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

담당 교수 : 김영재

조 / 조원 : 문상영

개발 기간 :~2021/10/2

1. **개발 목표**

pintos 상에서 입력된 command를 parsing하고 parsing 된 argument를 이용해 user stack을 construct 한다(Argument Passing).

virtual memory 에서 user 메모리 영역에서 kernel 메모리 영역을 접근하는 것을 막아 메모리 안정성을 확보한다. (User Memory Access)

syscall handler를 작성해서 userprogram 에서 systemcall 함수를 이용해서 system 작업을 수행할 수 있게 한다. (System call)

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

입력된 command를 parsing해서 실행파일을 load하고 user stack에 argument를 push 해서 저장한다.

1. User Memory Access

user 메모리 영역에서 kernel 메모리 영역에 접근하는 것을 막아 kernel 메모리의 안정성을 확보한다.

1. System Calls

kernel 메모리에 대해서 User Program 의 Memory Access가 제한되므로 상황에 따라 system 명령을 수행할 필요가 있을 때(파일 읽기, 쓰기 등), user program은 system call 을 이용해서 system 명령을 수행할 수 있다.

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

입력된 command 를 load할 file\_name과 argument를 parsing 한뒤, 할당된 user stack의 공간에 접근하기 위해서 intr\_frame f 구조체 요소인 (void \*) esp를 사용한다. parsing 된 argument 중 마지막 argument 부터 사이즈에 맞게 esp를 감소시키면서 downward 방향으로 push 한다. argument를 push하고나면 word alignment, null, 각 argument의 주소값, 배열 arg의 주소값, argc , return address 등을 차례대로 push 한다.

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

가상 메모리에서 각 프로세스의 user 메모리와 kernel 메모리가 있는데 user 메모리는 kernel 메모리 corruption을 방지하기 위해서 kernel 영역에 접근할 수 없고, 대신 system call을 이용해서 system 명령을 수행할 수 있다.

따라서 invalid memory access 는 user program 이 kernel 영역을 참조하거나, 존재하지 않는 메모리 주소를 참조하는 경우가 있다고 할 수 있다.

* + Invalid memory access를 어떻게 막을 것인지 설명

PHYS\_BASE를 기준으로 kernel 메모리 영역과 user 메모리 영역이 나뉘기 때문에 접근하는 메모리 영역의 주소를 PHYS\_BASE와 비교해서 PHYS\_BASE보다 크거나 같으면 kernel 메모리 영역 ,PHYS\_BASE 보다 작으면 user memory 영역으로 판단해서 user program에서 메모리를 참조할 때 접근을 체크한다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명

User Memory Access 에서 프로그램 안정성과 Virtualization에 의해서 다른 프로세스간의 메모리 영역, kernel 메모리 영역 접근이 금지되어 있다. 그런데 I/O 와 같이 User program 에서도 system 명령에 접근해야하는 경우가 있기 때문에 system call을 이용해서 해당 system 명령을 수행하도록 kernel 에 요청할 수 있다.

* + 이번 프로젝트에서 개발할 시스템 콜에 대한 간략한 설명 (하나의 시스템 콜 당 최대 3문장으로 간략히 설명; 3문장을 넘길 정도로 길게 작성하지 말 것)
* 1) halt(): pintos 시스템을 종료.
* 2) exit() : current user process(thread)를 종료하고 exit\_status를 kernel에 전달, exit message 출력
* 3) exec() : child process(thread)를 생성하고 실행
* 4) wait() : child process(thread)가 종료될 때 까지 대기하고 exit status를 return 받음
* 5) read() : 프로젝트1 에서는 fd=0 (stdin)에 대해서 file 읽기를 수행
* 6) write() : 프로젝트1 에서는 fd=1 (stdout)에 대해서 file 쓰기를 수행
* 7) fibonacci() : parameter로 전달된 integer n에 대해 n번째 피보나치 수를 반환
* 8) max\_of\_four\_int() : parameter로 전달된 4개의 integer 중 가장 큰 1개를 반환
  + 유저 레벨에서 시스템 콜 API를 호출한 이후 커널을 거쳐 다시 유저 레벨로 돌아올 때까지 각 요소를 설명

