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

담당 교수 : 박성용 교수님

학번 / 이름 : 20181683 전용본

개발 기간 : 9/19 ~ 9/27

1. **개발 목표**

* **해당 프로젝트에서 구현할 내용을 간략히 서술.**

초기 pintOS는 OS로서 기능을 전혀 하지 못하고 있다. 기본적인 system call, system call handler, argument passing, construct user stack 등의 OS로서 갖고 있어야할 기능들이 구현되어 있지 않기 때문이다. 예를 들어 ‘echo x’ 명령어를 실행할 때 내부에서 argument passing 기능이 없으므로 echo와 x를 구분하지 못하고, user stack이 없으므로 ‘x’를 argument로 넘겨줄 수도, write(system call)이 구현되어 있지 않기에 console에 출력할 수도 없다.

이러한 문제들을 해결하기 위해 OS로서의 기본적인 기능을 구현하고 pintOS의 21 tests 를 통과하고 간단한 새로운 system call을 만들어 보는 것이 목적이다.

1. **개발 범위 및 내용**
   1. **개발 범위**

* **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Argument Passing

초기 pintOS에서는 입력 받은 명령어에 대해서 argument parsing을 하지 않았기에 file, arguments들을 구분하지 못했고 stack에 이들을 적재해 사용하지 못했다. 적절하게 이들을 parsing한 후에 80x86 calling convention에 맞춰 적재할 수 있게 구현한다.

1. User Memory Access

기존에는 user가 kernel memory, invalid address등에 접근할 경우에 이를 확인하지 못해 kernel panic 등 error가 발생했기에 이를 조치할 수 있게 구현한다.

1. System Calls

기본적인 system calls(halt, exit, wait, exec, read, write)를 구현해 user program에서 이를 적절하게 사용할 수 있게 했다.

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

입력으로 들어온 command를 space ‘ ‘를 기준으로 parse해서 arguments를 구성한 후에 80x86 calling convention에 맞춰서 kernel stack에 적재한다. PHYS\_BASE부터 top에서 bottom 구조로 내려가면서 esp에 arguments, 계산을 용이하기 위해 word alignment를 적재해서 4의 배수를 맞추고, argument addresses, number of arguments, return address(fake)를 stack에 적재한다.

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

invalid memory access는 User Program이 사용할 수 있는 메모리 공간은 0x00000000 와 0xc0000000 사이이고 이 메모리가 아닌 다른 곳을 접근하려고 하는 것이다.

* + Invalid memory access를 어떻게 막을 것인지 설명

접근하려고 하는 virtual memory가 유효한지, 커널에 접근하려고 하는지 is\_user(kernel)\_vaddr 함수를 이용해 확인하고 invalid memory access하려고 하는 경우 exit(-1)를 이용해 종료한다.

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명

유저레벨의 프로그램들은 유저레벨의 함수들 만으로는 많은 기능을 구현하기 힘들기에 커널의 도움을 받는다. 도움을 받기 위해서 커널에게 권한을 주고 수행할 수 있게 할 때 시스템 콜이 필요하다.

* + 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

shutdown\_power\_off 함수를 이용해 pintOS를 종료시킨다.

* + 1. wait

자식(타겟) 프로세스가 종료되길 기다린 후에 reaping한다.

* + 1. exit

실행 중인 process를 종료하고 kernel에게 종료 status를 반환한다.

* + 1. exec

현재 메모리 공간에서 새로운 프로세스를 실행시킨다.

* + 1. read

input\_getc 함수를 이용해 STDIN으로 받은 내용을 읽는다.

* + 1. write

putbuf 함수를 이용해 버퍼의 내용을 STDOUT으로 출력한다.

* + 1. fibonacci

피보나치 수열의 n번째 수열을 반환한다.

* + 1. max\_of\_four\_int

입력으로 받은 4개의 인자 중 가장 큰 값을 반환한다.

