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

담당 교수 : 박성용 교수님

조 / 조원 : 20201635 전찬

개발 기간 : 2022.09.18 – 2022.10.02

1. **개발 목표**

이번 Pintos Project 1에서는 PintOS가 user program을 실행시킬 수 있도록 PintOS 상의 여러 기능을 구현해야 한다. 구현해야 하는 기능은 Argument Parsing와 Passing, System Call 이며, 이러한 기능에서 User Memory을 적절히 접근해야 한다. 마지막으로 fibonacci(), max\_of\_four\_int() 와 같이 추가적인 system call 또한 구현해야 한다.

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

Argument Passing 과정은 크게 Argument Parsing과 Argument Passing으로 나눌 수 있다. Argument Parsing 과정에서는 command line으로 입력받은 string에 대해 ‘ ’(공백 문자) 단위로 구분해서 나누어주어야 한다. Argument Passing 과정에서는 Parsing 과정으로 나눈 입력을 esp parameter을 사용해서 적절한 형태로 stack에 저장해 주어야 한다.

1. User Memory Access

PintOS 상에서 user program을 수행할 때, user program이 user memory 영역이 아닌, kernel memory 영역 등으로 비정상적인 접근을 수행할 수 있다. 이러한 문제를 방지하기 위해 user program이 user memory 영역으로 정상적으로 접근하는지를 확인해야 한다. 이를 확인할 수 있는 함수가 is\_user\_vaddr 이 있다. 이를 통해 어떠한 메모리 영역에 접근하기 이전에 해당 메모리 영역이 user memory 인지를 확인하며, user memory가 아니라면, exit(-1)으로 해당 프로그램을 비정상적으로 종료시켜야 한다. 또한 user 영역에서 비정상적인 접근이라면 바로 종료시키는 과정 또한 구현해야 한다.

1. System Calls

초기 PintOS 상태에서는 모든 system call이 구현되어 있지 않다. 따라서 halt, exit, exec, wait, read, write system call을 구현하며, 추가로 fibonacci, max\_of\_four\_int system call을 구현해야 한다.

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

Argument Parsing으로 구분한 입력에 대해, stack에 적절한 형태로 parsing한 입력을 쌓아주며 이를 system call에서의 parameter 등으로 활용할 수 있다. 이때 argument을 쌓는 calling convention은 아래와 같다.

1. Parsing을 수행한 data을 뒤에서부터 거꾸로 쌓는다.

2. WORD alignment을 맞추어 준다.

3. 구분을 위한 NULL 을 쌓는다.

4. Parsing을 수행한 data의 주소값(argv)을 거꾸로 각각 쌓아준다.

5. argv의 pointer을 쌓는다.

6. argc을 쌓는다.

7. return address인 NULL(0) 을 쌓는다.

위와 같은 형태로 입력 argument을 쌓을 수 있다. 예를 들어, 입력으로 “echo abc” 가 들어오는 경우, 각각은 아래와 같다.

1. “abc”, “echo” 을 esp에 쌓는다.

2. “abc” = 4개, “echo” = 5개의 char 이므로, word align을 위해 3개의 char을 쌓는다.

3. NULL을 쌓는다.(코드 상에서 argv[2]에 해당한다.)

4. “abc”, “echo” 의 pointer인 argv[1], argv[0] 쌓는다.

5. argv가 존재하는 \*esp + 4 을 쌓는다.

6. argc을 쌓는다.

7. return address 0 을 쌓는다.

* User Memory Access
  + Pintos 상에서의 invalid memory access 개념

PintOS 에서 user stack을 가리키는 esp는 0xc0000000 부터 아래로 쌓아가며 PHYS\_BASE 까지의 메모리 영역을 사용한다. 만약 user program이 이 이외의 영역에 접근한다면, 이는 비정상적인 접근으로, invalid memory access 라고 할 수 있다.

* + Invalid memory access를 막는 방법

따라서 이를 방지하기 위해, 두 가지 구현을 수행해야 한다. 첫 번째로, system call을 부를 때, esp을 통해 접근하는 system call의 parameter가 user memory 영역인지 확인해주어야 한다. 이를 위해 이미 정의되어 있는 함수인 is\_user\_vaddr() 을 사용하여 해당 영역이 user memory인지 아닌지를 파악할 수 있다. 두 번째로, userprog/exception.c 에 있는 page\_fault 함수를 보면, 현재는 page\_fault 를 발생하지만, 비정상적인 접근에 대해 User program을 바로 종료하지 않는다는 것을 알 수 있다. 따라서 이 또한 코드로 구현해주어야 한다.