system call user API를 호출하고 해당 함수 내에서 syscall 매크로를 호출한다.해당 매크로는 어셈블리 코드로 syscall number와 해당 매크로의 argument 개수에 해당하는 만큼의 argument를 전달받아 userstack에 argument를 push하고 int $0x30로 interrupt를 raise한다. 이 때, 0x30은 syscall handler가 등록된 주소로 intr\_handler에서 syscall\_handler를 호출하고 이번 프로젝트에서 작성할 syscall handler가 수행된다. syscall handler는 intr\_frame\* f를 참조해 esp로 userstack에 push 된 syscall number를 참조하고 push 된 다른 argument를 parameter로 해당 number에 알맞은 작업을 수행한뒤 요청한 system 명령을 수행하고 필요하면 결과를 intr\_frame \*f의 eax 에 저장해서 반환한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

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

1) 프로젝트 설계 (9/15 ~ 9/16)

2) Argument Passing : 입력된 command를 parsing 하는 parer와 user stack을 construct 하는 과정. (9/17 ~ 9/20)

3) User Memory Access : user stack 을 참조하거나 주소를 참조할 때 invalid 한 메모리 영역에 대해서 접근을 방지하고 메모리를 보호 (9/20 ~ 9/21)

4) System Call Handler : intr\_frame의 esp를 참조해서 system call handler에서 system number에 맞게 argument를 참조하고 잘 passing 되었는지 확인하기. (9/22 ~ 9/26)

5) Systme Call Implementation: wait를 구현하기 전까지는 반복문을 이용해서 busy waiting을 이용한다. intr\_frame\* f 의 esp를 dereferencing 해서 parameter를 전달하고, 각 system call의 기능을 구현한다. return 값은 intr\_frame의 eax에 저장한다.

(9/22 ~ 9/26)

6) Additional Implementation : max\_of\_four\_int()와 fibonacci() 를 system call로 구현하기위해 syscall4 매크로를 정의하고 syscall number와 그에 따른 작업을 syscall handler 에 등록한다. (9/27 ~ 9/29)

* 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

입력된 command 를 load할 file\_name과 argument를 parsing 하기 위해, src/userprog/process.c의 start\_process에서 parameter로 전달 받은 file\_name을 parsing 해주는 parse\_file\_name 이라는 함수를 정의해 사용한다.

해당 함수는 공백 기준으로 file\_name을 나누어 argc개의 argument를 char\* parsed\_filename\_argv[] 에 저장한다.

user stack을 construct 하기 위해서 push\_userstack 이라는 함수를 정의한다.

parsed\_filename\_argv, argc, &if\_.esp를 parameter로 전달해서 push\_userstack 안에서 esp값을 감소시키면서 downward로 argument를 push해 저장한다.

* User Memory Access

invalid memory access 는 user program 이 kernel 영역을 참조하거나, 존재하지 않는 메모리 주소를 참조하는 경우가 있는데, 이를 확인하기 위해서 PHYS\_BASE 값과 대소를 비교해보면

user memory 영역주소 <PHY\_BASE <= kernel memory 영역 이므로 intr\_frame \*f 를 이용해 f->esp를 dereferencing 할 때, f->esp의 값을 PHY\_BASE와 비교해 user 메모리 영역이 아니면 exit(-1)로 처리한다.

또 page\_fault가 발생 했을때, exception.c의 page\_fault() 함수에서 잘못된 메모리 영역을 참조한다면 오류를 발생시키지 않도록 exit(-1)을 호출한다.

* System Calls

정의된 system call user API를 통해서 syscall1,2,3 매크로를 호출해 interrupt handler 가 호출되고 interrupt handler에서 syscall\_handler가 호출되므로 프로젝트에서 작성해야 할 부분은 syscall\_handler 부분이다. syscall1, syscall2, syscall3 매크로에서 argument의 개수에 따라서 stack에 argument를 push하고 syscall number를 push 했으므로 intr\_frame \*f 를 이용해서 f->esp를 dereferencing 해syscall number를 확인하고 이에 따라 switch 문으로 각각 처리해준다.