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

유저레벨에서 시스템 콜을 호출하면 시스템 콜 번호와 인자들이 스택에 쌓인다. int $0x30 instruction을 이용해 interrupt를 발생시켜 인터럽트 핸들러를 호출하고 여기서 vector table을 이용해 시스템 콜 핸들러를 호출한다. 여기서 스택에 쌓인 인자들을 이용해 시스템 콜 번호에 맞는 적절한 시스템 콜을 호출한 뒤에 작업이 끝나면 intr\_exit 함수를 이용해 유저 레벨로 돌아온다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

* **II. A.의 개발 범위를 포함하여 구현 내용에 대한 일정 작성**
  + 1. Argument Passing 구현 – 9/19 ~.9/22
    2. User Memory Access 구현 – 9/23 ~ 9/24
    3. System calls 구현 – 9/25~ 9/28
    4. Additional System calls 구현 – 9/29 ~.9/30
    5. Document 작성 – 9/31 ~ 10/3
  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

pintos/src/userprog/process.c load 함수를 수정한다. parameter로 주어진 file\_name을 space ‘ ‘를 기준으로 parsing해서 배열에 저장한다. parsing한 executable file name을 이용해 filesys\_open 함수를 이용해 file을 open한다.

setup\_stack 함수에서 esp가 초기화된 후에 construct\_stack 함수를 구현하여 esp를 이용해 위에서 아래로 argv, address of argv, argc, return address를 80x86 convention에 맞춰서 stack에 적재한다.

* + 1. User Memory Access

pintos/src/userprog/exception.c page\_fault 함수를 수정한다. pintos/src/ threads/vaddr.h 에 있는 is\_user(kernel)\_vaddr 함수를 이용해 kernel memory 영역에 접근할 경우 exit(-1)을 호출해 page fault를 방지한다.

pintos/src/userprog/syscall.c syscall\_handler 함수를 수정한다. system call의 인자들의 주소가 user 메모리 영역에 있는지 is\_user(kernel)\_vaddr 함수를 이용해 확인한다. kernel memory 영역에 접근할 경우 exit(-1)을 호출해 invalid memory access를 방지한다.

pintos/src/userprog/process.c process\_execute 함수를 수정한다. user가 입력한 command line에서 executable file name이 memory에 있는지 filesys\_open 함수를 이용해 thread를 생성하기 전에 확인해서 open failed를 방지한다.

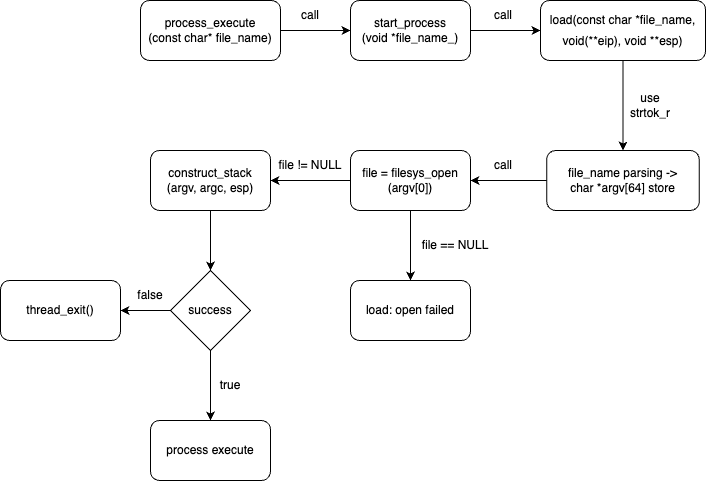
* + 1. System calls

pintos/src/userprog/syscall.c syscall\_handler 함수를 수정한다. pintos/src/ lib/syscall-hr.h에 있는 system call number를 파악해서 stack에 argument가 몇개가 있는지 알 수 있다. 인자로 넘어온 intr\_frame 구조체의 esp로 argument들을 파악하고 이를 이용해 switch-case 구문에서 handler가 각 system call을 적절하게 호출한다. return 값이 있는 system call은 intr\_frame 구조체의 eax에 저장해서 값이 반환될 수 있게 한다.