* System Calls
  + 시스템 콜의 필요성

system call은 user program이 직접 수행할 수 없는 동작들에 대해(read, write 등) kernel에 있는

* + 이번 프로젝트에서 개발할 System Call

이번 프로젝트에서 개발할 system call은 아래와 같다.

halt : shutdown\_power\_off 함수를 통해 PintOS 자체를 종료한다.

exit : thread\_exit 함수를 통해 현재 process을 종료한다.

exec : process\_execute 함수를 통해 cmd\_line으로 들어온 입력을 실행한다.

wait : process\_wait 함수를 통해 child process을 기다리는 wait을 수행한다.

read : 이번 project에서는, fd = 0인 command line input을 처리할 수 있는 형태를 구현한다. input\_getc 함수를 이용하여 수행할 수 있다.

write : fd = 1인, STDOUT 으로의 출력을 구현한다. putbuf 함수를 통해 수행한다.

fibonacci : input n에 대한 Fibonacci number 을 구한다.

max\_of\_four\_int : 4개의 input 에 대한 최대값을 구하며, return 한다.

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

user level 에서 system call API을 호출하게 되면, system call, 그리고 각각의 argument가 esp stack에 쌓인다. 이렇게 쌓는 과정은 lib/user/syscall.c 에 syscall1, syscall2, syscall3 와 같이 구현되어 있다. 이후에 int $0x30으로 interrupt을 발생시키며, userprog/syscall.c에 구현되어 있는 syscall\_handler을 호출하게 된다. 이에 따라 syscall\_handler 은 현재 esp pointer에 저장되어 있는 값을 통해 어떠한 system call을 수행할 지 결정하고, esp + 4, esp + 8 등에 저장되어 있는 argument와 함께 해당 system call 함수를 실행시킨다. 또한 이러한 system call의 결과값은 f->eax 에 저장하여 이후에 활용할 수 있도록 한다.

1. **추진 일정 및 개발 방법**
   1. **추진 일정**

9.18 - 9.21 : PintOS 실행 과정 이해 및 개발 목표 이해

9.22 - 9.23 : Argument Passing 과정 구현

9.24 - 9.27 : System call 구현

9.28 - 9.29 : 추가적인 System call 구현 및 테스트를 통한 보완

9.30 - 10.02 : 구현 레포트 작성

* 1. **개발 방법**

1. Argument Passing

userprog/process.c 의 load 함수에 Parsing / Passing 하는 부분을 각각 적절한 위치에 추가한다. 이를 위해 나누어진 argument을 저장할 pstring, pdata, 그리고 주소를 저장할 argv 변수를 추가적으로 정의한다. 이를 활용해 esp에 calling convention을 바탕으로 변수, 주소들을 각각 저장한다.

2. User Memory Access

userprog/syscall.c 에서 check\_address 라는 함수를 새롭게 정의한다. 이 함수는 만약 해당 주소가 user address가 아니라면, exit(-1)을 통해 해당 프로그램을 종료시키는 함수이다. 이를 통해 user program의 kernel 영역에 대한 비정상적인 접근을 방지한다. 각각의 argument 개수에 따라 f->esp + 4, f->esp + 8, f->esp + 12 등의 주소에 대해 check\_address 함수를 적용시켜 User Memory Access 를 방지한다.

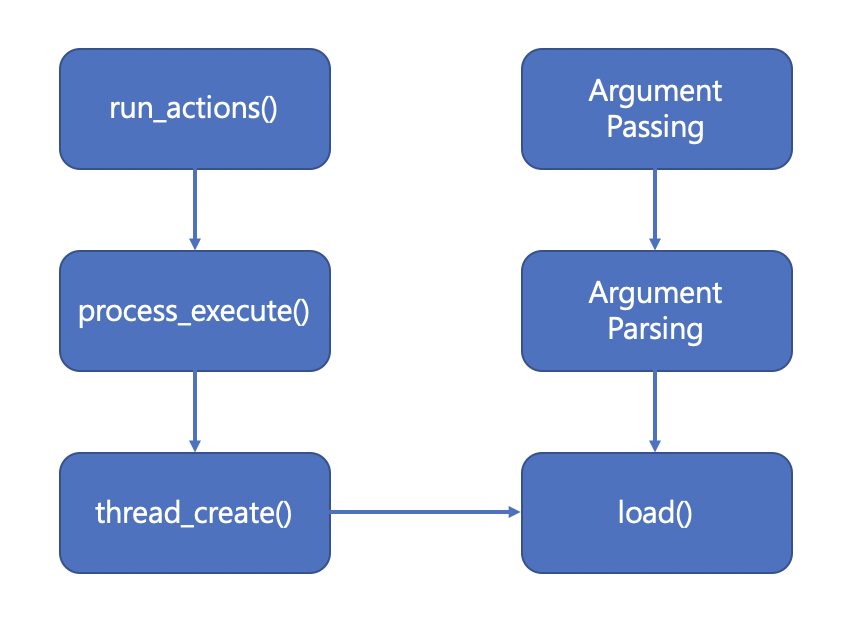
위에서 Argument에 대해, user memory 영역인지를 판단할 수 있다. 하지만 위 방법만으로 모든 문제를 방지할 수는 없다. 간단한 예로, User program 자체에서 Kernel 영역에 접근하는 형태가 존재할 수 있다. 이러한 비정상적인 접근 또한 방지해 주어야 하는데, 이를 방지할 수 있는 방법이 userprog/exception.c 의 page\_fault 함수에 추가적인 코드를 구현하는 것이다. 이를 통해 User program 자체에서의 비정상적인 접근을 방지해낼 수 있다.