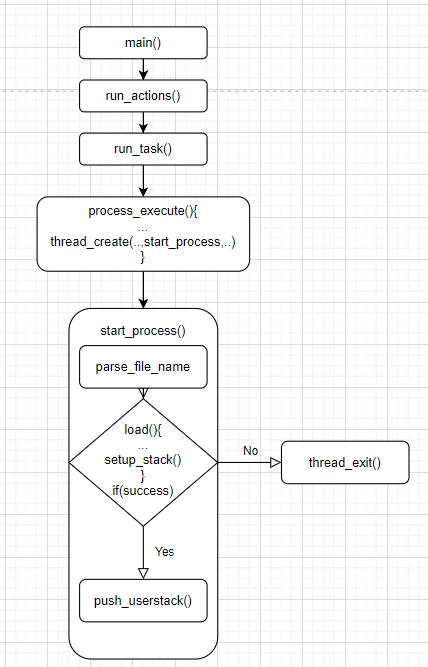
이때, push 된 다른 argument를 확인하기 위해서 f->esp이 가리키는 주소값에 값을 더해서 stack의 bottom 방향으로 argument를 dereferencing 하는데, 해당 주소값이 user 메모리 영역이 맞는지 PHYS\_BASE 값과 비교한뒤 implement한 syscall 함수들을 수행한다. 또한 리턴값이 있는경우 f->eax에 저장한다.

additional system call을 구현할 때, lib/syscall-nr.h 에 구현할 fibonacci() 와max\_of\_four\_int()에 해당하는 syscall number를 추가하고, src/examples에 4개의 argument를 전달받는 additional.c를 작성한다. object code를 make 할 수 있도록 Makefile도 수정한다. 그 이후 추가된 syscall number에 대해서 작업을 수행할 수 있도록 syscall\_handler에 switch문을 modify한다. max\_of\_four\_int 의 경우 syscall4 가 정의되어야하므로 다른 syscall 매크로를 참고해서 작성한다.

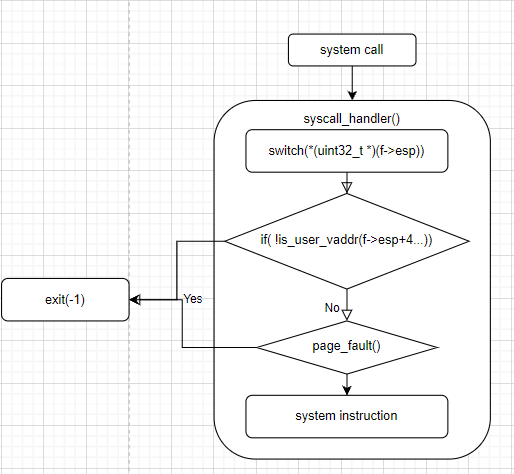
1. **연구 결과**
   1. **Flow Chart**

* **II. B. 개발 내용에 대한 Flow Chart를 작성**

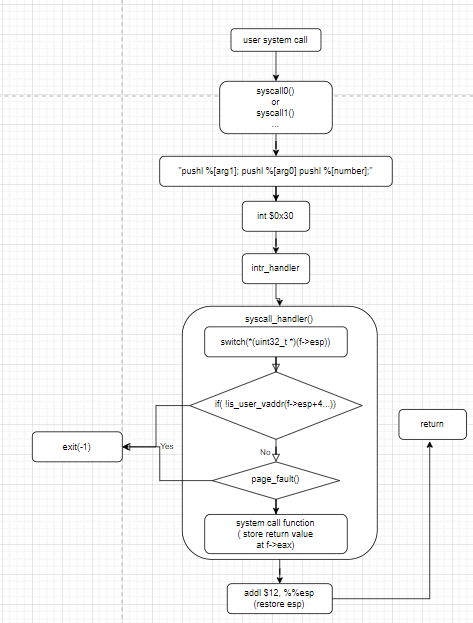
1. Argument Passing



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 passing을 구현하기 위해서 우선 입력받은 command를 parsing 하는 함수 parse\_file\_name을 구현했다.

