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

담당 교수 : 박성용

조 / 조원 : 20181687 정지석

개발 기간 : 2022-09-20 ~ 2022-10-03

1. **개발 목표**

Pintos에서 입력 받은 argument 들을 stack에 넘겨준 뒤, 해당 argument에 해당하는system call을 수행하는 기능을 구현한다. 이번 프로젝트에서는 halt, exit, exec, wait, read, write system call을 구현하고, 추가적으로 Fibonacci와 max\_of\_four\_int system call 또한 구현해주는 것이 목표이다.

1. **개발 범위 및 내용**
   1. **개발 범위**
2. Argument Passing

Argument가 stack에 입력되어 원하는 system call을 실행할 때 사용한다.

1. User Memory Access

올바른 memory만 access가 가능하도록 해준다. 올바른 memory가 아닐 경우, exit()을 실행하여 해당 thread를 종료한다.

1. System Calls

직접 구현한 system call handler에서 각 system call에 맞는 작업을 수행한다.

Parent/child thread 간에 실행 순서가 정해진다.

* 1. **개발 내용**
* Argument Passing
  + 커널 내 스택에 argument를 쌓는 과정 설명

사용자가 pintos에 여러 개의 argument를 입력할 경우, 이를 모두 구분하여 memory에 입력해주어야 한다. 입력 받은 argument를 strtok\_r() 함수를 사용하여 띄어쓰기를 기준으로 각 argument 들을 나눠준다. 그 다음 tokenize된 각 argument들과 그 주소 값들을 esp를 사용하여 차례대로 stack 안에 넣어주면 된다.

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

Pintos 상에서의 invalid memory access란 NULL pointer에 access하는 것, allocate 되지 않은 virtual memory space에 access하는 것, 혹은 kernel memory에 access 하는 것을 의미한다.

* + Invalid memory access를 어떻게 막을 것인지 설명

NULL pointer access: NULL일 경우 exit()을 사용하여 종료

allocate 되지 않은 virtual memory space에 access: pagedir\_get\_page() 함수를 사용하여 access 방지

kernel memory에 access: is\_kernel\_vaddr() 함수를 사용하여 access 방지

* System Calls
  + 시스템 콜의 필요성에 대한 간략한 설명

User가 직접 memory에 access하여 read나 write 같은 작업을 수행할 경우, 안정성이나 보안성에 문제가 생길 수 있기 때문에 kernel 만이 실행할 수 있는 명령어들을 지정하였는데, 이렇게 할 경우 user는 kernel 내부의 필요한 api들을 사용하지 못하게 된다는 문제점이 발생하게 된다. 이런 문제를 해결하기 위해 user가 system call을 통해 필요한 명령을 요청하면, kernel이 이를 간접적으로 실행해준다.

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

SYS\_HALT: pintos를 종료하는 system call이다. shutdown\_power\_off() 함수를 사용하여 구현하였다.

SYS\_EXIT: 현재 thread를 종료하는 system call이다. Thread\_exit() 함수를 이용하여 구현하였다.

SYS\_EXEC: 새로운 thread를 만들어주는 system call이다. Process\_execute() 함수를 사용하여 구현하였다.

SYS\_WAIT: parent가 child 보다 먼저 종료되는 것을 방지해주는 system call이다. Process\_wait() 함수를 사용하여 구현하였다.

SYS\_READ: open 상태인 file을 읽는 system call이다. Input\_getc() 함수를 사용하여 구현하였다.

SYS\_WRITE: file에 data를 저장하는 system call이다. Putbuf() 함수를 사용하여 구현하였다.

SYS\_FIB: 추가적으로 구현한 Fibonacci() 함수를 실행하는 system call이다. 입력 받은 숫자에 대한 Fibonacci 값을 return 해준다.

SYS\_MOFI: 추가적으로 구현한 max\_of\_four\_int() 함수를 실행하는 system call이다. 4개의 숫자를 입력 받아 그 중 가장 큰 값을 return 해준다.

