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

담당 교수 : 김 영 재

조 / 조원 :20171680 이 호 성

개발 기간 : 9/12~

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**
* 여러가지 system call 이 들어왔을 때 명령어를 스택에 쌓는 것부터 하여 정상적으로 system call을 호출하여 작업이 완료되도록 하는 것이 해당 프로젝트의 개발 목표이다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

들어온 arguments 들이 띄어 쓰기 단위로 parsing 되어 esp 스택에 쌓여야 한다.

1. User Memory Access

접근 하려는 address 가 valid 한 address 인지 확인 해야 한다.

1. System Calls

현재 작동하고 있지 않은 여러가지 system call(halt, exit, exec, wait, read(stdin), write(stdout)) 들이 해당 작업을 할 수 있게 구현해야 한다.

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

pintos –filesys-size=2 -p ../examples/echo -a echo -- -f -q run ‘echo x’ 에서 들어온 argument ‘echo x’ 가 file\_name 으로 들어온다. 이 때 file\_name 을 띄어쓰기 단위로 parsing 해서 esp 스택에 passing을 한다. 그 다음 word align을 맞춘 후 NULL을 넣는다. 그다음 arguments 들의 주소값들을 쌓고 agrv의 주소값, argc, 0을 순서대로 stack에 쌓는다.

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

user mode 에서는 kernel address에 접근을 하여서는 안된다. 권한이 없는 kernel address 에 접근 할 때 invalid memory access 가 발생한다.

* + Invalid memory access를 어떻게 막을 것인지 설명

~/pintos/src/threads/vaddr.h 에 있는 is\_user\_vaddr 을 이용하여 address 가 valid 한지 확인 할 것이다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명

시스템 콜은 운영체제에 의해 사용 가능하게 된 서비스에 대한 인터페이스를 제공한다. 이와 같은 서비스는 누구나 조건 없이 kernel 모드에 접근을 할 수 있다면 악의적인 의도로 인해 시스템 전체가 쉽게 망가질 수 있으므로 권한이 필요하다. 그러므로 user process 들은 kernel 모드로 전환을 하여야 해당 작업을 수행할 권한이 생기도록 하였고 이를 수행하는 시스템 콜이 필요해진 것이다.

* + 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 강제로 종료한다.

exit : 해당 thread 를 종료한다. 종료한 stauts도 함께 저장한다.

exec : 인자로 들어온 cmd\_line 을 수행한다.

wait : 인자로 들어온 pid가 끝날 때 까지 기다린다.

read(stdin) : stdin의 filediscriptor 가 0이므로 fd==0 인 경우 다 읽은 후 읽은 byte 수를 반환 한다.

write(stdout) : stdout의 filediscriptor 가 1이므로 fd==1 인 경우 buffer 를 putbuf()를 통해 출력한 후 size를 return 한다.

fibonacci : 피보나치 수열의 n 번째 수를 return 한다.

max\_of\_four\_int : 4가지 정수 중 가장 큰 수를 return 한다.

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

syscall 명령이 들어오면 ~/pintos/src/lib/user/syscall.c에 있는 해당 명령어의 함수들이 syscall()을 호출한다. 이때 ~/pintos/src/lib/user/syscall.c 에있는 #define syscall()에서 Trap 명령어인 int $0x30으로 kernel영역인 Interrupt Vector table의 0x30 영역인 syscall\_handler()를 호출한다. 여기서 ~/pintos/src/userprog/syscall.c의 syscall\_handler()로 가게 되며 여기서 구현할 함수들을 통해 system call 작업을 수행한 후 return-from-trap하여 다시 유저 레벨로 돌아온다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

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

9.13 – argument passing 구현

9.14 – argument parsing 구현

9.20 – user memory access , system call 구현

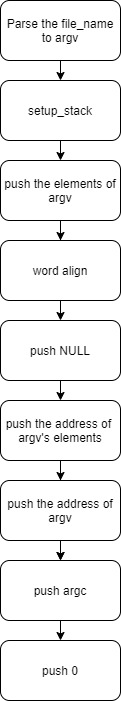
9.21 – fibonacci(), max\_of\_four\_int() 구현

* 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
  + ~/pintos/src/userprog/process.c의 load 함수에서 문자열 parsing하는 부분과 setup\_stack(\*esp) 가 완료된 후 passing을 진행하는 코드를 추가할 것이다.
* User Memory Access
  + ~/pintos/src/userprog/syscall.c에서 check\_user\_vaddr() 함수를 만들어 valid 한 address인지 확인할 것이다.
* System Call
  + ~/pintos/src/userprog/syscall.c 에 들어올 수 있는 여러가지 intr\_frame 에 대하여 switch 문을 통하여 case 를 나누고 각각의 함수를 호출한다. 위에서 언급했던 halt, exit, exec, wait, read(stdin), write(stdout) 과 fibonacci, max\_of\_four\_int 를 추가할 것이다.

