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

**(설계 프로젝트 수행 결과)**

과목 명 : [CSE4070] 운영체제

담당 교수 : 김영재 교수님

조 / 조원 : 46조, 20141284, 20141340

개발 기간 : 2019/10/6 ~ 2019/11/3

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

**제출일 : 2019년 11월 3일**

**참여 조원 : 이기현, 장주호**

1. **개발목표**

* Pintos OS 의 환경과 기본적으로 구현되어 있는 부분을 어느 정도 숙지하고, Process와 thread의 개념을 이해하여 Pintos 환경에서 user program 이 작동할 수 있도록 한다. 따라서 기초 단계로 argument passing을 구현한다. 또한 기본적인 System call 인 halt, exit, read, exec, write, wait 등을 구현하고 Pintos 환경에서 user program이 작동할 수 있도록 환경을 구축한다.

1. **개발 범위 및 내용**
2. **개발범위**

/threads/ thread.h thread.c

/userprog/ syscall.h syscall.c process.h process.c exception.c

/example/ syscall-nr.h

/lib/ syscall.h syscall.c

/lib/user/ sum.c Makefile

* **Argument Passing**

입력받은 arguments를 tokenizing 하여 잘라서 80x86 Calling Convention에 맞추어 ESP(stack pointer)의 가장 밑 부분부터 차례대로 채운다. /userprog/process.c 에서 stack을 setup 한 뒤, tokenizing한 argument를 stack에 push 해준다.

* **User Memory Access**

유저가 잘못된 메모리 접근을 할 경우, 오류로 처리하여 종료시켜 주어야 한다.

1. User Virtual Memory 가 아닌 다른 장소를 access 할 때
2. return address 보다 위의 메모리에 접근할 때
3. 실제 사용할 수 있는 0x0804800 보다 아래의 메모리에 접근할 때

* **System Calls**

User program 내에서 system call 호출을 위한 system call handler의 구현과 halt, exit, exec, wait, read, write 의 기본적인 system call 함수를 구현한다. 자식 thread의 exit과 부모 thread의 wait의 싱크를 맞추기 위해 thread 구조체에 필요한 변수를 추가하고 이에 주의하여 system call 함수를 구현한다.

1. **개발 내용**

thread는 CPU의 기본 단위로 process가 작동하기 위한 공간이다. 각 thread는 모두 각자의 state와 stack을 갖고, 이를 page단위로 관리하고 page directory를 이용하여 physical memory에 연결된다. 또한 각 thread는 고유한 tid를 갖는다. 한 process에서 생긴 thread는 code와 address space 등을 공유한다.

각 page는 virtual memory로 연속적인 공간으로써 여겨진다 page table 과 page directory로 physical memory와 연결되며 총 4KB의 크기를 갖는다. 1KB는 kernel을 위해 존재한다.

user memory와 kernel memory는 서로 접근하면 안되는데 이를 피하기 위해서 user memory의 주소인지 kernel memory의 주소인지 체크하여 만약 user process가 kernel memory에 접근하게 되면 error를 발생시킬 수 있다.

부모 thread가 자식 thread의 수행여부를 알 수 없기 때문에 동기화작업을 해주어야 하는데 이 때, busy-waiting 기법과 semaphore 방식 등이 있다. 우리는 semaphore를 이용하여 자식 thread가 종료되지 않았다면 부모 thread를 block하여 종료되지 않게 해주고 자식 thread가 종료될 때, 부모 thread가 이를 인식할 수 잇게 한다.

1. **추진 일정 및 개발 방법**
2. **추진 일정**

10/6 (일) ~ 10/25 (금) : ppt 및 매뉴얼 분석

10/26 (토) ~ 10/27 (일) : Argument passing, halt, read, write 구현

10/28 (월) ~ 10/29 (화) : system handler, fibonacci, sum\_of\_four\_int 구현

10/30 (수) ~ 10/31 (목) : exit, exec, wait 구현

11/1 (금) ~ 11/3 (일) : 디버깅 및 보고서 작성

1. **개발 방법**

* **Argument passing**

stack에 exp를 이용하여 argument를 쌓는 makestack 함수를 먼저 구현한다. argument의 길이는 128bytes 이하로 가정한다. process.c 의 load 함수 내에서 각각의 argument를 parsing한 후, makestack 함수를 호출하여 인자들을 스택에 차곡차곡 쌓는다. 이후에 스택에 잘 쌓였는지 확인하기 위해 hex\_dump함수를 이용하여 결과를 확인한다.