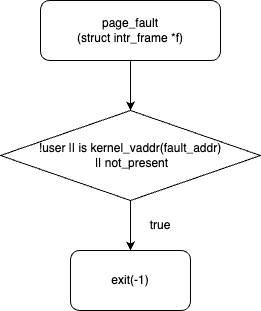
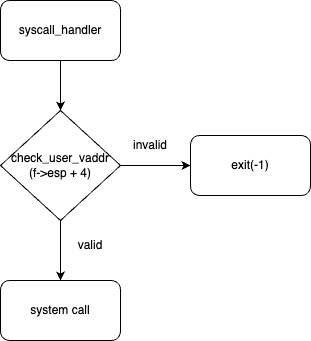
1. **연구 결과**
   1. **Flow Chart**

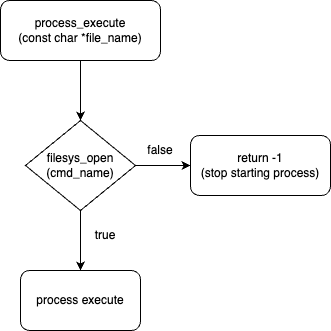
* **II. B. 개발 내용에 대한 Flow Chart를 작성**

1. Argument Passing

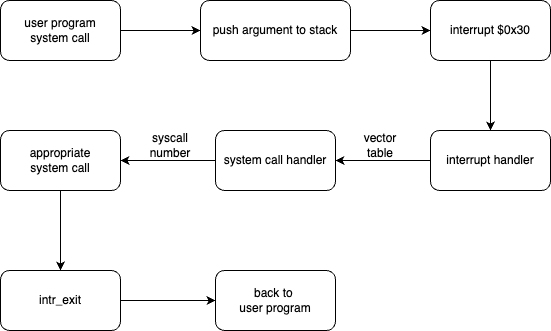


1. User Memory Access



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
   1. Parse file\_name

bool

load (const char \*file\_name, void (\*\*eip) (void), void \*\*esp)

{

struct thread \*t = thread\_current ();

struct Elf32\_Ehdr ehdr;

struct file \*file = NULL;

off\_t file\_ofs;

bool success = false;

int i;

/\* Allocate and activate page directory. \*/

t->pagedir = pagedir\_create ();

if (t->pagedir == NULL)

goto done;

process\_activate ();

/\* TODO : parse file name \*/

char temp\_filename[128];

char \*argv[64]; // parsed argv

int argc = 0; // number of argv

char \*next\_ptr;

char \*token;

strlcpy(temp\_filename, file\_name, strlen(file\_name) + 1);

token = strtok\_r(temp\_filename, " ", &next\_ptr);

while(token != NULL){

argv[argc++] = token;

token = strtok\_r(NULL, " ", &next\_ptr);

}

pintos/src/lib/stdlib.h 에서 주어진 strlcpy, strtok\_r 함수를 이용해 인자로 주어진 file name을 executable file name, arguments 들로 tokenize해서 char\* argv[64]에 저장한다.

* 1. check executable file is valid

/\* Open executable file. \*/

file = filesys\_open (argv[0]);

if (file == NULL)

{

printf ("load: %s: open failed\n", file\_name);

goto done;

}

pintos/src/userprog/filesys.h 에서 주어진 filesys\_open 함수를 이용해 executable file이 open 가능한 지 확인한다. 초기 pintOS에서는 filesys\_open의 parameter로 tokenize되지 않은 file name이 들어가서 error가 발생했기에 수정했다.