3. System Calls

userprog/syscall.c에서 각각의 system call을 위에서 설명한 형식으로 구현한다. syscall\_handler 함수는 f->esp에 저장되어 있는 값을 통해 어떠한 system call을 수행할 지 파악해야 하며, 이는 switch 문을 통해 구현할 수 있다. 또한 wait process call에서 call하는, 초기에 return -1; 으로 구현되어 있는 process\_wait 함수를 구현하는 것 또한 중요하다. 이를 구현하기 위해 busy waiting 방법을 파악해야 하며, child process가 종료되었는지 파악할 수 있는 방법 또한 구현해야 한다.

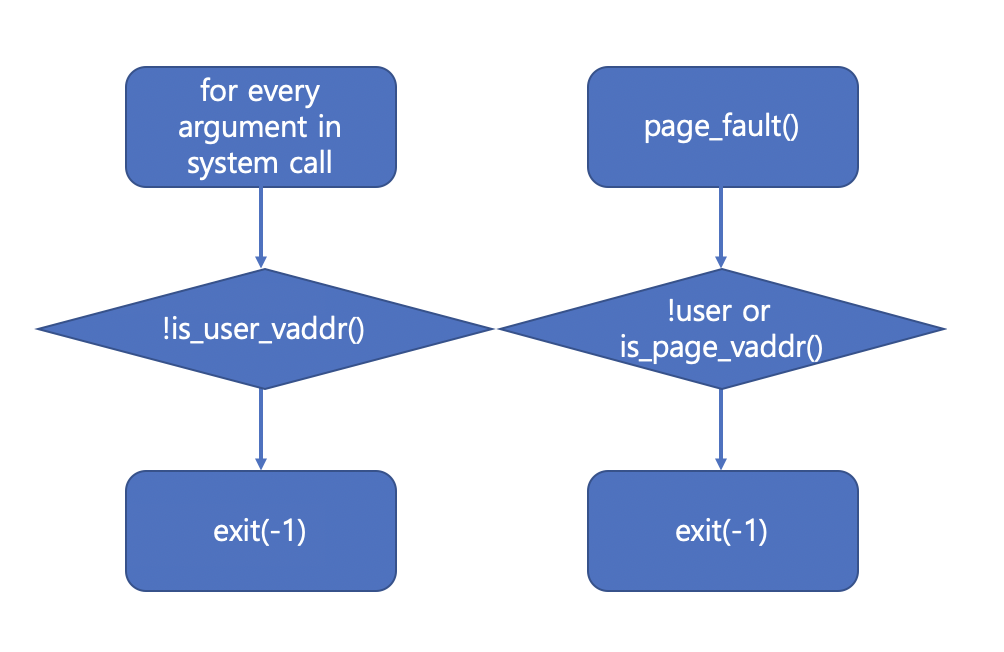
추가로 이후에 구현해야 할 max\_of\_four\_int system call을 위해 lib/user/syscall.c 에 구현되어 있는 syscall macro에 syscall4 macro을 동일한 형식으로 추가해야 한다. 또한 examples file에 있는 Makefile 또한 수정해 주어야 한다.