* **User memory access**

Pintos 상에서 제공되는 스택 메모리에 접근할 때, 잘못된 접근으로 오류가 발생할 수 있으므로 잘못된 접근을 오류로 처리하여 종료시킨다. “threads/vaddr.h”에 들어있는 is\_kernel\_vaddr() 함수나 is\_user\_vaddr()함수를 사용하여 주소를 참조하는 SYSCALL\_HANDLER의 모든 부분에  
위의 함수를 사용하여 esp > PHYS\_BASE 인지 확인 후 아닐 시 종료를 시키는 방식으로 구현을 진행한다.

* **System call\_handler**

우선 /lib/syscall-nr.h에 각 system call을 enumeration 시켜놓았다. system call이 있을 때마다 스택포인터인 esp 포인터가 가리키는 값을 switch문을 이용하여 이전에 system call을 enueration한 값과 비교하여 맞는 system call을 호출해준다.

* **System call Inplementation**

halt : pintos 프로그램을 종료시키는 함수로, shutdown\_power\_off를 호출한다.

exec : 인자로 받은 command를 수행하는 함수로 /userprog/process.c 의 process\_execute 함수를 호출한다.

exit : 현재 thread를 종료하는 함수이다. 현재 thread의 parent thread가 기다리고 있기 때문에 끝남을 알려주고, 끝나는 status를 알려주어야 한다. 그렇기 때문에 parent thread를 찾아서 parent에게 status를 알려주고 process\_exit을 호출한다. process\_exit 내부에서 thread가 끝남을 semaphore를 통해 알려준다.

wait : child thread를 기다리기 위한 함수로 /userprog/process.c 의 process\_wait을 호출한다.

read : stdin을 통해 입력을 받기 때문에 file descriptor가 0이며 input\_getc를 for문을 통해 buffer에 한 글자 씩 추가한다. size만큼 읽지만 그 전에 file의 끝이 읽히면 종료한다. 실제로 읽은 개수를 리턴해준다.

write : stdout을 통해 출력하므로 file descriptor가 1이며 put\_buf를 통해 size만큼 buffer로부터 불러온다.

* **Additional Implementation**

/userprog/syscall.c 에 fibonacci와 sum\_of\_four\_int 의 system call 호출을 위한 함수를 먼저 정의한다.

실행파일을 만들기위해 /example/ 에 fibonacci와 sum\_of\_four\_int system call을 하는 파일 sum.c 를 구현하고, Makefile에 해당 파일을 추가해준다.

또한, system call handler에서 성공적으로 system call을 구분하여 호출해주기 위해 SYS\_FIBO와 SYS\_SUM을 /lib/syscall-nr.h 에 추가해주고, 4개의 인자를 받는 system call에 대해서 stack의 메모리를 처리하는 어셈블리코드를 /lib/user/syscall.c 에 매크로로 추가로 정의해주고 어셈블리매크로를 호출하는 fibonacci와 sum\_of\_four\_int 함수를 정의해준다.