int parse\_file\_name(char \*input, char \*\*parsed\_filename\_argv);

parse\_file\_name 함수는 전달된 input string을 공백기준으로 토큰화 하는데, 공백에 null문자를 채우고 토큰의 시작주소를 parsed\_filename\_argv[] 배열에 저장하기 때문에 start\_process 에 전달된 file\_name\_을 strlcpy를 이용해서 복사한뒤 parsing 했다.

strlcpy(input,file\_name,strlen(file\_name)+1);

argc=parse\_file\_name(input, parsed\_filename\_argv);

이렇게 parsing 된 입력에서 가장 첫번째가 load 할 file의 이름이므로 load함수에 parsed\_filename\_argv[0]을 전달한다.

load의 return 값이 true 이면 load가 성공했다는 뜻이므로 user stack을 구성한다. push\_userstack 함수를 이용해서 구현했다.

void push\_userstack(char\*\* parsed\_filename\_argv, int argc,void \*\*esp);

push 하는 argument는

1. argv[argc-1][..]~ argv[0][..]

2. word-align(4의 배수 단위가 되도록)

3. null + argv[argc-1] ~argv[0]

4. argv ,argc, return address

의 단계로 나누어 전달 받은 \*esp 값을 감소(downward)시키면서 \*\*esp 의 위치에 자료형에 맞게 값을 저장했다.

* 개발상 발생한 이슈와 해결방법

esp의 자료형이 void\*\* 였기 때문에 data를 저장할 때 올바르게 dereferencing 할 수 있게끔 type casting을 오류없이 해야했다. 이때, pintos는 메모리 주소 영역이 4byte 였기 때문에 uint32\_t \*\* 로 type casting 해서 argv[argc-1] ~ argv[0] , argv의 주소값을 저장했다.

1. User Memory Access

src/userprog/syscall.c 에서 syscall\_handler 함수가 정의되어 있다. 해당 함수에서 user stack 에 저장된 syscall number와 argument 들을 확인하기 위해서 intr\_frame \*f를 이용해 f->esp, f->esp +4 ,… 가 가리키는 값을 확인한다.

이때, 해당 주소값에 접근하기 전에 user memory 영역이 맞는지 확인한다. 이를 확인하기 위해서 src/threads 의 vaddr.h 에 정의된 is\_user\_vaddr() 함수를 사용했다.

static inline bool is\_user\_vaddr (const void \*vaddr) {

return vaddr < PHYS\_BASE;

}

is\_user\_vaddr을 이용해서 user 메모리 영역인지 확인한뒤, 만약 아니라면 exit(-1)을 수행했다.

또한 잘못된 주소 참조 때문에 page\_fault 가 발생했을 때, src/userprog/exception.c 에 page\_fault() 함수가 실행되는데 이를 modify 해서 잘못된 메모리 영역의 참조에 대해서 is\_kernel\_vaddr() 함수를 이용해서 오류가 발생하지 않도록 처리해줬다.

static inline bool is\_kernel\_vaddr (const void \*vaddr) {

return vaddr >= PHYS\_BASE;

}

1. System Calls

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

정의된 system call 을 통해서 user program에서 요청을 하면 syscall1,2,3 매크로를 통해서 interrupt handler 가 호출되고 interrupt handler에서 syscall\_handler가 호출되는 과정은 이미 구현되어 있으므로, 이번 프로젝트에서 syscall\_handler 부분을 작성했다.

syscall1, syscall2, syscall3 매크로에서 argument의 개수에 따라서 stack에 argument를 push하고 syscall number를 push 했으므로 intr\_frame \*f 를 이용해서 f->esp를 참조해서 syscall number를 확인한다.

syscall number에 따라서 argument의 개수와 어떤 data type 의 argument가 push 되었는지 다르므로 syscall number를 읽고 함수 원형을 확인한 다음. argument 들을 dereferencing한다. 이때 esp가 void\* 이므로 저장된 data type 에 따라 type casting을 알맞게 해서 가리키는 주소값의 data를 적절하게 읽어올 수 있도록 한다. 또한 앞서 말했듯, User Memory Access 를 같이 고려하면서 구현했다.