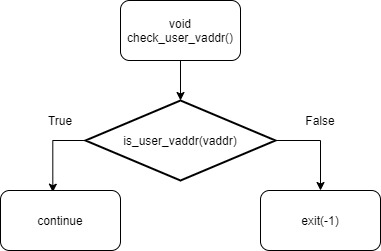
1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

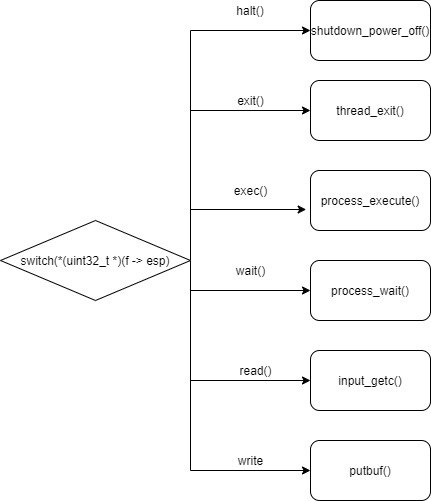
1. Argument Passing



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

* ~/pintos/src/userprog/process.c의 load 함수에서 우선 \*\*argv에 argument parsing을 해준다. 그 후 setup\_stack(esp)가 완료된 후 esp 스택에 쌓기를 시작하는데 우선 \*argv의 원소들을 먼저 쌓은 후 word allign을 해준다 그 후 NULL, \*argv의 원소들의 주소값, agrv의 주소값, argc, 0을 순차적으로 스택에 쌓는다.

1. User Memory Access

* System call을 할 때 address 가 valid 한지를 확인할 것이기 때문에 ~/pintos/src/syscall.c 에 확인하는 함수 check\_user\_vaddr()을 작성한다. 그 후 syscall.c에서 switch문을 활용한 case에서 활용하여 address들을 확인하며 invalid memory access를 방지한다.

1. System Calls

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

~/pintos/src/userprog/syscall.c에 들어올 수 있는 여러가지 intr\_frame에 대하여 switch 문을 통하여 case를 나누고 각각의 함수를 호출한다. 위에서 언급했던 halt, exit, exec, wait, read(stdin), write(stdout)를 구현한다. 매뉴얼에 적힌 함수의 프로토 타입을 이용하여 함수를 구현한다.

**halt()**의 프로토타입은 void halt (void)로 shutdown\_power\_off() 함수를 이용하여 pintos를 강제 종료하면 된다.

**read()** 와 **write()**의 프로토타입은 각각 int read(int fd,void \*buffer, unsigned size) / int write(int fd,const void \*buffer, unsigned size) 로 아직은 stdin/stdout 에서만 구현하면 되기 때문에 read() 는 input\_getc()를 이용하며 write() 는 putbuf() 함수를 이용하여 buffer를 처리하고 size를 return 한다. 이때 return 값은 eax에 저장되어야 하므로 syscall\_handler에서 f->eax에 함수의 return 값을 저장하도록 한다. 하지만 여기까지 한 후 pintos 명령어로 echo x를 출력하려고 할 때 출력이 되지 않았는데 write를 할 때에는 f->esp + 4 로 출력을 하려하면 출력이 되지 않았다. 그래서 hex\_dump로 확인 한 결과 +20부터 시작해야 정상적으로 echo x 가 출력이 되기 시작하였다.

**exit()**의 프로토 타입은 void exit (int status)로 thread\_exit() 함수를 이용할 것이다. 이때 어떤 thread를 어떤 status로 종료한지를 출력을 하고 thread\_exit()을 한다. 이 때 출력을 해 보면 echo : exit(0)이 아닌 echo x : exit(0)이 뜨게 되는데 load 함수 말고도 process\_execute()에서도 parsing을 해야하는 부분이 있었다. filesys\_open()하는 부분을 file\_name전체가 아닌 첫 인자만 넘겨야 하기 때문에 제일 앞 인자만 parsing 을 하여 filesys\_open(argv)/thread\_create(argv,…)로 변경하였다.