1. **연구원 역할 분담**

이기현 – thread 구조 수정 및 process\_wait 구현

system call (exit, wait, read) 구현

Additional implementation 구현

장주호 – Argument passing 구현

user memory Access 구현

system call handler 구현

system call (halt, exec, write) 구현

1. **연구결과**
2. **합성 내용**

**스크린샷이(가) 표시된 사진

자동 생성된 설명**

1. **제작 내용**

**/userprog/process.c**

**void makestack (char \*filename, void \*\*esp)**

입력받은 argument들을 스택에 쌓는 과정으로 최초에 strtok\_r 함수를 이용해서 총 몇 개의 argument가 들어왔는지 filename을 tokenize 하면서 확인 후, argument의 개수만큼 임시 배열을 할당받는다. 그리고 나서 마지막 argument부터 순서에 맞게 stack에 쌓고, word allignment를 해준다. 그 후에는 NULL값과 각 argument의 주소를 stack에 쌓고 argument의 주소와 최후에 return address를 쌓아준다.

void makestack (char \*filename, void \*\*esp) {

char \*\*arg;

int inputlen = 0, len = 0, result = 0, i = 0;

char tmp[256];

char \*ptr, \*token;

inputlen = strlen(filename) + 1;

strcpy(tmp, filename, inputlen);

token = strtok\_r(filename, “ “, &ptr);

while (token != NULL) {

result++;

token = strtok\_r(NULL, “ “, &ptr);

}

arg = (char \*\*)malloc(sizeof(char \*) \* result);

token = strtok\_r(tmp, “ “, &ptr);

for(i = 0; i < result; i++) {

arg[i] = token;

token = strtok\_r(NULL, “ “, &ptr);

}

for (i = result – 1; i >= 0; i--) {

inputlen = strlen(arg[i]) + 1;

\*esp -= inputlen;

len += inputlen;

strlcpy(\*esp, arg[i], inputlen);

arg[i] = \*esp;

}

int wordallign = len % 4 == 0 ? 0 : 4 – (len % 4) ;

for(I = 0; I < wordallign; i++) {

\*esp -= 1;

\*\*(uint8\_t \*\*)esp = 0;

}

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

\*esp -= 4; \*\*(uint32\_t \*\*) esp = result;

\*esp -= 4; \*\*(uint32\_t \*\*) esp = 0;

free(arg);

}

**void parsing(char \*dest, char \*src)**

입력받은 argument를 parsing 해주는 함수로 process\_execute 함수 내에서 thread\_create로 인자를 전달할 때, 앞의 명령부분만을 전달하기 위해서 parsing 함수를 통해 입력받은 argument를 parsing 한다.

void parsing(char\* dest, char\* src){

int i=0;

strlcpy(dest, src, strlen(src)+1);

while(1){

if(dest[i] == ‘\0’ || dest[i] == ‘ ‘) break;

i++;

}

dest[i] = ‘\0’;

}

**tid\_t process\_execute (const char \* file\_name)**

process\_execute 함수는 입력받은 filename을 이름으로 하는 자식 thread를 생성한다. 기본 제공되는 함수에 입력받은 argument를 parsing 후, valid 한지 체크하기 위해 filesys\_open을 통해 체크하고 valid하지 않다면 -1을 return한다. 그 후, 입력받은 argument를 thread\_create에 인자로 전달한다.

thread\_create가 끝나면 새로 생성한 thread의 par\_tid로 지금 실행 중인 thread의 tid를 설정해준다.

tid\_t process\_execute (const char \*file\_name) {

char \*fn\_copy;

tid\_t tid;

char tmp[256];

parsing(tmp, file\_name);

/\* Make a copy of FILE\_NAME.

Otherwise there's a race between the caller and load(). \*/

fn\_copy = palloc\_get\_page (0);

if (fn\_copy == NULL)

return TID\_ERROR;

strlcpy (fn\_copy, file\_name, PGSIZE);

if(filesys\_open(tmp)==NULL) return -1;

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

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

if (tid == TID\_ERROR)

palloc\_free\_page (fn\_copy);

tid\_thread(tid)->par\_tid = thread\_current()->tid;

return tid;

}

**int process\_wait(tid\_t child\_tid)**

process\_wait 함수는 child thread 가 실행 중에 parent thread 가 종료되지 않게 하기 위한 함수이다. 그렇기 때문에 child가 수행 중인 것을 알아야 하는데 이를 위해 semaphore를 이용한다. 우선 child의 tid를 갖고 있으므로 child의 tid를 이용하여 tid\_thread로 child thread를 찾아준다. 만약 그러한 thread를 찾지 못한다면 -1을 리턴하고 찾았다면 sema\_down을 통해 child thread가 종료되기를 기다린다. child가 process\_exit을 통해 sema\_up으로 종료를 알려주면 child의 exit system call로부터 받은 child의 종료 status인 thread\_current()->child\_status를 리턴해준다.