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



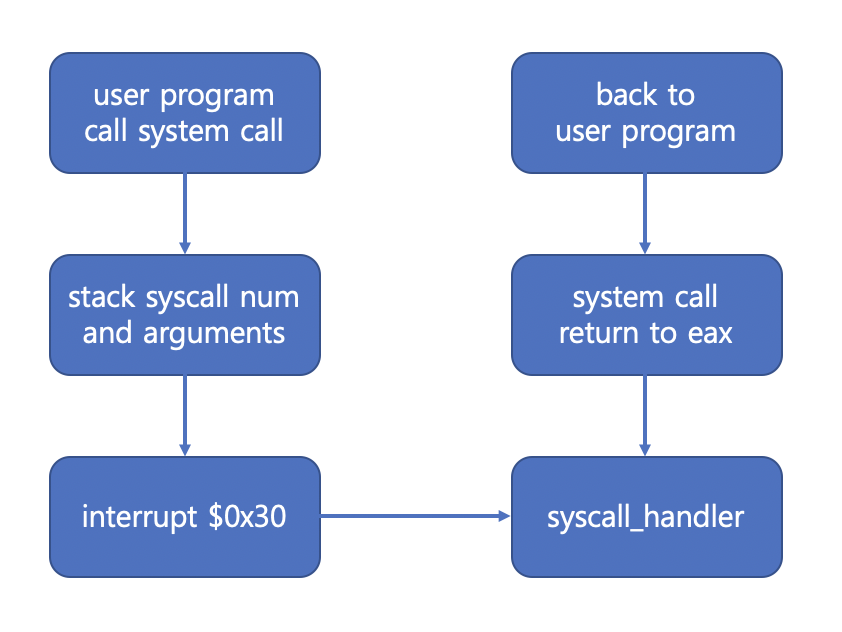
<Argument Passing flowchart>

1. User Memory Access



<User memory access flowchart>

1. System Calls



<system call flowchart>

* 1. **제작 내용**

1. Argument Passing

Argument passing 과정에서는, 크게 Parsing, Passing 두 가지로 분류하여 userprog/process.c의 load 함수 내에 코드를 작성했다. 우선 Parsing 과정은 아래와 같다.

/\* parse file name \*/

int file\_name\_len = strlen(file\_name); // file\_name 의 길이를 저장하는 변수

int argc = 0;

char\* file\_name\_save = (char\*)malloc(sizeof(char) \* (file\_name\_len + 1));

// saveptr은 strtok\_r에 사용, pstring은 strtok\_r의 결과값을 저장할 포인터이다.

char\* saveptr;

char\* pstring;

// parsed\_string을 각각 저장할 변수이다. (초기 size = 10으로 할당했다.)

char\* pdata[128];

strlcpy(file\_name\_save, file\_name, file\_name\_len + 1);

//printf("%s\n", file\_name\_save);

pstring = strtok\_r(file\_name\_save, " ", &saveptr);

pdata[0] = pstring;

//printf("%s\n", pdata[0]);

argc++;

while((pstring = strtok\_r(NULL, " ", &saveptr))){

pdata[argc] = pstring;

argc++;

}

<Argument Parsing>

file\_name에 대해, file\_name은 const char\* 형태이기 때문에 strlcpy로 file\_name\_save 변수에 copy 한다. 이를 strtok\_r 함수의 반복을 통해 “ ” 단위로 끊어내며, pdata[] 의 각각 원소에 pointer을 저장한다. 이러한 형태로 pdata 변수에 Argument Parsing된 file\_name을 저장할 수 있다.

이후 Argument Passing 과정은 아래와 같다.

/\* esp stack에 알맞는 형태로 data를 저장한다. 추가로 argv를 구현한다. \*/

int sum\_data\_len = 0;

char\*\* argv = (char\*\*)malloc(sizeof(char\*)\*(argc + 1)); // 마지막에 NULL을 할당하기 위해 + 1

//printf("%d\n", argc);

// parse 된 string을 각각 저장

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

int len = strlen(pdata[i]) + 1;

sum\_data\_len += len;

\*esp -= len;

strlcpy(\*esp, pdata[i], len); // esp에 저장

//print("%s\n", \*esp);

argv[i] = \*esp; // argv[] 을 구현

}

// 마지막 원소에 NULL 대입

argv[argc] = NULL;

//printf("%p\n", \*esp);

// word alignment 구현

for(int i = 0; i < 4-sum\_data\_len%4; i++){

\*esp -= 1;

}

//printf("%p\n", \*esp);

// argv[] 을 저장, 각각은 해당하는 string의 pointer 값

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

\*esp -= 4;

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

}

// argv의 위치를 저장

\*esp -= 4;

\*\*(char\*\*\*)esp = \*esp + 4;

// argc 저장

\*esp -= 4;

\*\*(int\*\*)esp = argc;

// return address 0 을 저장

\*esp -= 4;

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

<Argument Passing>

위와 같은 과정을 통해 calling convention에 적절한 형태로 esp에 각각의 data을 저장했다.

1. User Memory Access

User Memory Access 을 관리하기 위해 첫 번째로 userprog/syscall.c 내에 check\_address 함수를 새롭게 정의했으며, userprog/exception.c 의 page\_fault 함수 내에 추가적인 기능을 구현했다. 또한 각각의 파일에 is\_user\_vaddr / is\_kernel\_vaddr 함수를 사용하기 위해 헤더 파일으로 #include “threads/vaddr.h” 을 추가해주어야 한다. check\_address 함수는 아래와 같다.