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

System call이 호출될 경우, /lib/user/syscall.c에 선언된 해당 system call 부분으로 이동하여 user stack에 입력 받은 인자를 채워 넣는다. 그 다음 int 30을 호출하여 interrupt vector table로 이동한 후, userprog/syscall.c로 넘어가서 직접 구현한 system call을 실행한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

09/20~09/23: argument passing 구현, stack 구현

09/24~09/26: 각 system call, system call handler 구현

09/27~09/29: memory access 구현, thread 구조 수정

09/30~10/02: semaphore 사용해 sync 조정, 추가 기능 구현

10/03: 최종 test 및 보고서 작성

* 1. **개발 방법**

**Userprog/syscall.c**

각 system call을 구현하기 위해 syscall\_handler()에 switch 문을 사용하여 각

case에 맞는 system call을 작성하였다.

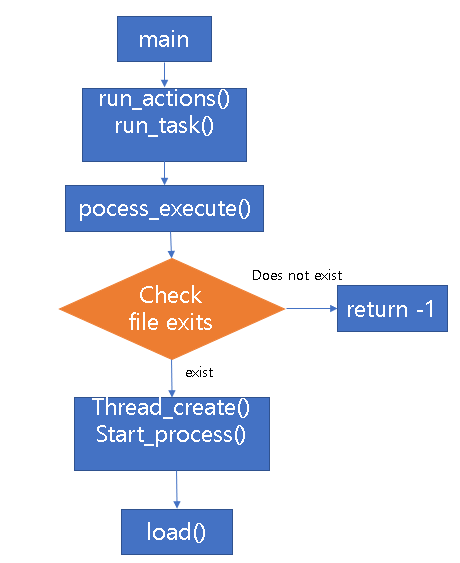
user memory access인지를 확인하기 위해 verfi\_acs() 라는 함수를 구현하였다. 이 함수는 is\_user\_vadder() 함수를 사용하여 user memory 인지 아닌지를 체크하는 역할을 한다.

**Userprog/process.c**

Process\_execute() 함수에서 file의 이름을 뽑아내고, file의 존재 유무를 확인한다. file이 존재하지 않으면 -1을 return하고, 존재할 경우 해당 file의 이름으로 thread를 생성한다.

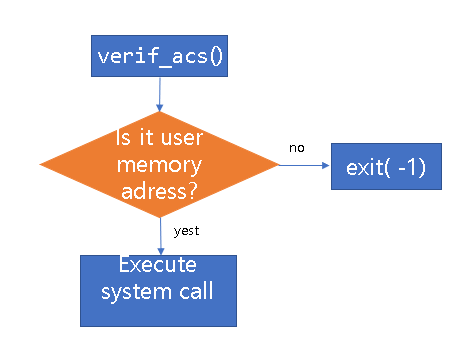
Argument passing을 구현하기 위해 load() 함수에서 입력받은 argument를 strtok\_r() 함수를 통해 parsing하고, 이를 stack에 집어넣는다.

1. **연구 결과**
   1. **Flow Chart**
2. Argument Passing



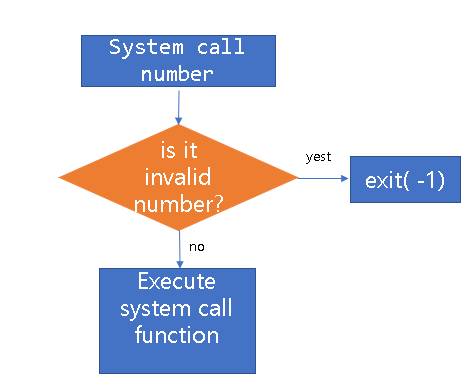
In **userprog/process.c**

1. User Memory Access



In **userprog/syscall.c**

1. System Calls



In **userprog/syscall.c**

* 1. **제작 내용**

1. Argument Passing

userprog/processs.c

  strlcpy(tmp, *file\_name*, strlen(*file\_name*)+1);

  ptr = strtok\_r(tmp, " ", &next\_ptr);

  strlcpy(cmd, ptr, strlen(ptr)+1);

*if* (filesys\_open(tmp) == NULL)

*return* -1;

*/\* Create a new thread to execute FILE\_NAME. \*/*

  tid = thread\_create (cmd, PRI\_DEFAULT, start\_process, fn\_copy);

*if* (tid == TID\_ERROR)

    palloc\_free\_page (fn\_copy);