int process\_wait (tid\_t child\_tid) {

struct thread \* child = NULL;

int ret\_stat;

child = tid\_thread(child\_tid);

if(child == NULL) return -1;

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

ret\_stat = thread\_current()->child\_status;

return ret\_stat;

}

**void process\_exit(void)**

process\_exit은 현재 process를 종료하는 함수인데 child가 이 함수를 진행할 때, 종료한다는 신호를 parent thread에게 sema\_up을 통해 알려준다.

void process\_exit (void) {

…

sema\_up(&(cur->child\_sync)); // 함수의 마지막 부분

}

**/threads/thread.h**

process\_wait에서 semaphore를 이용하여 child thread의 종료를 기다리기 위해 thread struct 내부에 semaphore인 child\_sync와 child thread 의 종료 시 status를 알기 위한 child\_status와 해당 thread의 부모 thread 를 찾기 위해 부모의 tid 값인 par\_tid 를 추가해준다.

또, tid 값을 이용해서 thread 를 찾는 함수 tid\_thread를 선언해준다.

#include “threads/sync.h”

struct thread {

…

#ifdef USERPROG

uint32\_t \*pagedir;

struct semaphore child\_sync;

int child\_status;

tid\_t par\_tid;

#endif

…

}

struct thread \* tid\_thread(tid\_t tid);

**/threads/thread.c**

**static void init\_thread()**

새롭게 추가해준 child\_sync를 thread를 initialize 하는 과정에 sema\_init 해준다.

static void init\_thread {

…

list\_push\_back (&all\_list, &t->allele);

#ifdef USERPROG

seam\_init(&(t->child\_sync), 0);

#endif

}

**struct thread \* tid\_thread (tid\_t num)**

tid\_thread는 tid를 이용하여 thread를 찾는 함수로 all\_list에 모든 thread가 정의 되어 있으므로 list\_iterator를 이용하여 all\_list 의 begin부터 end까지 탐색을 한다. 탐색을 하면서 iterator가 가리키는 entry 값의 tid가 인자로 받은 tid 가 같다면 해당 thread를 찾은 것으로 그대로 리턴해준다. for문이 종료된다면 해당 thread가 없기 때문에 NULL을 리턴한다.

struct thread \* tid\_thread (tid\_t num) {

struct list\_elem \*list\_it;

struct thread \* child;

for (list\_it = list\_begin(&all\_list); list\_it != list\_end(&all\_list); list\_it = list\_next(list\_it)) {

child = list\_entry(list\_it, struct thread, allelem);

if (child->tid == num) {

return child;

}

}

return NULL;

}

**/userprog/syscall.h**

system call 을 위해 halt, exit, exec, wait, read, write 와 추가 구현 fibonacci, sum\_of\_four\_int 함수를 선언해준다.

void halt(void);

void exit(int);

pid\_t exec (const char \*);

int wait (pid\_t);

int read (int, void, unsigned);

int write (int, const void \*, unsigned);

int fibonacci (int);

it sum\_of\_four\_int (int, int, int, int);

**/userprog/syscall.c**

system call 을 위해 halt, exit, exec, wait, read, write 와 추가 구현 fibonacci, sum\_of\_four\_int 함수를 정의해준다.