#include "threads/vaddr.h"

// esp로 전달된 argument가 타당한 위치에 있는지 파악하는 함수이다.

void check\_address(void\* addr){

if(!is\_user\_vaddr(addr)){

exit(-1);

}

}

<각 system call Argument을 판단할 check\_address 함수>

이를 통해 아래와 같이 어떠한 system call을 수행할 때, 해당 system call의 Argument 수에 따라 check\_address을 수행하며 User Memory Access을 보장할 수 있다. 아래는 SYS\_READ인 경우의 예시이다.

case SYS\_READ:

check\_address(f->esp + 4);

check\_address(f->esp + 8);

check\_address(f->esp + 12);

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

break;

<check\_address 사용 예시>

또한 위에서 설명한 것처럼 exception.c 에서 page\_fault 함수 또한 수정해야 한다. 아래는 Determine cause 부분에서 user 값을 계산한 이후에 추가되는 부분이다.

// sc-bad-sp 을 방지하기 위해

if(!user || is\_kernel\_vaddr(fault\_addr)){

exit(-1);

}

<page\_fault 함수에서 추가되는 부분>

이를 통해 실제 테스트에서 sc-bad-sp 부분을 PASS 할 수 있으며, User Memory Access을 보장할 수 있다.

1. System Calls

이번 프로젝트에서는 userprog/syscall.c에서 각각의 system call을 구현하며, 각 system call 함수에 적절한 형태로 argument passing 하는 것이 가장 중요하다. 먼저 구현한 각각의 system call은 아래와 같다.

// pintos 자체를 종료하는 함수

void halt(void)

{

shutdown\_power\_off();

}

<halt system call>

halt system call은 shutdown\_power\_off 함수를 이용해 간단하게 수행할 수 있다.

// 현재 user program을 종료

void exit(int status)

{

// int status를 thread sturuct에 할당한다.

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

//cur->status = THREAD\_DYING;

cur->exit\_status = status;

//process\_exit();

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

thread\_exit();

//printf("EXIT func executed.\n");

}

<exit system call>

exit system call은 thread\_current() 로 현재 running process을 파악하고, 해당 process을 종료하는 형식으로 수행할 수 있다. 또한 exit status을 저장하기 위해 struct thread에 exit\_status을 추가했는데, 이는 아래에서 자세하게 설명할 예정이다.

// cmd\_line을 실행시키는 함수

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

{

// cmd\_line 기준 file name이 잘못된 경우??

return process\_execute(cmd\_line);

}

<exec system call>

exec system call은 간단하게 process\_execute로 수행할 수 있다.

int wait(tid\_t pid)

{

// child thread ID check

// exit status 받기

return process\_wait(pid);

}

<wait system call>

wait system call 또한 간단하게 process\_wait으로 구현할 수 있다.

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

{

// standard input이 아닌 경우

if(fd != 0){

return -1;

}

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

if(input\_getc() == '\0'){

break;

}

}

}

<read system call>

read system call은 fd = 0 인 경우에 대해, input\_getc 함수를 반복하는 형태로 구현했다.

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

{

// standard output이 아닌 경우

if(fd != 1){

return -1;

}

putbuf(buffer, size);

return size;

}

<write system call>

write system call 또한 fd = 1 인 경우에 대해 putbuf 함수로 간단하게 구현할 수 있다.

syscall\_handler 함수에서는 설명 비디오 영상에서 설명해주신 것과 같이 case 문을 사용해서 각각의 경우에 대해 Argument을 check\_address하고, 해당 system call을 실행하는 형식으로 구현했다.

static void

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

{

//printf("current system call num : %d\n", \*(uint32\_t \*)(f->esp));

//hex\_dump(f->esp, f->esp, 100, 1);

switch (\*(uint32\_t \*)(f->esp)) {

case SYS\_HALT:

halt();

break;

case SYS\_EXIT:

check\_address(f->esp + 4);

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

break;

case SYS\_EXEC:

check\_address(f->esp + 4);

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

break;

case SYS\_WAIT:

check\_address(f->esp + 4);

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

break;

case SYS\_READ:

check\_address(f->esp + 4);

check\_address(f->esp + 8);

check\_address(f->esp + 12);

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

break;

case SYS\_WRITE:

check\_address(f->esp + 4);

check\_address(f->esp + 8);

check\_address(f->esp + 12);

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

//printf("%d %p %d\n", \*(uint32\_t \*)(f->esp + 4), \*(uint32\_t \*)(f->esp + 8), \*(uint32\_t \*)(f->esp + 12));

break;

}

<system call handler 함수>

또한 wait system call을 구현하기 위해 userprog/process.c의 process\_wait 함수 또한 구현해야 한다. 강의 자료에 나와 있는 것처럼, 처음에는 busy waiting 형식으로 구현을 수행했다. 이는 아래와 같다.

while(1){

//printf("loop executed\n");

child = NULL;

// 전체 thread list에 존재하지 않는다면, 종료된 것이며, 이 경우 return을 수행한다.

for(e = list\_begin(&(cur->child)); e != list\_end(&cur->child); e = list\_next(e)){

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

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

child = check;

break;

}

}

// allelem에 존재하지 않는다면, 종료된 것으로 busy waiting loop을 break 한다.

if(child == NULL){

break;

}

else{

thread\_yield();

}

}

<busy waiting 방식으로의 구현>

기본적인 아이디어는 busy wait으로, loop을 돌면서, 현재 list에 입력받은 child\_tid가 존재하는지를 확인하고, 아직 존재한다면 thread\_yield()로 현재 process의 수행을 미루는 형식으로 구현했다.