* 1. construct stack

void construct\_stack(char \*\*argv, int argc, void \*\*esp){

/\* push argument \*/

for(int i = argc-1 ; i >= 0 ; i--){

for(int j = strlen(argv[i]) ; j >= 0 ; j--){

/\* 스택 주소 감소시키면서 인자를 스택에 삽입 \*/

\*esp = \*esp -1;

\*\*(char \*\*)esp = argv[i][j];

}

argv[i] = \*esp;

}

/\* Word alignment \*/

int argv\_length = (int)(PHYS\_BASE - \*esp);

int word\_alignment = 0;

if(argv\_length % 4 != 0){

word\_alignment = 4 - (argv\_length) % 4;

}

\*esp -= word\_alignment;

/\* Push address of argv in stack , argv[argc] ~ argv[0] \*/

\*esp -= 4;

\*\*(uint32\_t \*\*)esp = 0;

for (int i = argc - 1; 0 <= i; i--){

\*esp -= 4;

\*\*(uint32\_t \*\*)esp = argv[i];

}

/\* Push address of start of argv \*/

\*esp -= 4;

\*\*(uint32\_t \*\*)esp = \*esp + 4;

/\* Push num of argv \*/

\*esp -= 4;

\*\*(uint32\_t \*\*)esp = argc;

/\* Push return address \*/

\*esp -= 4;

\*\*(uint32\_t\*\*)esp = 0;

}

setup\_stack(esp)에서 적절하게 초기화된 esp를 기준으로 위에서 tokenize한 arguments들을 80x86 convention에 맞게 stack에 적재하는 함수이다. 먼저 argv에 담겨 있는 chracter를 하나씩 esp를 감소시키면서 stack에 넣는다. argv를 다 담은 후에 argv들의 총 length를 보고 4의 배수가 아니라면 계산에 용이하기 위해 word alignment를 집어넣는다. 그 다음엔 각 argv들의 주소값, argv들의 갯수, retrun address(fake)를 차례대로 적재한다.

1. User Memory Access
   1. Page fault

/\* check invalid memory access \*/

if (!user || is\_kernel\_vaddr (fault\_addr) || not\_present) {

exit(-1);

}

pintos/src/?/ 에 주어진 page\_fault 함수를 수정했다. is\_kernel\_vaddr 함수를 이용해 user가 kernel memory 영역에 접근할 경우 exit(-1)을 호출해 page fault를 방지한다. is\_kernel\_vaddr함수는 0xc0000000를 기준으로 현재 주소값이 kernel memory 영역인지 판단하는 함수이다.

* 1. System call handler

switch(syscall\_num){

case SYS\_HALT:

halt();

break;

case SYS\_EXIT:

check\_user\_vaddr(f->esp + 4);

exit((int)\*(uint32\_t \*)(f->esp + 4));

break;

void check\_user\_vaddr(const void \*vaddr) {

if(!vaddr)

exit(-1);

if(!is\_user\_vaddr(vaddr))

exit(-1);

}

check\_user\_vaddr이라는 함수를 새로 구현했다. 이 함수는 현재 주소값이 유효한지, is\_user\_vaddr 함수를 이용해 현재 주소값이 user memory 영역인지 판단하고 invalid 하다면 exit(-1)을 호출해 종료하는 함수이다. system call handler에서는 이 함수를 이용해 system call을 할 때마다 인자로 넘어온 주소값이 user memory 영역인지 확인한다.

* 1. Check executable file

char cmd\_name[256]; // executable file name

int i;

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

for(i=0 ; cmd\_name[i]!='\0' && cmd\_name[i] != ' ' ; i++);

cmd\_name[i] = '\0';

if(!filesys\_open(cmd\_name))

return -1;

pintos/src/threads/process.c process\_execute 함수를 수정했다. thread를 생성하기 전에 주어진 file name에서 executable file이 open 가능한지 미리 확인하고 불가능하다면 thread를 생성시키지 않고 process\_execute 함수를 종료시킨다.