**void halt (void)**

pintos를 종료시키는 system call로 shutdown\_power\_off() 함수를 호출해준다.

void halt (void) {

shutdown\_power\_off();

}

**void exit (int status)**

exit 함수는 현재 thread를 종료시킨다. cur는 현재 실행중인 thread로 child이고, par은 현재 thread의 부모 thread를 가리킨다. 부모 thread는 자식 thread의 종료를 기다리고, 기다리고 나서 반환 값으로 자식 thread의 종료 시 status를 갖기 때문에 par의 child\_thread를 현재 종료하는 child의 종료 status로 지정해준다. 그 후, 현재 종료하는 thread의 이름과 종료 status를 출력해주고 thread를 종료시킨다.

void exit(int status) {

struct thread\* cur, \*par;

int par\_tid;

cur = thread\_current();

par\_tid = cur->par\_tid;

par = tid\_thread(par\_tid);

par->child\_stauts = status;

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

thread\_exit();

}

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

인자로 받은 cmd\_line을 실행하는 system call로 process\_execute(cmd\_line) 함수를 호출하고 이 때, cmd\_line의 thread를 새로 생성하는데 이 thread의 tid값을 리턴한다.

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

return process\_execute(cmd\_line);

}

**int wait (pid\_t pid)**

자식 thread가 종료되기 전 부모 thread가 종료되는 것을 방지하기 위한 함수로써 인자로 받은 pid를 갖는 thread가 끝날 때까지 기다리면 해당 thread가 종료시의 status를 리턴한다.

int wait (pid\_t pid) {

return process\_wait (pid);

}

**int read (int, void \*, unsigned)**

read 함수는 stdin으로부터 읽어준다. 이 때, fd 값은 0이다. for문을 통해 input\_getc로 버퍼에 문자를 담는데 size만큼 다 돌지 않았는데 파일의 마지막이 읽힌 경우 그 때까지 읽은 문자의 리턴하고 size만큼 버퍼에 담았다면 size를 리턴한다. 만약 fd값이 0이 아니라면 -1을 리턴한다.

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

int I;

void \*temp = buffer;

if (fd == 0) {

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

\*(uint8\_t \*)temp = input\_getc();

if (\*(uint8\_t \*)temp++ == ‘\0’) {

break;

}

}

return i;

}

return -1;

}

**int write (int, const void \*, unsigned)**

write 함수는 stdout으로 출력한다. 이 때, fd 값이 1인 경우, putbuf 함수를 통해서 입력받은 buffer를 인자로 받은 size 만큼 출력하고 출력한 사이즈를 리턴한다. fd 값이 1이 아닌 경우에는 -1을 리턴한다.

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

If(fd==1){

putbuf(buffer, size);

return size;

}

return -1;

}

**int fibonacci (int n)**

fibonacci 함수는 n번 째 fibonacci 값을 구해준다.

int fibonacci (int n) {

int a = 0, res = 1, i;

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

int t = res; res += a; a = t;

}

return res;

}

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

sum\_of\_four\_int 함수는 네 수의 합을 구해준다.

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

return a + b + c + d;

}

**static void syscall\_handler (struct intr\_frame \*f)**

interrupt가 들어오면 esp를 확인하여 알맞은 system call을 호출한다. 그리고 리턴값이 있는 syscall은 eax 에 넣어주었다.

switch문 내에서 is\_user\_vaddr() 함수를 사용해서 memory access를 체크하는 부분을 추가했다.

#include “threads/vaddr.h”

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

{

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

case SYS\_HALT:

halt(); break;

case SYS\_EXIT:

if(!is\_user\_vaddr(f->esp+4)) exit(-1);

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

case SYS\_EXEC:

if(!is\_user\_vaddr(f->esp+4)) exit(-1);

f->exec = exec(\*(char \*\*)(f->esp + 4)); break;

case SYS\_WAIT:

if(!is\_user\_vaddr(f->esp+4)) exit(-1);

f->eax = wait(\*(int\*)(f->esp+4)); break;

case SYS\_READ:

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:

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

case SYS\_FIBONACCI:

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

case SYS\_SUM:

f->eax = sum\_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));

}}

**/example/sum.c**

n 번째 fibonacci 값과 네 수의 합을 구하는 Userprogram sum 을 작성하였다.

인자의 개수가 총 5개로 들어왔는지 먼저 파악 후, system call을 통해 fibonacci값과 네 수의 합을 계산 후, 출력해준다.