하지만 위와 같이 구현했을 때, “echo x” 와 같은 기본적인 PintOS 실행은 가능했으나, wait에 관련한 모든 test FAIL이 되었다. 따라서 synchonization을 바탕으로 한 wait을 구현했다. 이는 아래와 같다.

#include "threads/synch.h"

process\_wait (tid\_t child\_tid)

{

int exit\_status;

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

struct thread\* child = NULL;

struct list\_elem\* e;

for(e = list\_begin(&(cur->child)); e != list\_end(&(cur->child)); e = list\_next(e)){

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

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

child = check;

//exit\_status = child->exit\_status;

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

exit\_status = child->exit\_status;

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

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

return exit\_status;

}

}

}

<구현한 process\_wait 함수>

위 구현에 대해서 설명하자면, 우선 parent process가 child process보다 먼저 종료되면 안된다는 기본적인 개념을 바탕이 된다. 따라서 parent process가 child process가 process\_exit이 될 때까지 기다려 주어야 하는데, 이를 구현할 수 있는 방법이 현재 실행중인 thread인 cur 변수에 대해, child list을 검색해가며 해당하는 child process을 찾아내고, 해당 child process가 종료된 이후에 wait을 수행하고, child list에서 해당 child을 제거하고, 다시 process\_exit으로 돌아가는 것이다. 이를 정리하면 아래와 같다.

1. child process 가 종료되어 process\_exit 함수를 실행한다.

2. process\_exit 함수에서 child\_lock 에 대해 sema\_up을 수행한다.

3. process\_wait 함수에서 child\_lock의 sema\_down을 수행하고, child list에서 제거한다.

4. process\_wait 함수에서 mem\_lock의 sema\_up을 수행한다.

5. process\_exit 함수에서 mem\_lock을 sema\_down 하고 process\_exit 또한 종료한다.

이를 통해 child process가 완전하게 종료되기 전에 process\_wait 을 수행하는 형식을 구현할 수 있다. 또한 이를 위해 process\_exit에도 맨 마지막에 semaphore 관련 코드를 추가해주어야 한다. 추가로 process\_exit에서 exit print 문도 추가해 주어야 한다. 따라서 process\_exit 함수에서 추가해준 코드는 아래와 같다.

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

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

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

<process\_exit 함수의 추가 코드>

또한 위와 같이 semaphore을 사용해주기 위해, threads/thread.h, threads/thread.c 에 있는 thread 구조체 정의와 init\_thread 또한 semaphore, child list, exit status을 추가한 형식으로 다시 구현해줄 필요가 있다. 이는 아래와 같다.

struct thread

{

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

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

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

char name[16]; /\* Name (for debugging purposes). \*/

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

int priority; /\* Priority. \*/

struct list\_elem allelem; /\* List element for all threads list. \*/

/\* 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. \*/

int exit\_status;

struct list child;

struct list\_elem child\_elem;

struct semaphore child\_lock;

struct semaphore mem\_lock;

#endif

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

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

};

<thread.c의 struct thread>

#ifdef USERPROG

sema\_init(&(t->child\_lock), 0);

sema\_init(&(t->mem\_lock), 0);

list\_init(&(t->child));

list\_push\_back(&(running\_thread()->child), &(t->child\_elem));

#endif

<init\_thread 에 추가한 내용>

위와 같은 구현을 통해 wait system call이 정확하게 동작할 수 있도록 구현할 수 있다.

1. Additional System calls

마지막으로 Additional System call 을 구현하기 위해 총 7가지 작업을 수행해야 한다. 각각은 아래와 같다.

4-1. lib/syscall-nr.h의 enum 형식에 SYS\_FIBONACCI, SYS\_MAX\_OF\_FOUR\_INT 추가

/\* 추가적인 syscall인 fibonacci, max\_of\_four\_int 구현 \*/

SYS\_FIBONACCI,

SYS\_MAX\_OF\_FOUR\_INT

<enum의 마지막에 추가하는 형식>

4-2. lib/user/syscall.c에 syscall4 macro 구현

/\* syscall4 을 수행할 수 있는 macro을 설정한다.