1. System Calls

* **이번 프로젝트에서 개발한 시스템 콜을 구현 관점에서 상세히 서술.**
  1. System call handler

static void

syscall\_handler (struct intr\_frame \*f UNUSED)

{

uint32\_t syscall\_num = \*(uint32\_t \*)f->esp;

switch(syscall\_num){

case SYS\_HALT:

halt();

break;

case SYS\_EXIT:

check\_user\_vaddr(f->esp + 4);

exit((int)\*(uint32\_t \*)(f->esp + 4));

break;

case SYS\_EXEC:

check\_user\_vaddr(f->esp + 4);

f->eax=(uint32\_t)exec((const char\*)\*(uint32\_t\*)(f->esp+ 4));

break;

case SYS\_WAIT:

check\_user\_vaddr(f->esp + 4);

f->eax = wait((uint32\_t)\*(uint32\_t \*)(f->esp + 4));

break;

case SYS\_READ:

check\_user\_vaddr(f->esp + 4);

f->eax = read((int)\*(uint32\_t \*)(f->esp + 4), (void \*)\*(uint32\_t \*)(f->esp + 8), (unsigned)\*((uint32\_t \*)(f->esp + 12)));

break;

case SYS\_WRITE:

check\_user\_vaddr(f->esp + 4);

f->eax = write((int)\*(uint32\_t \*)(f->esp + 4), (void \*)\*(uint32\_t \*)(f->esp + 8), (unsigned)\*((uint32\_t \*)(f->esp + 12)));

break;

case SYS\_FIBO:

check\_user\_vaddr(f->esp + 4);

f->eax = fibonacci((int)\*(uint32\_t \*)(f->esp + 4));

break;

case SYS\_MAX:

check\_user\_vaddr(f->esp + 4);

f->eax = max\_of\_four\_int((int)\*(uint32\_t \*)(f->esp + 4), (int)\*(uint32\_t \*)(f->esp + 8), (int)\*(uint32\_t \*)(f->esp + 12), (int)\*(uint32\_t \*)(f->esp + 16));

break;

}

}

system call handler의 인자로 넘어온 intr\_frame \*f의 esp를 이용해 system call number, arguments 들을 얻을 수 있다. switch – case 문법을 이용해 system call number에 따라 적절한 system call을 호출할 수 있게 구현했다. 시스템 콜을 확인하기 전에 invalid memory access를 항상 확인한다. sysem call을 호출할 때 인자들이 몇 개인지, 어떤 형 변환을 해야하는지 pintOS 매뉴얼에 따라 적절하게 판단한다.

* 1. halt

void halt(){

/\* shutdown PintOS \*/

shutdown\_power\_off();

}

pintos/src/devices/shutdown.h 에서 주어진 shutdown\_power\_off 함수를 이용해 pintOS를 강제종료하는 시스템 콜이다.

* 1. exit

void exit(int status){

/\* get running thread struct \*/

struct thread\* cur = thread\_current();

/\* store child exit status \*/

cur->child\_exit\_status = status;

/\* print process termination message \*/

printf("%s: exit(%d)\n", cur->name, status);

/\* thread exit \*/

thread\_exit (); // 내부에서 process\_exit() 호출

}

void

process\_exit (void)

{

struct thread \*cur = thread\_current ();

uint32\_t \*pd;

pd = cur->pagedir;

if (pd != NULL)

{

cur->pagedir = NULL;

pagedir\_activate (NULL);

pagedir\_destroy (pd);

}

/\* talk child process exit to parent \*/

sema\_up(&(cur->child\_lock));

/\* but wait reaping \*/

sema\_down(&(cur->mem\_lock));

}

현재 process 를 exit시키는 것 뿐만 아니라 적절한 process termination messages를 출력해야 하는 시스템 콜이다. 종료 status는 exit 시스템 콜을 호출할 때 인자로 넘어오기에 알 수 있지만 어떤 process(thread)가 종료되는지, process의 이름이 무엇인지 알기 위해 pintos/src/threads/thread.h 에 있는 thread\_current 함수를 이용했다. 적절한 termination message를 출력한 뒤에 thread\_exit 함수를 호출하는데 이 함수는 내부에서 process\_exit 함수를 호출한다. 여기서 child의 종료를 담당하는 semaphore를 이용해 child process가 종료되었으니 process\_wait에서 기다리고 있던 parent process가 진행되면서 child process를 reaping하게 된다.