*return* tid;

process\_execute()에서 file 이름을 확인해 해당 file이 존재하는지 확인한 후, 존재할 경우, 그 file의 이름으로 새로운 thread를 생성한다.

uint32\_t stk\_ptr[255];

int i = count - 1;

*while*(i >= 0)

  {

  (\**esp*) = (\**esp*) - (strlen(cur\_ptr[i]) + 1);

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

    memcpy(\**esp*, cur\_ptr[i], strlen(cur\_ptr[i]) + 1);

    i--;

  }

  (\**esp*) = (void \*)ROUND\_DOWN((uint32\_t)(\**esp*), 4);

(\**esp*) = (\**esp*) - sizeof(char \*);

  i = count - 1;

*while*(i >= 0)

  {

    \**esp* = \**esp* - sizeof(char \*);

    \*\*(uint32\_t \*\*)*esp* = (uint32\_t)&stk\_ptr[i];

    memcpy(\**esp*, &stk\_ptr[i], sizeof(char \*));

    i--;

  }

load() 함수에서는 넘겨받은 argument 자체와 그 주소를 저장하는 배열을 만들었다. 반복문을 돌며 마지막 명령어부터 첫번째 명령어까지 ‘명령어의 길이 +1’ 만큼 각각 esp pointer를 내리고, 그 주소에 명령어를 저장하는 과정을 반복하였다. 또한 이 반복문이 실행될 때 명령어가 저장된 위치도 stk\_ptr에 저장해준다. 반복문이 끝나면 esp를 현재 위치에서 가장 가까운 esp 아래의 4의 배수로 이동시킨다. 다시 while문을 실행하여 이전의 반복문에서 저장했던 명령어와 그 주소를 stack에 넣어준다.

1. User Memory Access

userprog/syscall.c

bool

verif\_acs(uint32\_t \**args*, int *argc*)

{

*for* (int i = 0; i < *argc*; i++)

*if* (!is\_user\_vaddr(*args* + i \* 4))

*return* false;

*return* true;

}

Syscall.c에서 직접 구현한 verfi\_acs라는 함수를 통해 system call을 수행할 때마다 참조하려는 위치가 user memory인지 여부를 확인한다. 만약 user memory 영역이 아니라면 exit(-1)을 실행하여 해당 thread를 종료한다.

1. System Calls

userprog/syscall.c

void halt(void){

  shutdown\_power\_off();

}

halt()의 경우 shutdown\_power\_off() 함수를 통해 halt가 실행될 경우 pintos가 종료되도록 구현하였다.

void

exit(int *stat*)

{

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

  cur->exit\_stat = *stat*;

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

*if* (cur->parent)

    cur->parent->exit\_stat = cur->exit\_stat;

  thread\_exit();

}

exit() system call이 실행될 경우 현재 실행 중인 thread를 종료하고 parent에게 종료 상태를 넘겨주는 작업을 수행한다. thread\_exit()을 통해 thread를 종료해준다.

wait(pid\_t *pid*)

{

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

}

process\_wait() 함수를 사용해서 child thread가 실행될 때까지 기다린다.

pid\_t

exec(const char \**file*)

{

*return* (pid\_t)process\_execute(*file*);

}

Process\_execute() 함수를 사용하여 child thread의 pid를 받아온다.

int

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

{

*if* (*fd* == 0)

*return* input\_getc();

*else*

*return* -1;

}

File descriptor가 0(read)일 경우, input\_getc() 함수를 사용하여 file의 data를 읽어오는 작업을 수행한다.

int

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

{

*if*(*fd* == 1){

    putbuf(*buffer*, *size*);

*return* true;

  }

*else*

*return* false;

}

File descriptor가 1(write)일 경우, putbuf() 함수를 사용하여 file에 data를 쓰는 작업을 수행한다.