이 marco는 max\_of\_four\_int 에서 사용될 수 있다. \*/

#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; \

})

<syscall4의 구현>

4-3. lib/user/syscall.c에 Fibonacci, max\_of\_four\_int 구현

/\* fibonacci, max\_or\_four\_int \*/

int fibonacci(int n)

{

return syscall1(SYS\_FIBONACCI, n);

}

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

{

return syscall4(SYS\_MAX\_OF\_FOUR\_INT, a, b, c, d);

}

<각각의 함수 구현>

4-4. userprog/syscall.c에 실제 함수 구현

int fibonacci(int n){

int n\_1 = 0;

int n\_2 = 0;

int result = 1;

for(int i=1; i<n; i++){

n\_2 = n\_1;

n\_1 = result;

result = n\_1 + n\_2;

}

return result;

}

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;

}

<system call 함수 실제 구현>

4-5. userprog/syscall.c의 syscall\_handler에 추가

case SYS\_FIBONACCI:

check\_address(f->esp + 4);

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

break;

case SYS\_MAX\_OF\_FOUR\_INT:

check\_address(f->esp + 4);

check\_address(f->esp + 8);

check\_address(f->esp + 12);

check\_address(f->esp + 16);

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

break;

<syscall\_handler에 추가되는 부분>

4-6. examples/additional.c 구현

#include <stdio.h>

#include <stdlib.h>

#include <syscall.h>

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

int input\_data[4];

if(argc != 5){

printf("additional needs 4 inputs but there are %d inputs.\n", argc-1);

return 1;

}

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

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

}

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

return 0;

}

<additional.c 구현>

4-7. examples/Makefile 수정

PROGS = cat cmp cp echo halt hex-dump ls mcat mcp mkdir pwd rm **shell** \

bubsort lineup matmult recursor additional

와 같이 수정해주어야 한다. 이를 통해 새로운 system call인 fibonacci, max\_of\_four\_int system call을 직접 만들어낼 수 있다.