* 1. exec

pid\_t exec (const char \*cmd\_line)

{

return process\_execute(cmd\_line);

}

주어진 command line을 이용해 process를 생성하고 execute하는 시스템 콜이다. process\_execute 함수에서 start\_process 함수를 호출하고 여기서 load 함수를 이용해 memory에 정보들을 적재하는 순서로 process를 실행시킨다. 실행된 process id가 반환된다.

* 1. wait

int wait (pid\_t pid)

{

return process\_wait ((tid\_t) pid);

}

주어진 process id를 가진 process가 종료되기를 기다리는 시스템 콜이다. process\_wait 함수를 호출하는데 여기서 child list들을 탐색하며 인자로 들어온 thread id를 가진 thread를 찾는다. 이후에 semaphore를 이용해 child process가 종료되기를 기다릴 수 있다. 이를 구현하기 위해 pintos/src/threas/thread.h 의 thread 구조체에 semaphore , struct list 등을 추가했다. child process의 exit status가 반환된다

.

int

process\_wait (tid\_t child\_tid UNUSED)

{

struct thread\* t;

int exit\_status = -1;

for(struct list\_elem\* e = list\_begin(&(thread\_current()->child\_list));

e != list\_end(&(thread\_current()->child\_list)); e = list\_next(e))

{

t = list\_entry(e, struct thread, child\_elem);

if(child\_tid == t->tid){

/\* wait for child process(loading) exit\*/

sema\_down(&(t->child\_lock));

/\* get child process exit status \*/

exit\_status = t->child\_exit\_status;

list\_remove(&(t->child\_elem));

/\* parent do reaping, so stop waiting \*/

sema\_up(&(t->mem\_lock));

return exit\_status;

}

}

return exit\_status;

}

struct thread

{

/\* Owned by thread.c. \*/

tid\_t tid; /\* Thread identifier. \*/

enum thread\_status status; /\* Thread state. \*/

char name[16]; /\* Name \*/

uint8\_t \*stack; /\* Saved stack pointer. \*/

int priority; /\* Priority. \*/

struct list\_elem allelem;

/\* Shared between thread.c and synch.c. \*/

struct list\_elem elem; /\* List element. \*/

#ifdef USERPROG

/\* Owned by userprog/process.c. \*/

uint32\_t \*pagedir; /\* Page directory. \*/

/\* child running semaphore \*/

struct semaphore child\_lock;

/\* memory(reaping) semaphore \*/

struct semaphore mem\_lock;

/\* child list \*/

struct list child\_list;

/\* child list element \*/

struct list\_elem child\_elem;

/\* child exit status \*/

int child\_exit\_status;

#endif

/\* Owned by thread.c. \*/

unsigned magic; /\* Detects stack overflow. \*/

};

* 1. write

int write (int fd, const void \*buffer, unsigned size) {

if (fd == STDOUT\_FILENO) {

putbuf((char \*)buffer, size);

return size;

}

return -1;

}

인자로 주어진 file descriptor에 buffer의 내용을 작성하는 시스템 콜이나 프로젝트 1에서는 최대한 간소화되었다. file descriptor에 stdout만 들어오며 console에 buffer의 내용을 출력한다. pintos/src/lib/kernel/stdio.h 의 putbuf 함수를 이용했다. 출력이 있을 경우 출력의 갯수를 반환했다.

