 확인한다. 그리고 찾으면 sema\_down을 해줘서 parent process가 안 끝나도록 하고 child 종료시 exit에서 sema\_up을 해주면 되는데 wait내에서 child 종료 후 이를 list에서 제거하는 list\_remove 작업이 있다. exit에서 sema\_up을 통해 child가 종료되면 list\_remove에서 temp가 가리키는 것이 사라져 버리는 문제점이 생긴다. 그래서 exit쪽에서 sema\_up을 통해 child을 죽인 후에 다시 한번 sema\_down을 다른 lock으로 건다(앞의 것과 달라야 한다. 여기선 cleansema\_lock 이용 앞에선 childsema\_lock). 그러고 나면 이제 parent process가 child process의 시체를 처리한 다음에 wait쪽에서 다시 sema\_up을 해주면 list\_remove가 원활하게 이루어진다.

마지막으로 exec의 경우도 process\_execute가 호출되는데 여기서 char cop2[100]을 선언해서 이를 str1cpy로 file\_name 복사 후 앞의 parseline 함수를 통해 parse를 해주고 thread\_create 부분에 argv[0]을 넣어준다.

1. Additional System calls

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

**방법을 순서대로 작성하였다.**

* 먼저 examples로 가서 additional.c를 추가한다. $ ./additional 10 20 62 40 와 같이parameter가 5개가 넘어오는 방식으로 실행되므로 argv[1]~argv[4]를 먼저 atoi를 통해 int 형으로 바꿔주고 fibonacci(a), max\_of\_four\_int(a,b,c,d)의 결과를 출력하게 끔 짜면 된다.
* 다음으로는 lib/syscall-nr.h로 가서 enum 안에 SYS\_FIBONACCI, SYS\_MAX\_OF\_FOUR\_INT를 추가한다.
* lib/user/syscall.h에서 다음 선언을 추가한다

int fibonacci(int a); int max\_of\_four\_int(int a, int b, int c, int d);

- lib/user/syscall.c에서 인자가 4개 용도인 syscall4 함수를 define 한다. syscall3 함수에서 arg3을 하나 더 추가해서 짜면 된다. 그리고 아래에 syscall.h에서 선언한 함수를 정의하는데 fibonacci의 경우는 syscall1, max\_of\_four\_int의 경우는 syscall4를 이용한다.

- 마지막으로 userprog/syscall.c로 가서 두 함수 내용을 구현하면 된다. 이는 2.B에서 설명하였다.

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

  자동 생성된 설명