프로젝트 1에서 max\_of\_four\_int() 와 fibonacci() 함수를 제외하고 구현한 userprog system call은 우선 halt(), exit(), exec(), wait(), read(), write() (read 와 write는 각각 stdin, stdout에 대해서만 구현)이다. 구현한 방법을 간단히 설명하면.

1) halt(): shutdown\_power\_off() 함수를 사용해서 시스템을 종료.

2) exit() : exit 문을 출력하기 위해서 thread\_current()를 이용해 현재 thread 구조체를 가져온다. 그리고 parameter로 전달받은 int status를 현재 thread 구조체의 exit\_status에 저장하고, thread\_name() 함수를 사용해서 해당 thread 의 이름과 exit 상태를 출력한다. 그리고 thread\_exit()을 이용해서 thread를 종료한다.

3) exec() : process\_execute() 함수를 사용해서 구현, parameter로 const char\* cmd\_line을 전달해서 수행한다.

4) wait() : process\_wait() 함수를 통해 child process(thread)를 기다린다. 이때 thread 구조체에 child thread list와 semaphore를 추가해서 process\_wait 함수에서 child thread list를 순회하면서 wait 하려는 thread의 semaphore를 down시켜 가 wake 될 때까지 기다리다가 child thread 가 종료되면 semaphore 가 up 되어 parent를 wake 하고 parnent 도 종료된다.

5) read() : fd가 0(stdin)이면 buffer를 읽어, file read를 진행한다. size를 return 한다.

6) write() : fd가 1(stdout)이면 putbuf() 함수를 사용해서 file에 쓰기를 진행한다.

1. Additional System calls

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

lib/syscall-nr.h 에 fibonacci() 와 max\_of\_four\_int()에 해당하는 syscall number ( SYS\_FIBO, SYS\_MAXFOUR )를 추가하고, src/examples에 additional.c를 작성한다. additional.c 는 시스템 콜인 fibonacci() 와 max\_of\_four\_int()를 호출한다. object code를 make 할 수 있도록 Makefile도 수정한다.

max\_of\_four\_int 의 경우 syscall4 가 정의되어야하므로 lib/user/syscall.c에 다른 syscall 매크로를 참고해서 4개의 argument와 새로 추가한 system call number가 user stack 에 push 될 수 있도록 하고, 다시 esp를 복구하는 과정도 argument 개수에 맞게 addl $20 %%esp 한다.

/\* lib/user/syscall.c \*/

int fibonacci(int n){

return syscall1(SYS\_FIBO, n);

}

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

return syscall4(SYS\_MAXFOUR,a,b,c,d);

}

그 이후 src/userprog/syscall.c 파일의 syscall\_handler 함수에서, 추가된 syscall number에 대해서 작업을 수행할 수 있도록 switch 문을 modify 하는데 f->esp를 이용해 참조하는 메모리 주소가 user 메모리 영역인지 is\_user\_vaddr() 함수로 확인하고 실제 작업을 수행하는 implement 함수에 parameter를 전달한다.

/\* src/userprog/syscall.c \*/

int fibonacci(int n);

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

int를 return 하도록 구현했으므로, f->eax에 return 값을 저장한다.

* 1. **시험 및 평가 내용**
* **fibonacci 및 max\_of\_four\_int 시스템 콜 수행 결과를 캡처하여 첨부.**
* **수행결과 1. (10 20 62 40) 예상 결과: 55 62 실제 결과 :55 62**
* 텍스트이(가) 표시된 사진

  자동 생성된 설명
* **수행결과2. ( 5 1 13 2) 예상 결과: 5 13 실제 결과 :5 13**
* 텍스트이(가) 표시된 사진

  자동 생성된 설명