* 1. read

int read(int fd, const void \*buffer, unsigned size){

if(fd == STDIN\_FILENO){

int i;

for(i = 0 ; i < (int)size ; i++){

((char \*)buffer)[i] = input\_getc();

if(((char \*)buffer)[i] == '\0')

break;

}

if(i != (int)size)

return -1;

else

return size;

}

return -1;

}

인자로 주어진 file descriptor에서 내용을 읽어와 buffer에 저장하는 시스템 콜이나 프로젝트 1에서는 write 와 같이 간소화되었다. file descriptor로 stdin만 들어오며 pintos/src/devices/input.h 의 input\_getc 함수를 이용해 입력을 받아들였다. 입력이 있을 경우 받은 입력의 갯수를 반환했다.

1. Additional System calls

* **새로운 시스템 콜(fibonacci, max\_of\_four\_int)을 구현하기 위해 수정하거나 작성한 코드에 대해 서술**
  1. pintos/src/lib/syscall-hr.h

/\* Project 2 additional \*/

SYS\_FIBO,

SYS\_MAX

새로운 system call을 구현하기 위해 가장 먼저 system call number를 등록했다.

* 1. pintos/src/lib/user/syscall.h

/\* Project 2 additional \*/

int fibonacci(int n);

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

user program에서 system call을 사용할 수 있도록 system call을 선언했다.

* 1. pintos/src/lib/user/syscall.c

#define syscall4(NUMBER, ARG0, ARG1, ARG2, ARG3)

({ \

int retval; \

asm volatile \

("pushl %[arg3]; pushl %[arg2]; pushl %[arg1]; pushl %[arg0]; " \

"pushl %[number]; int $0x30; addl $20, %%esp" \

: "=a" (retval) \

: [number] "i" (NUMBER), \

[arg0] "r" (ARG0), \

[arg1] "r" (ARG1), \

[arg2] "r" (ARG2), \

[arg3] "r" (ARG3) \

: "memory"); \

retval; \

})

기존의 pintOS에는 3개 이하의 인자만을 가진 system call만을 호출할 수 있었기에 additional system call을 구현하기 위해선 4개의 인자를 가진 system call을 호출할 수 있어야 하기에 새로이 구현했다.

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\_MAX, a, b, c, d);

}

새로이 구현한 sycall4를 이용해 Fibonacci, max\_of\_four\_int system call을 호출할 수 있게 되었다.

* 1. pintos/src/userprog/syscall.c

int fibonacci(int n){

if(n < 0){

return -1;

}

int \*fibo = (int\*)malloc(sizeof(int)\*n);

fibo[0] = 0; fibo[1] = 1;

for(int i = 2 ; i <= n ; i++){

fibo[i] = fibo[i-1] + fibo[i-2];

}

return fibo[n];

}

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

int max = b;

if(max < a)

max = b;

if(max < c)

max = c;

if(max < d)

max = d;

return max;

}

Fibonacci system call은 Fibonacci 수열의 n번째 수를 반환하고 max\_of\_four\_int system call은 들어온 4개의 인자 중 최대값을 반환한다.

* 1. pintos/src/examples/additional.c 생성

int

main (int argc, char \*argv[])

{

if(argc != 5){

return EXIT\_FAILURE;

}

printf("%d ",fibonacci(atoi(argv[1])));

printf("%d\n",max\_of\_four\_int(atoi(argv[1]), atoi(argv[2]), atoi(argv[3]), atoi(argv[4])));

return EXIT\_SUCCESS;

}

새로운 system call의 test를 위해 새로운 test file을 생성했다. 새로 구현한 system call을 호출하고 반환값을 출력한다.

* 1. pintos/src/examples/Makefile

PROGS = cat cmp cp echo halt hex-dump ls mcat mcp mkdir pwd rm shell bubsort lineup matmult recursor additional

# Should work from project 1 additional

additional\_SRC = additional.c

새로 만든 additional.c 함수를 컴파일해서 실행 파일로 만들기 위해 Makefile을 수정했다.

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

****

**텍스트이(가) 표시된 사진

자동 생성된 설명**