이후 make check를 해보면 sc-bad test 들이 kernel 영역을 참조하고 있다고 안될 때 exit(-1)이 출력이 되어야 하는데 page fault 가 떠버리며 출력이 안되는 것 같다. 그러므로 ~/pintos/src/userprog/exception.c 에 들어가서 user program이 kernel mode 에 참조하려 할 때 kill 하기 전에 exit(-1)을 실행 하면 될 것 같아 수정 하였고 통과 하였다.

**exec()**의 경우 프로토타입은 pid\_t exec (const char\* cmd\_line)으로 인자로 들어온 cmd\_line을 수행을 하는 system call이다. 그러므로 process\_execute()를 통해 구현한다.

**wait()**의 프로토 타입은 int wait(pid\_t pid)로 인자로 들어온 process가 끝나길 기다리는 것이므로 process\_wait()을 수정하여 활용한다. 여기서 기다린다는 것을 구현하기 위해 synchronization 기법을 사용해야한다. ~pintos/src/threads/synch.c를 보니 semaphore에 관한 함수들이 있어 semaphore를 사용하여 wait()을 구현하기로 한다. ~/pintos/src/userprog/process.c process\_wait()에 가면 child\_tid 가 죽은 방식에 따라 return 값이 달라지므로 parent 는 child 의 thread를 알고 있어야 하므로 thread 구조체에 자식에 관한 원소들을 추가하여야한다. ~pintos/src/threads/thread.h 의 struct thread 에 synchronization을 구현 하기 위한 construct semaphore child\_lock, pintos 의 linked list를 만드는 struct list child, linked list의 head와 tail을 알 수 있게 struct list\_elem child\_elem, exit\_status 를 알 수 있는 int exit\_status를 추가하고 thread.c 의 init\_thread 부분에 위에 추가한 변수들을 initialize 하는 부분도 추가한다. 이제 ~pintos/src/userprog/process.c의 process\_wait()를 수정해야한다. thread\_current()의 child들을 돌아보면서 내가 기다리고 있는 child\_tid 가 맞다면 sema\_down()을 통해 child 가 끝날 때까지 sema\_down을 통해 기다린다. 여기서 sema\_down() 은 sema-value가 0일 때 thread\_block()을 계속 해준다. 그러므로 child\_tid 가 process\_exit()할 때 sema\_up()을 해주게 한다면 child\_tid가 끝날 때까지 wait 할 수 있게 된다. 그 후 sema\_down이 끝나면 exit\_status를 설정해주고 child\_elem을 list에서 제거해준다. 여기까지 했을 때 pintos 명령어를 실행 해 보면 wait하지 못하고 시스템이 죽어버리는데. 그 이유는 process\_exit을 할 때 child\_tid 가 죽는다면 list\_remove를 못하기 때문이다. 그러므로 다른 하나의 semaphore mem\_lock을 만들어 process\_exit()에서 sema\_up(&(cur->child\_lock)), sema\_down(&(cur->mem\_lock))를 추가하고 process\_wait()에서 list\_remove() 다음에 sema\_up(&(cur->mem\_lock))을 추가한다. 그 결과 parent thread는 sema\_down()을 통해 child\_tid가 끝날 때까지 기다린다. child\_tid 가 process\_exit()을 통해 종료될 때 sema\_up(&(cur->child\_lock))을 통해 process\_wait()을 하고 있는 parent thread에게 child\_thread를 list\_remove()할 수 있게 한다. parent thread의 list에서 child\_tid를 삭제하였다면 이제서야 child\_tid를 종료할 수 있게 sema\_up(&(mem\_lock))을 통해 child\_tid가 종료될 수 있게 한다.

1. Additional System calls

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

~/pintos/src/examples/Makefile에서 additional \_SRC = addirional.c 라는 코드를 추가하고 ~/pintos/src/examples/ 에서 additional.c 파일을 만든 후 make 하여 additional 이라는 pintos 명령어가 들어왔을 때 처리 할 수 있게 해준다.

~/pintos/src/lib/user/syscall.h에 두 함수의 프로토타입 추가

* ~/pintos/src/lib/user/syscall.c에서 두 함수가 불렸을 때 syscall 호출

~/pintos/src/lib/syscall-nr.h의 enum 부분에 SYS\_FIB, SYS\_MOFI 추가

~/pintos/src/userprog/syscall.h에도 프로토타입 추가

~/pintos/src/userprog/syscall.c에서 실제 fibonacci() 와 max\_of\_four\_int()를 구하는 함수를 작성한다. fibonacci()는 iterative 하게 작성하며 max\_of\_four\_int()는 조건문을 통해 가장 큰 수를 찾아낸다.

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