* 1. **시험 및 평가 내용**
* **make check로 확인한 결과 (Test cases에 빨간색 표시)**

pass tests/userprog/args-none

pass tests/userprog/args-single

pass tests/userprog/args-multiple

pass tests/userprog/args-many

pass tests/userprog/args-dbl-space

pass tests/userprog/sc-bad-sp

pass tests/userprog/sc-bad-arg

pass tests/userprog/sc-boundary

pass tests/userprog/sc-boundary-2

pass tests/userprog/sc-boundary-3

pass tests/userprog/halt

pass tests/userprog/exit

pass tests/userprog/create-normal

FAIL tests/userprog/create-empty

FAIL tests/userprog/create-null

FAIL tests/userprog/create-bad-ptr

FAIL tests/userprog/create-long

FAIL tests/userprog/create-exists

pass tests/userprog/create-bound

pass tests/userprog/open-normal

FAIL tests/userprog/open-missing

pass tests/userprog/open-boundary

FAIL tests/userprog/open-empty

pass tests/userprog/open-null

FAIL tests/userprog/open-bad-ptr

pass tests/userprog/open-twice

pass tests/userprog/close-normal

pass tests/userprog/close-twice

pass tests/userprog/close-stdin

pass tests/userprog/close-stdout

pass tests/userprog/close-bad-fd

FAIL tests/userprog/read-normal

FAIL tests/userprog/read-bad-ptr

FAIL tests/userprog/read-boundary

FAIL tests/userprog/read-zero

pass tests/userprog/read-stdout

pass tests/userprog/read-bad-fd

FAIL tests/userprog/write-normal

FAIL tests/userprog/write-bad-ptr

FAIL tests/userprog/write-boundary

FAIL tests/userprog/write-zero

pass tests/userprog/write-stdin

pass tests/userprog/write-bad-fd

pass tests/userprog/exec-once

pass tests/userprog/exec-arg

pass tests/userprog/exec-bound

pass tests/userprog/exec-bound-2

pass tests/userprog/exec-bound-3

pass tests/userprog/exec-multiple

pass tests/userprog/exec-missing

pass tests/userprog/exec-bad-ptr

pass tests/userprog/wait-simple

pass tests/userprog/wait-twice

pass tests/userprog/wait-killed

pass tests/userprog/wait-bad-pid

pass tests/userprog/multi-recurse

FAIL tests/userprog/multi-child-fd

FAIL tests/userprog/rox-simple

FAIL tests/userprog/rox-child

FAIL tests/userprog/rox-multichild

FAIL tests/userprog/bad-read

FAIL tests/userprog/bad-write

pass tests/userprog/bad-read2

pass tests/userprog/bad-write2

FAIL tests/userprog/bad-jump

pass tests/userprog/bad-jump2

FAIL tests/userprog/no-vm/multi-oom

FAIL tests/filesys/base/lg-create

FAIL tests/filesys/base/lg-full

FAIL tests/filesys/base/lg-random

FAIL tests/filesys/base/lg-seq-block

FAIL tests/filesys/base/lg-seq-random

FAIL tests/filesys/base/sm-create

FAIL tests/filesys/base/sm-full

FAIL tests/filesys/base/sm-random

FAIL tests/filesys/base/sm-seq-block

FAIL tests/filesys/base/sm-seq-random

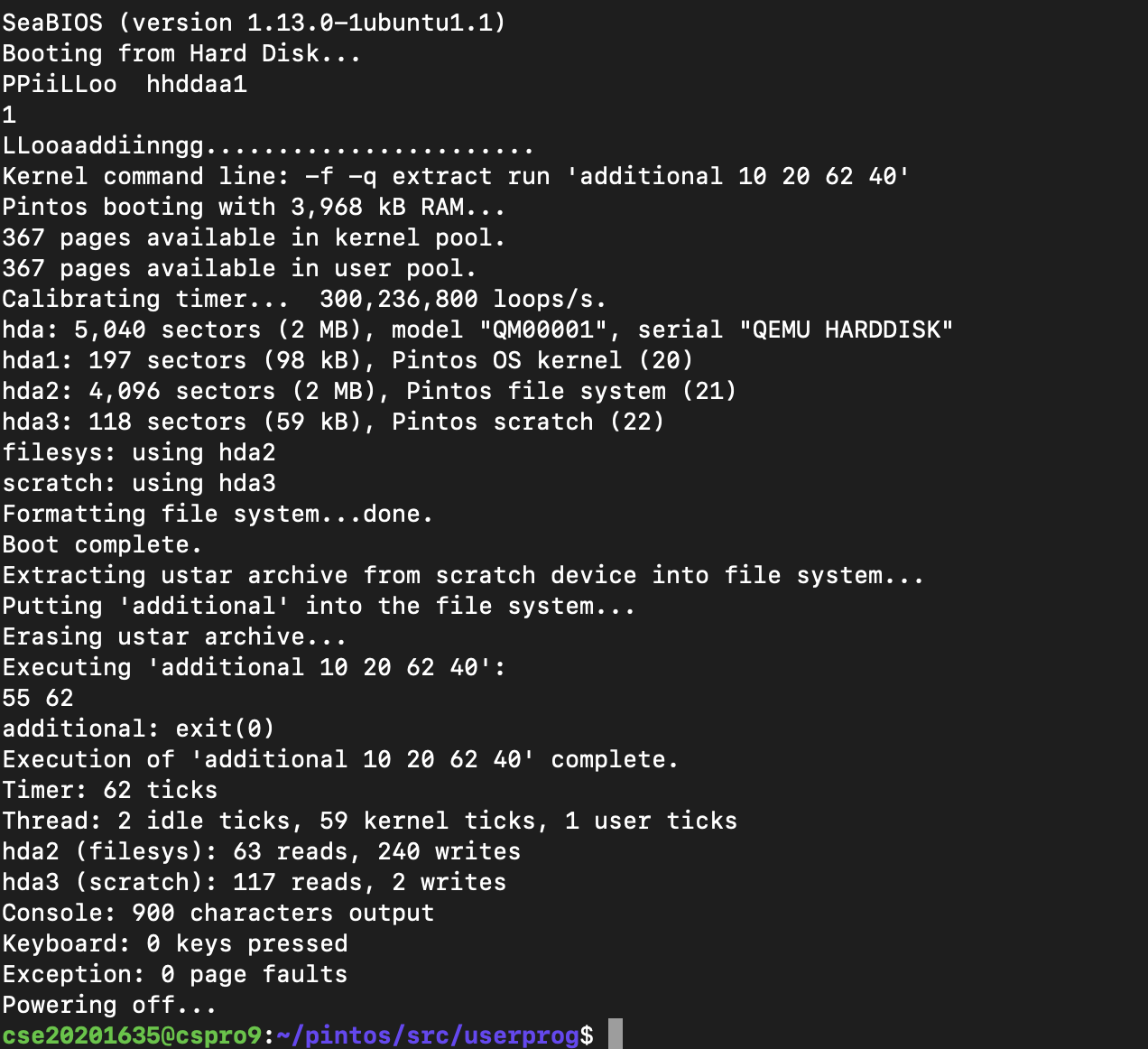
FAIL tests/filesys/base/syn-read

FAIL tests/filesys/base/syn-remove

FAIL tests/filesys/base/syn-write

37 of 80 tests failed.

* **fibonacci 및 max\_of\_four\_int 시스템 콜 수행 결과 캡처본**



<Fibonacci, max\_of\_four\_int system call 수행 결과>