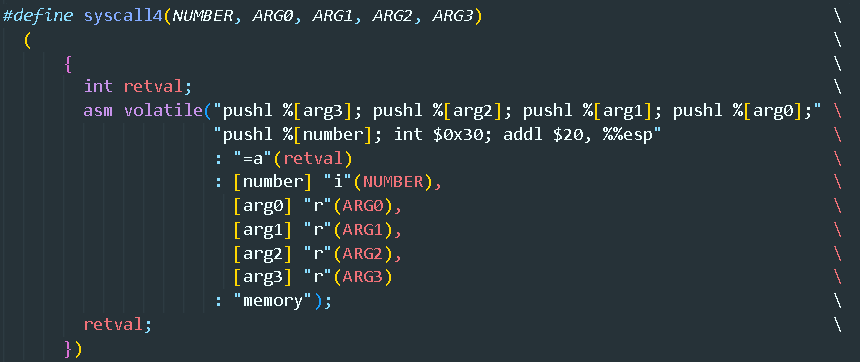
1. Additional System calls

lib/user/syscall.h

int fibonacci(int *n*);

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

추가적으로 구현할 system call을 header 파일에 추가한다.

lib/user/syscall.c

int

fibonacci(int *n*)

{

*return* syscall1(SYS\_FIB, *n*);

}

int

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

{

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

}

max\_of\_four\_int 함수의 경우 4개의 인자를 입력으로 받기 때문에 syscall4를 추가적으로 구현해주어야 한다.

lib/syscall-nr.h

    SYS\_FIB,

    SYS\_MOFI,

추가적으로 구현한 system call에 대해 system call number를 부여한다.

userprog/syscall.h

int fibonacci(int *n*);

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

userprog/syscall.c

int

fibonacci(int *n*)

{

*if*(*n* == 0)

*return* 0;

*else* *if*(*n* == 1 || *n* == 2)

*return* 1;

*else*

*return* fibonacci(*n*-1) + fibonacci(*n*-2);

}

하나의 정수를 받아 그에 대한 Fibonacci 값을 출력하는 함수를 작성하였다.

int

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

{

  int max = *a*;

*if* (max < *b*)

    max = *b*;

*if* (max < *c*)

    max = *c*;

*if* (max < *d*)

    max = *d*;

*return* max;

}

4개의 정수를 받아 그 중 가장 큰 값을 return하는 함수를 작성하였다.

examples/additional.c

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

{

    int input[4];

*if* (*argc* == 5)

    {

*for* (int i = 0; i < 4; i++)

            input[i] = atoi(*argv*[i + 1]);

        printf("%d %d\n", fibonacci(input[0]), max\_of\_four\_int(input[0], input[1], input[2], input[3]));

*return* EXIT\_SUCCESS;

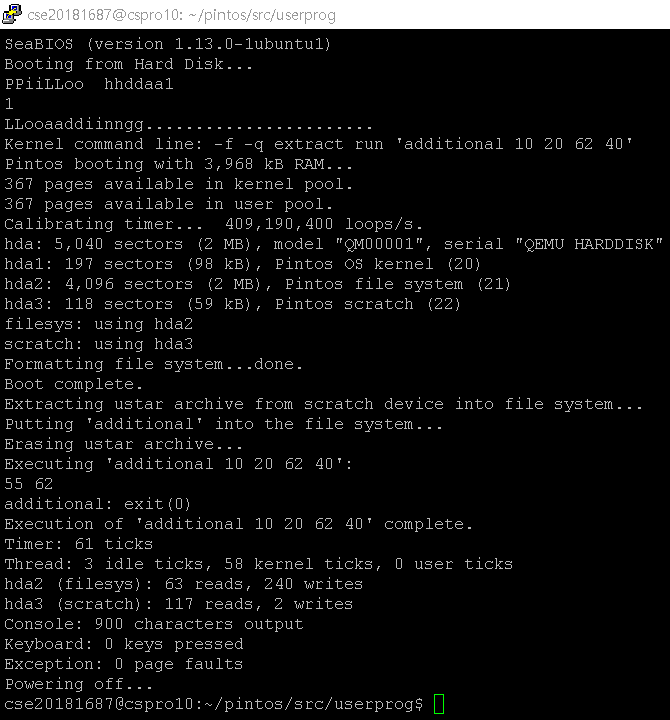
    }

*else*

*return* EXIT\_FAILURE;

}

구현한 system call이 제대로 작동하는지 확인하기 위해 추가적인 test 파일을 작성한다.

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