#include <stdio.h>

#include <stdlib.h>

#include <syscall.h>

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

int a, b, c, d, fibo\_res, sum\_res;

if (argc != 5) {

printf(“usage: sum A B C D\n”);

return EXIT\_FAILURE;

}

a = atoi(argv[1]); b = atoi(argv[2]); c = atoi(argv[3]); d = atoi(argv[4]);

fibo\_res = fibonacci(a);

sum\_res = sum\_of\_four\_int(a, b, c, d);

printf(“%d %d\n”, fibo\_res, sum\_res);

return EXIT\_SUCCESS;

}

**/examples/Makefile**

sum.c의 실행파일이 생성될 수 있도록 make 파일을 수정하였다.

…

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

bubsort insult lineup matmult recursor sum

…

sum\_SRC = sum.c

…

**/lib/syscall-nr.h**

syscall\_handler 에서 fibonacci 와 sum\_of\_four\_int 의 system 호출을 구분하기 위해 추가해준다.

enum {

…

SYS\_FIBONACCI,

SYS\_SUM

};

**/lib/user/syscall.h**

userprog/syscall.c 의 fibonacci 와 sum\_of\_four\_int 함수에 대해 선언하였다.

int fibonacci (int);

int sum\_of\_four\_int (int, int, int, int);

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

sum\_of\_four\_int 는 인자를 4개 받는 system call이기 때문에 해당 system call이 가능하도록 어셈블리어를이용한 syscall4 매크로를 구현해주었다.

#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] “g” (ARG0), \

[arg1] “g” (ARG1), \

[arg2] “g” (ARG2), \

[arg3] “g” (ARG3) \

: “memory”); \

retval; \

})

userprog/syscall.c 의 fibonacci 와 sum\_of\_four\_int 함수에 대해 정의하였다.

int fibonacci (int n) {

return syscall1 (SYS\_FIBONACCI, n);

}

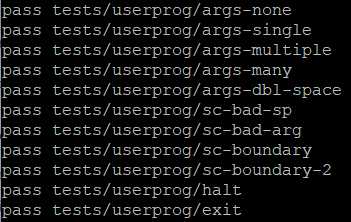
int sum\_of\_four\_int(int a, int b, int c, int d) {

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

}

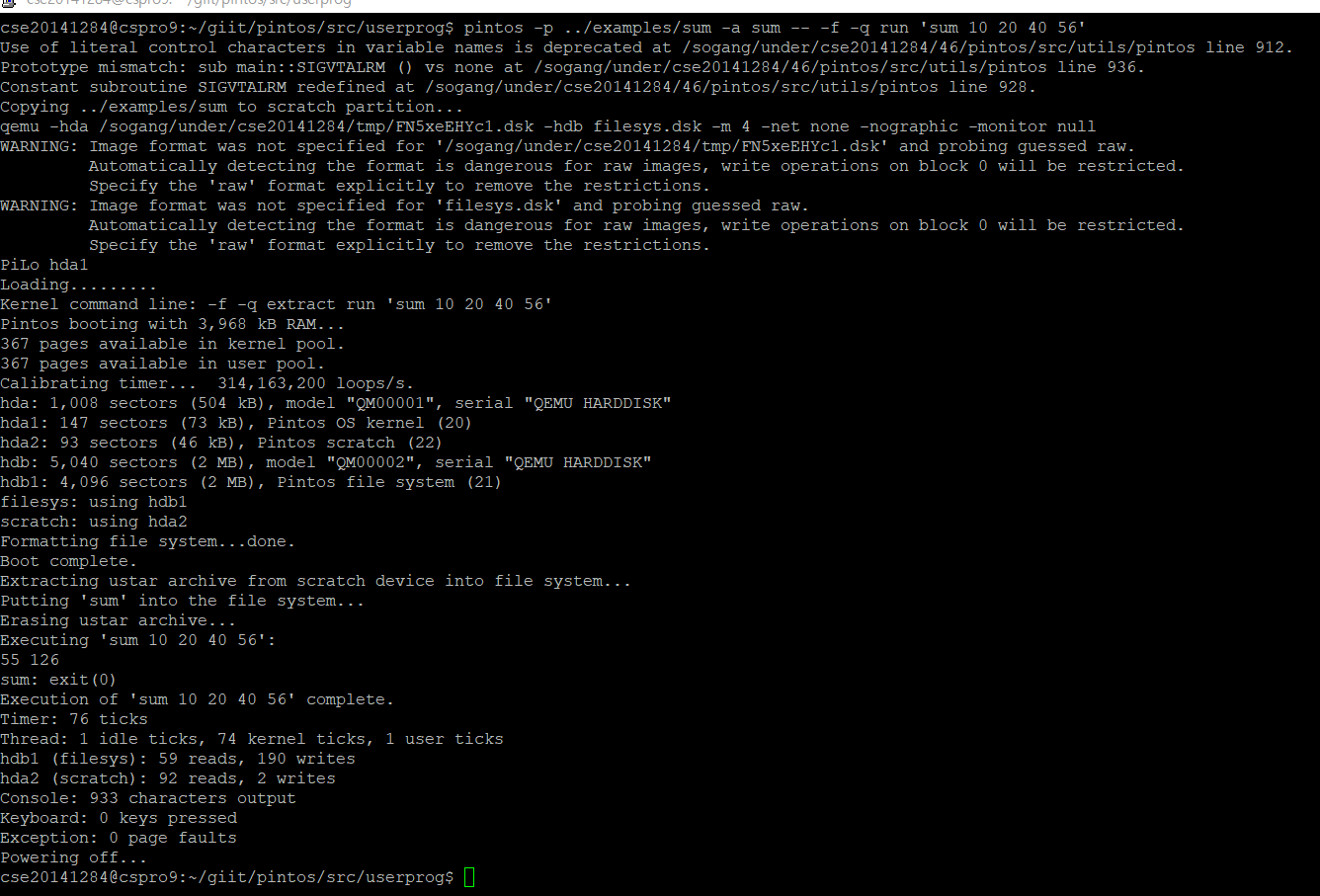
1. **시험 및 평가 내용**

* test 결과

**텍스트, 실내, 병이(가) 표시된 사진

자동 생성된 설명**

* Additional Implementation test 결과



1. **기타**
2. **연구 조원 기여도**

이기현 : 50%

장주호 : 50%

1. **소감**

이기현 : 처음 프로젝트 파일을 열었을 때는 아무것도 할 수가 없었다. 매뉴얼을 2번 정도 보고 전체강의 녹음을 몇 번은 더 듣고 나서야 뭘 해야 하는지 뭘 건들어야 하는지 대강 알 수 있었다. 생각보다 더 많이 파일간의 의존도가 높아서 코드를 한 줄만 잘못 작성해도 make가 안되는 것을 보고 진땀을 많이 흘렸던 것 같다. 게다가 협업을 하면서 include 한 줄이 있고 없음에 따라 누구는 make가 되고 누구는 make가 되지 않아 프로젝트를 진행하기에 있어 더 힘들었다. 그래도 매뉴얼을 여러 번 보고 함수의 의미를 파악하면서 진행방향을 잡을 수 있어 그나마 편했고 다음 프로젝트에서는 무엇에 유의해야 할 지 빠르게 파악할 수 있을 것 같다.

장주호 : 이전에 들었던 다른 수업들처럼 무작정 코딩을 하면서 시작할 경우 교착상태에 빠질 수 있다는 것을 배웠다.

무작정 프로젝트에 손을 댈 경우에는 다른 부분에서의 문제점이 발생할 수 있기 때문에 신중히 접근했어야 했으며 복합적이고, 상호 의존적인 프로젝트에 대한 경험이 없었기 때문에 흥미로웠던 한편, 어려움을 많이 겪었다. 매번 벽에 부딪힐 때 마다 조원과 상의를 통해서 닥친 문제들을 해결해 나갔으며 프로젝트 팀의 중요성을 느낄 수 있는 계기가 되었다.